

MPI: A Message-Passing Interface Standard

Version 4.1

(Draft)

Unofficial, for comment only

Message Passing Interface Forum

February 20, 2023

DRAFT

1 This document describes the 2022 Draft Specification of the Message-Passing Interface
2 (MPI) standard, intended for comment. This is a working document; the MPI Forum
3 continues to work on MPI-4.1 and expects additional changes before MPI-4.1 is finalized. It
4 is not an official version of the standard.

5 The MPI standard includes point-to-point message-passing, collective communications,
6 group and communicator concepts, process topologies, environmental management, process
7 creation and management, one-sided communications, extended collective operations, ex-
8 ternal interfaces, I/O, some miscellaneous topics, and multiple tool interfaces. Language
9 bindings for C and Fortran are defined.

10 Historically, the evolution of the standard is:

- 11 • MPI-1.0 (May 5, 1994): Initial release.
- 12
- 13 • MPI-1.1 (June 12, 1995): Minor updates and bug fixes.
- 14
- 15 • MPI-1.2 (July 18, 1997): Several clarifications and additions.
- 16
- 17 • MPI-2.0 (July 18, 1997): New functionality and all the clarifications and additions
18 from MPI-1.2.
- 19 • MPI-1.3 (May 30, 2008): For historical reasons, combining the MPI-1.1, MPI-1.2, and
20 several errata documents into one combined document.
- 21
- 22 • MPI-2.1 (June 23, 2008): Combining the previous documents.
- 23
- 24 • MPI-2.2 (September 4, 2009): Additional clarifications and seven new routines.
- 25
- 26 • MPI-3.0 (September 21, 2012): Extension of MPI-2.2.
- 27
- 28 • MPI-3.1 (June 4, 2015): Clarifications and minor extensions to MPI-3.0.
- 29
- 30 • MPI-4.0 (June 9, 2021): Significant new features beyond MPI-3.1.

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

- 32 1. Subscribe to <https://lists.mpi-forum.org/mailman/listinfo/mpi-comments>
- 33
- 34 2. Send your comment to: mpi-comments@lists.mpi-forum.org, together with the version
35 of the MPI standard and the page and line numbers on which you are commenting.

36 Your comment will be forwarded to MPI Forum committee members for consideration.
37 Messages sent from an unsubscribed e-mail address will not be considered.
38

39
40
41
42
43
44
45 ©1993, 1994, 1995, 1996, 1997, 2008, 2009, 2012, 2015, 2021 University of Tennessee,
46 Knoxville, Tennessee. Permission to copy without fee all or part of this material is granted,
47 provided the University of Tennessee copyright notice and the title of this document appear,
48 and notice is given that copying is by permission of the University of Tennessee.

2022 Draft Specification, XXXX, 2022. This document contains a draft of the MPI specification as of the date of publication. It has not been adopted as an official MPI specification, and is provided for comment only. This document includes updates and new features that will be present in the final MPI-4.1 document.

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

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

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

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

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

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

Version 2.0: July 18, 1997. Beginning after the release of MPI-1.1, the MPI Forum began meeting to consider corrections and extensions. MPI-2 has been focused on process creation and management, one-sided communications, extended collective communications, external

1 interfaces and parallel I/O. A miscellany chapter discusses items that do not fit elsewhere,
2 in particular language interoperability.

3
4 Version 1.2: July 18, 1997. The MPI-2 Forum introduced MPI-1.2 as Chapter 3 in the
5 standard “MPI-2: Extensions to the Message-Passing Interface”, July 18, 1997. This section
6 contains clarifications and minor corrections to Version 1.1 of the MPI Standard. The only
7 new function in MPI-1.2 is one for identifying to which version of the MPI Standard the
8 implementation conforms. There are small differences between MPI-1 and MPI-1.1. There
9 are very few differences between MPI-1.1 and MPI-1.2, but large differences between MPI-1.2
10 and MPI-2.

11
12 Version 1.1: June, 1995. Beginning in March, 1995, the Message-Passing Interface Forum
13 reconvened to correct errors and make clarifications in the MPI document of May 5, 1994,
14 referred to below as Version 1.0. These discussions resulted in Version 1.1. The changes
15 from Version 1.0 are minor. A version of this document with all changes marked is available.

16
17 Version 1.0: May, 1994. The Message-Passing Interface Forum, with participation from
18 over 40 organizations, has been meeting since January 1993 to discuss and define a set of
19 library interface standards for message passing. The Message-Passing Interface Forum is
20 not sanctioned or supported by any official standards organization.

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

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

Contents

List of Figures	xix
List of Tables	xxi
Acknowledgments	xxiii
1 Introduction to MPI	1
1.1 Overview and Goals	1
1.2 Background of MPI-1.0	2
1.3 Background of MPI-1.1, MPI-1.2, and MPI-2.0	2
1.4 Background of MPI-1.3 and MPI-2.1	3
1.5 Background of MPI-2.2	4
1.6 Background of MPI-3.0	4
1.7 Background of MPI-3.1	4
1.8 Background of MPI-4.0	4
1.9 Background of 2022 Draft Specification	5
1.10 Who Should Use This Standard?	5
1.11 What Platforms Are Targets for Implementation?	5
1.12 What Is Included in the Standard?	5
1.13 Organization of This Document	6
2 MPI Terms and Conventions	9
2.1 Document Notation	9
2.2 Naming Conventions	9
2.3 Procedure Specification	10
2.4 Semantic Terms	11
2.4.1 MPI Operations	11
2.4.2 MPI Procedures	12
2.4.3 MPI Datatypes	15
2.5 Datatypes	15
2.5.1 Opaque Objects	15
2.5.2 Array Arguments	17
2.5.3 State	18
2.5.4 Named Constants	18
2.5.5 Choice	19
2.5.6 Absolute Addresses and Relative Address Displacements	19
2.5.7 File Offsets	20
2.5.8 Counts	20
2.6 Language Binding	21
2.6.1 Deprecated and Removed Interfaces	21
2.6.2 Fortran Binding Issues	22
2.6.3 C Binding Issues	23

2.6.4	Functions and Macros	24
2.7	Processes	24
2.8	Error Handling	24
2.9	Implementation Issues	26
2.9.1	Independence of Basic Runtime Routines	26
2.9.2	Interaction with Signals	26
2.10	Examples	27
3	Point-to-Point Communication	29
3.1	Introduction	29
3.2	Blocking Send and Receive Operations	30
3.2.1	Blocking Send	30
3.2.2	Message Data	31
3.2.3	Message Envelope	33
3.2.4	Blocking Receive	34
3.2.5	Return Status	36
3.2.6	Passing MPI_STATUS_IGNORE for Status	39
3.2.7	Blocking Send-Receive	39
3.3	Datatype Matching and Data Conversion	42
3.3.1	Type Matching Rules	42
	Type MPI_CHARACTER	44
3.3.2	Data Conversion	45
3.4	Communication Modes	46
3.5	Semantics of Point-to-Point Communication	51
3.6	Buffer Allocation and Usage	55
3.6.1	Model Implementation of Buffered Mode	57
3.7	Nonblocking Communication	58
3.7.1	Communication Request Objects	59
3.7.2	Communication Initiation	60
3.7.3	Communication Completion	67
3.7.4	Semantics of Nonblocking Communications	71
3.7.5	Multiple Completions	72
3.7.6	Non-Destructive Test of status	79
3.8	Probe and Cancel	80
3.8.1	Probe	80
3.8.2	Matching Probe	83
3.8.3	Matched Receives	85
3.8.4	Cancel	88
3.9	Persistent Communication Requests	90
3.10	Null Processes	96
4	Partitioned Point-to-Point Communication	99
4.1	Introduction	99
4.2	Semantics of Partitioned Point-to-Point Communication	100
4.2.1	Communication Initialization and Starting with Partitioning	102
4.2.2	Communication Completion under Partitioning	106
4.2.3	Semantics of Communications in Partitioned Mode	107

4.3	Partitioned Communication Examples	108
4.3.1	Partition Communication with Threads/Tasks Using OpenMP 4.0 or later	108
4.3.2	Send-only Partitioning Example with Tasks and OpenMP version 4.0 or later	109
4.3.3	Send and Receive Partitioning Example with OpenMP version 4.0 or later	110
5	Datatypes	113
5.1	Derived Datatypes	113
5.1.1	Type Constructors with Explicit Addresses	115
5.1.2	Datatype Constructors	115
5.1.3	Subarray Datatype Constructor	127
5.1.4	Distributed Array Datatype Constructor	129
5.1.5	Address and Size Functions	134
5.1.6	Lower-Bound and Upper-Bound Markers	138
5.1.7	Extent and Bounds of Datatypes	140
5.1.8	True Extent of Datatypes	142
5.1.9	Commit and Free	144
5.1.10	Duplicating a Datatype	145
5.1.11	Use of General Datatypes in Communication	146
5.1.12	Correct Use of Addresses	149
5.1.13	Decoding a Datatype	150
5.1.14	Examples	158
5.2	Pack and Unpack	166
5.3	Canonical MPI_PACK and MPI_UNPACK	173
6	Collective Communication	177
6.1	Introduction and Overview	177
6.2	Communicator Argument	180
6.2.1	Specifics for Intra-Communicator Collective Operations	180
6.2.2	Applying Collective Operations to Inter-Communicators	181
6.2.3	Specifics for Inter-Communicator Collective Operations	183
6.3	Barrier Synchronization	183
6.4	Broadcast	184
6.4.1	Example using MPI_BCAST	185
6.5	Gather	186
6.5.1	Examples using MPI_GATHER, MPI_GATHERV	190
6.6	Scatter	196
6.6.1	Examples using MPI_SCATTER, MPI_SCATTERV	200
6.7	Gather-to-all	202
6.7.1	Example using MPI_ALLGATHER	205
6.8	All-to-All Scatter/Gather	206
6.9	Global Reduction Operations	212
6.9.1	Reduce	212
6.9.2	Predefined Reduction Operations	214
6.9.3	Signed Characters and Reductions	217
6.9.4	MINLOC and MAXLOC	217

6.9.5	User-Defined Reduction Operations	221
	Example of User-Defined Reduce	225
6.9.6	All-Reduce	226
6.9.7	Process-Local Reduction	228
6.10	Reduce-Scatter	230
6.10.1	MPI_REDUCE_SCATTER_BLOCK	230
6.10.2	MPI_REDUCE_SCATTER	231
6.11	Scan	233
6.11.1	Inclusive Scan	233
6.11.2	Exclusive Scan	234
6.11.3	Example using MPI_SCAN	235
6.12	Nonblocking Collective Operations	237
6.12.1	Nonblocking Barrier Synchronization	239
6.12.2	Nonblocking Broadcast	240
	Example using MPI_IBCAST	241
6.12.3	Nonblocking Gather	241
6.12.4	Nonblocking Scatter	244
6.12.5	Nonblocking Gather-to-all	246
6.12.6	Nonblocking All-to-All Scatter/Gather	249
6.12.7	Nonblocking Reduce	253
6.12.8	Nonblocking All-Reduce	254
6.12.9	Nonblocking Reduce-Scatter with Equal Blocks	256
6.12.10	Nonblocking Reduce-Scatter	257
6.12.11	Nonblocking Inclusive Scan	258
6.12.12	Nonblocking Exclusive Scan	259
6.13	Persistent Collective Operations	260
6.13.1	Persistent Barrier Synchronization	262
6.13.2	Persistent Broadcast	262
6.13.3	Persistent Gather	263
6.13.4	Persistent Scatter	266
6.13.5	Persistent Gather-to-all	269
6.13.6	Persistent All-to-All Scatter/Gather	272
6.13.7	Persistent Reduce	276
6.13.8	Persistent All-Reduce	277
6.13.9	Persistent Reduce-Scatter with Equal Blocks	279
6.13.10	Persistent Reduce-Scatter	280
6.13.11	Persistent Inclusive Scan	281
6.13.12	Persistent Exclusive Scan	282
6.14	Correctness	283
7	Groups, Contexts, Communicators, and Caching	291
7.1	Introduction	291
7.1.1	Features Needed to Support Libraries	291
7.1.2	MPI's Support for Libraries	291
7.2	Basic Concepts	293
7.2.1	Groups	294
7.2.2	Contexts	294
7.2.3	Intra-Communicators	295

7.2.4	Predefined Intra-Communicators	295
7.3	Group Management	296
7.3.1	Group Accessors	296
7.3.2	Group Constructors	298
7.3.3	Group Destructors	304
7.4	Communicator Management	305
7.4.1	Communicator Accessors	305
7.4.2	Communicator Constructors	307
7.4.3	Communicator Destructors	324
7.4.4	Communicator Info	324
7.5	Motivating Examples	327
7.5.1	Current Practice #1	327
7.5.2	Current Practice #2	328
7.5.3	(Approximate) Current Practice #3	328
7.5.4	Communication Safety Example	329
7.5.5	Library Example #1	330
7.5.6	Library Example #2	331
7.6	Inter-Communication	333
7.6.1	Inter-Communicator Accessors	335
7.6.2	Inter-Communicator Operations	337
7.6.3	Inter-Communication Examples	340
	Example 1: Three-Group “Pipeline”	340
	Example 2: Three-Group “Ring”	342
7.7	Caching	343
7.7.1	Functionality	344
7.7.2	Communicators	345
7.7.3	Windows	350
7.7.4	Datatypes	354
7.7.5	Error Class for Invalid Keyval	357
7.7.6	Attributes Example	358
7.8	Naming Objects	360
7.9	Formalizing the Loosely Synchronous Model	364
7.9.1	Basic Statements	364
7.9.2	Models of Execution	364
	Static Communicator Allocation	365
	Dynamic Communicator Allocation	365
	The General Case	365
8	Process Topologies	367
8.1	Introduction	367
8.2	Virtual Topologies	368
8.3	Embedding in MPI	368
8.4	Overview of the Functions	369
8.5	Topology Constructors	370
8.5.1	Cartesian Constructor	370
8.5.2	Cartesian Convenience Function: <code>MPI_DIMS_CREATE</code>	371
8.5.3	Graph Constructor	372
8.5.4	Distributed Graph Constructor	374

8.5.5	Topology Inquiry Functions	381
8.5.6	Cartesian Shift Coordinates	389
8.5.7	Partitioning of Cartesian Structures	391
8.5.8	Low-Level Topology Functions	392
8.6	Neighborhood Collective Communication	394
8.6.1	Neighborhood Gather	395
8.6.2	Neighborhood Alltoall	399
8.7	Nonblocking Neighborhood Communication	406
8.7.1	Nonblocking Neighborhood Gather	406
8.7.2	Nonblocking Neighborhood Alltoall	409
8.8	Persistent Neighborhood Communication	413
8.8.1	Persistent Neighborhood Gather	413
8.8.2	Persistent Neighborhood Alltoall	416
8.9	An Application Example	421
9	MPI Environmental Management	425
9.1	Implementation Information	425
9.1.1	Version Inquiries	425
9.1.2	Environmental Inquiries	426
	Tag Values	427
	Host Rank	427
	IO Rank	427
	Clock Synchronization	428
	Inquire Processor Name	428
9.2	Memory Allocation	429
9.3	Error Handling	432
9.3.1	Error Handlers for Communicators	434
9.3.2	Error Handlers for Windows	436
9.3.3	Error Handlers for Files	438
9.3.4	Error Handlers for Sessions	440
9.3.5	Freeing Errorhandlers and Retrieving Error Strings	442
9.4	Error Codes and Classes	443
9.5	Error Classes, Error Codes, and Error Handlers	446
9.6	Timers and Synchronization	450
10	The Info Object	453
11	Process Initialization, Creation, and Management	461
11.1	Introduction	461
11.2	The World Model	462
11.2.1	Starting MPI Processes	462
11.2.2	Finalizing MPI	468
11.2.3	Determining Whether MPI Has Been Initialized When Using the World Model	471
11.2.4	Allowing User Functions at MPI Finalization	472
11.3	The Sessions Model	472
11.3.1	Session Creation and Destruction Methods	473
11.3.2	Processes Sets	476

11.3.3	Runtime Query Functions	477
11.3.4	Sessions Model Examples	480
11.4	Common Elements of Both Process Models	485
11.4.1	MPI Functionality that is Always Available	485
11.4.2	Aborting MPI Processes	485
11.5	Portable MPI Process Startup	487
11.6	MPI and Threads	489
11.6.1	General	490
11.6.2	Clarifications	491
11.7	The Dynamic Process Model	493
11.7.1	Starting Processes	493
11.7.2	The Runtime Environment	493
11.8	Process Manager Interface	494
11.8.1	Processes in MPI	494
11.8.2	Starting Processes and Establishing Communication	494
11.8.3	Starting Multiple Executables and Establishing Communication	499
11.8.4	Reserved Keys	502
11.8.5	Spawn Example	503
11.9	Establishing Communication	505
11.9.1	Names, Addresses, Ports, and All That	505
11.9.2	Server Routines	506
11.9.3	Client Routines	509
11.9.4	Name Publishing	510
11.9.5	Reserved Key Values	512
11.9.6	Client/Server Examples	513
11.10	Other Functionality	515
11.10.1	Universe Size	515
11.10.2	Singleton MPI Initialization	516
11.10.3	MPI_APPNUM	516
11.10.4	Releasing Connections	517
11.10.5	Another Way to Establish MPI Communication	519
12	One-Sided Communications	521
12.1	Introduction	521
12.2	Initialization	522
12.2.1	Window Creation	522
12.2.2	Window That Allocates Memory	525
12.2.3	Window That Allocates Shared Memory	527
12.2.4	Window of Dynamically Attached Memory	531
12.2.5	Window Destruction	535
12.2.6	Window Attributes	535
12.2.7	Window Info	537
12.3	Communication Calls	539
12.3.1	Put	539
12.3.2	Get	542
12.3.3	Examples for Communication Calls	543
12.3.4	Accumulate Functions	545
Accumulate		545

Get Accumulate	548
Fetch and Op	550
Compare and Swap	551
12.3.5 Request-based RMA Communication Operations	552
12.4 Memory Model	559
12.5 Synchronization Calls	560
12.5.1 Fence	563
12.5.2 General Active Target Synchronization	565
12.5.3 Lock	569
12.5.4 Flush and Sync	573
12.5.5 Assertions	575
12.5.6 Miscellaneous Clarifications	576
12.6 Error Handling	577
12.6.1 Error Handlers	577
12.6.2 Error Classes	577
12.7 Semantics and Correctness	577
12.7.1 Atomicity	585
12.7.2 Ordering	586
12.7.3 Progress	586
12.7.4 Registers and Compiler Optimizations	588
12.8 Examples	589
13 External Interfaces	599
13.1 Introduction	599
13.2 Generalized Requests	599
13.2.1 Examples	603
13.3 Associating Information with Status	605
14 I/O	609
14.1 Introduction	609
14.1.1 Definitions	609
14.2 File Manipulation	611
14.2.1 Opening a File	611
14.2.2 Closing a File	613
14.2.3 Deleting a File	614
14.2.4 Resizing a File	615
14.2.5 Preallocating Space for a File	616
14.2.6 Querying the Size of a File	616
14.2.7 Querying File Parameters	617
14.2.8 File Info	618
Reserved File Hints	620
14.3 File Views	622
14.4 Data Access	625
14.4.1 Data Access Routines	625
Positioning	626
Synchronism	626
Coordination	627
Data Access Conventions	627

14.4.2	Data Access with Explicit Offsets	628
14.4.3	Data Access with Individual File Pointers	635
14.4.4	Data Access with Shared File Pointers	645
	Noncollective Operations	646
	Collective Operations	649
	Seek	651
14.4.5	Split Collective Data Access Routines	653
14.5	File Interoperability	661
14.5.1	Datatypes for File Interoperability	663
14.5.2	External Data Representation: "external32"	665
14.5.3	User-Defined Data Representations	666
	Extent Callback	669
	Datarep Conversion Functions	670
14.5.4	Matching Data Representations	673
14.6	Consistency and Semantics	673
14.6.1	File Consistency	673
14.6.2	Random Access vs. Sequential Files	676
14.6.3	Progress	677
14.6.4	Collective File Operations	677
14.6.5	Nonblocking Collective File Operations	677
14.6.6	Type Matching	678
14.6.7	Miscellaneous Clarifications	678
14.6.8	MPI_Offset Type	678
14.6.9	Logical vs. Physical File Layout	679
14.6.10	File Size	679
14.6.11	Examples	679
	Asynchronous I/O	682
14.7	I/O Error Handling	683
14.8	I/O Error Classes	684
14.9	Examples	684
14.9.1	Double Buffering with Split Collective I/O	684
14.9.2	Subarray Filetype Constructor	687
15	Tool Support	689
15.1	Introduction	689
15.2	Profiling Interface	689
15.2.1	Requirements	689
15.2.2	Discussion	690
15.2.3	Logic of the Design	690
15.2.4	Miscellaneous Control of Profiling	691
15.2.5	MPI Library Implementation	692
15.2.6	Complications	693
	Multiple Counting	693
	Linker Oddities	694
	Fortran Support Methods	694
15.2.7	Multiple Levels of Interception	694
15.3	The MPI Tool Information Interface	695
15.3.1	Verbosity Levels	696

15.3.2	Binding MPI Tool Information Interface Variables to MPI Objects	696
15.3.3	Convention for Returning Strings	697
15.3.4	Initialization and Finalization	698
15.3.5	Datatype System	699
15.3.6	Control Variables	701
	Control Variable Query Functions	702
	Handle Allocation and Deallocation	705
	Control Variable Access Functions	707
15.3.7	Performance Variables	708
	Performance Variable Classes	708
	Performance Variable Query Functions	710
	Performance Experiment Sessions	713
	Handle Allocation and Deallocation	714
	Starting and Stopping of Performance Variables	715
	Performance Variable Access Functions	716
15.3.8	Events	721
	Event Sources	721
	Callback Safety Requirements	723
	Event Type Query Functions	725
	Handle Allocation and Deallocation	727
	Handling Dropped Events	732
	Reading Event Data	733
	Reading Event Meta Data	734
15.3.9	Variable Categorization	736
	Category Query Functions	736
	Category Member Query Functions	738
15.3.10	Return Codes for the MPI Tool Information Interface	740
15.3.11	Profiling Interface	740
16	Deprecated Interfaces	743
16.1	Deprecated since MPI-2.0	743
16.2	Deprecated since MPI-2.2	746
16.3	Deprecated since MPI-4.0	747
17	Removed Interfaces	751
17.1	Removed MPI-1 Bindings	751
17.1.1	Overview	751
17.1.2	Removed MPI-1 Functions	751
17.1.3	Removed MPI-1 Datatypes	751
17.1.4	Removed MPI-1 Constants	751
17.1.5	Removed MPI-1 Callback Prototypes	751
17.2	C++ Bindings	752
18	Semantic Changes and Warnings	753
18.1	Semantic Changes	753
18.1.1	Semantic Changes Starting in MPI-4.0	753
18.2	Additional Warnings	753
18.2.1	Warnings Starting in MPI-4.0	753

19 Language Bindings	755
19.1 Support for Fortran	755
19.1.1 Overview	755
19.1.2 Fortran Support Through the <code>mpi_f08</code> Module	756
19.1.3 Fortran Support Through the <code>mpi</code> Module	759
19.1.4 Fortran Support Through the <code>mpif.h</code> Include File	761
19.1.5 Interface Specifications, Procedure Names, and the Profiling Interface	762
19.1.6 MPI for Different Fortran Standard Versions	767
19.1.7 Requirements on Fortran Compilers	770
19.1.8 Additional Support for Fortran Register-Memory-Synchronization	772
19.1.9 Additional Support for Fortran Numeric Intrinsic Types	773
Parameterized Datatypes with Specified Precision and Exponent Range	773
Support for Size-specific MPI Datatypes	777
Communication With Size-specific Types	779
19.1.10 Problems With Fortran Bindings for MPI	780
19.1.11 Problems Due to Strong Typing	782
19.1.12 Problems Due to Data Copying and Sequence Association with Subscript Triplets	783
19.1.13 Problems Due to Data Copying and Sequence Association with Vector Subscripts	785
19.1.14 Special Constants	786
19.1.15 Fortran Derived Types	786
19.1.16 Optimization Problems, an Overview	788
19.1.17 Problems with Code Movement and Register Optimization	789
Nonblocking Operations	789
Persistent Operations	790
One-sided Communication	790
MPI_BOTTOM and Combining Independent Variables in Datatypes	790
Solutions	791
The Fortran ASYNCHRONOUS Attribute	792
Calling MPI_F_SYNC_REG	793
A User Defined Routine Instead of MPI_F_SYNC_REG	794
Module Variables and COMMON Blocks	795
The (Poorly Performing) Fortran VOLATILE Attribute	795
The Fortran TARGET Attribute	795
19.1.18 Temporary Data Movement and Temporary Memory Modification	796
19.1.19 Permanent Data Movement	798
19.1.20 Comparison with C	799
19.2 Support for Large Count and Large Byte Displacement	800
19.3 Language Interoperability	801
19.3.1 Introduction	801
19.3.2 Assumptions	801
19.3.3 Initialization	802
Concerns specific to the World Model	802
Concerns specific to the Sessions Model	802
Concerns common to both the World Model and the Sessions Model	803

19.3.4	Transfer of Handles	803
19.3.5	Status	805
19.3.6	MPI Opaque Objects	807
	Datatypes	808
	Callback Functions	809
	Error Handlers	809
	Reduce Operations	810
19.3.7	Attributes	810
19.3.8	Extra-State	813
19.3.9	Constants	814
19.3.10	Interlanguage Communication	814
A	Language Bindings Summary	817
A.1	Defined Values and Handles	817
A.1.1	Defined Constants	817
A.1.2	Types	831
A.1.3	Prototype Definitions	832
	C Bindings	832
	Fortran 2008 Bindings with the <code>mpi_f08</code> Module	833
	Fortran Bindings with <code>mpif.h</code> or the <code>mpi</code> Module	836
A.1.4	Deprecated Prototype Definitions	838
A.1.5	String Values	839
	Default Communicator Names	839
	Default Datatype Names	839
	Default Window Names	839
	Reserved Data Representations	839
	Process Set Names	839
	Info Keys	840
	Info Values	841
A.2	Summary of the Semantics of all Op.-Related Routines	841
A.3	C Bindings	842
A.3.1	Point-to-Point Communication C Bindings	842
A.3.2	Partitioned Communication C Bindings	845
A.3.3	Datatypes C Bindings	845
A.3.4	Collective Communication C Bindings	849
A.3.5	Groups, Contexts, Communicators, and Caching C Bindings	857
A.3.6	Process Topologies C Bindings	860
A.3.7	MPI Environmental Management C Bindings	864
A.3.8	The Info Object C Bindings	865
A.3.9	Process Creation and Management C Bindings	865
A.3.10	One-Sided Communications C Bindings	867
A.3.11	External Interfaces C Bindings	870
A.3.12	I/O C Bindings	870
A.3.13	Language Bindings C Bindings	874
A.3.14	Tools / Profiling Interface C Bindings	875
A.3.15	Tools / MPI Tool Information Interface C Bindings	875
A.3.16	Deprecated C Bindings	878

A.4	Fortran 2008 Bindings with the <code>mpi_f08</code> Module	879
A.4.1	Point-to-Point Communication Fortran 2008 Bindings	879
A.4.2	Partitioned Communication Fortran 2008 Bindings	889
A.4.3	Datatypes Fortran 2008 Bindings	890
A.4.4	Collective Communication Fortran 2008 Bindings	897
A.4.5	Groups, Contexts, Communicators, and Caching Fortran 2008 Bindings	919
A.4.6	Process Topologies Fortran 2008 Bindings	927
A.4.7	MPI Environmental Management Fortran 2008 Bindings	936
A.4.8	The Info Object Fortran 2008 Bindings	939
A.4.9	Process Creation and Management Fortran 2008 Bindings	940
A.4.10	One-Sided Communications Fortran 2008 Bindings	943
A.4.11	External Interfaces Fortran 2008 Bindings	950
A.4.12	I/O Fortran 2008 Bindings	951
A.4.13	Language Bindings Fortran 2008 Bindings	964
A.4.14	Tools / Profiling Interface Fortran 2008 Bindings	965
A.4.15	Deprecated Fortran 2008 Bindings	965
A.5	Fortran Bindings with <code>mpif.h</code> or the <code>mpi</code> Module	966
A.5.1	Point-to-Point Communication Fortran Bindings	966
A.5.2	Partitioned Communication Fortran Bindings	969
A.5.3	Datatypes Fortran Bindings	969
A.5.4	Collective Communication Fortran Bindings	972
A.5.5	Groups, Contexts, Communicators, and Caching Fortran Bindings	977
A.5.6	Process Topologies Fortran Bindings	982
A.5.7	MPI Environmental Management Fortran Bindings	985
A.5.8	The Info Object Fortran Bindings	987
A.5.9	Process Creation and Management Fortran Bindings	988
A.5.10	One-Sided Communications Fortran Bindings	990
A.5.11	External Interfaces Fortran Bindings	994
A.5.12	I/O Fortran Bindings	994
A.5.13	Language Bindings Fortran Bindings	999
A.5.14	Tools / Profiling Interface Fortran Bindings	999
A.5.15	Deprecated Fortran Bindings	999
B	Change-Log	1001
B.1	Changes from Version 4.0 to Version 4.1	1001
B.1.1	Fixes to Errata in Previous Versions of MPI	1001
B.1.2	Changes in MPI-4.1	1002
B.2	Changes from Version 3.1 to Version 4.0	1002
B.2.1	Fixes to Errata in Previous Versions of MPI	1002
B.2.2	Changes in MPI-4.0	1002
B.3	Changes from Version 3.0 to Version 3.1	1006
B.3.1	Fixes to Errata in Previous Versions of MPI	1006
B.3.2	Changes in MPI-3.1	1008
B.4	Changes from Version 2.2 to Version 3.0	1008
B.4.1	Fixes to Errata in Previous Versions of MPI	1008
B.4.2	Changes in MPI-3.0	1009
B.5	Changes from Version 2.1 to Version 2.2	1014

B.6 Changes from Version 2.0 to Version 2.1	1017
Bibliography	1021
General Index	1027
Examples Index	1035
MPI Constant and Predefined Handle Index	1052
MPI Declarations Index	1058
MPI Callback Function Prototype Index	1059
MPI Function Index	1060

DRAFT

List of Figures

2.1	State transition diagram for blocking operations	12
2.2	State transition diagram for nonblocking operations	12
2.3	State transition diagram for persistent operations	13
6.1	Collective communications, an overview	179
6.2	Inter-communicator allgather	182
6.3	Inter-communicator reduce-scatter	183
6.4	Gather example	190
6.5	Gatherv example with strides	191
6.6	Gatherv example, 2-dimensional	192
6.7	Gatherv example, 2-dimensional, subarrays with different sizes	193
6.8	Gatherv example, 2-dimensional, subarrays with different sizes and strides	195
6.9	Scatter example	200
6.10	Scatterv example with strides	201
6.11	Scatterv example with different strides and counts	202
6.12	Race conditions with point-to-point and collective communications	286
6.13	Overlapping communicators example	290
7.1	Inter-communicator creation using <code>MPI_COMM_CREATE</code>	313
7.2	Inter-communicator construction with <code>MPI_COMM_SPLIT</code>	317
7.3	Recursive communicator creation with <code>MPI_COMM_SPLIT_TYPE</code>	322
7.4	Three-group pipeline	341
7.5	Three-group ring	342
8.1	Neighborhood gather communication example	397
8.2	Cartesian neighborhood allgather example for 3 and 1 processes in a dimension	397
8.3	Cartesian neighborhood alltoall example for 3 and 1 MPI processes in a dimension	402
8.4	Set-up of MPI process structure for two-dimensional parallel Poisson solver	421
8.5	Communication routine with local data copying and sparse neighborhood all-to-all	422
8.6	Communication routine with sparse neighborhood all-to-all-w and without local data copying	423
8.7	Two-dimensional parallel Poisson solver with persistent sparse neighborhood all-to-all-w and without local data copying	424
11.1	Session handle to communicator	474
11.2	Process set examples	477
12.1	Schematic description of the public/private window operations in the <code>MPI_WIN_SEPARATE</code> memory model for two overlapping windows	560
12.2	Active target communication	562

12.3 Active target communication, with weak synchronization	563
12.4 Passive target communication	564
12.5 Active target communication with several processes	568
12.6 Symmetric communication	587
12.7 Deadlock situation	587
12.8 No deadlock	588
14.1 Etypes and filetypes	610
14.2 Partitioning a file among parallel processes	610
14.3 Displacements	623
14.4 Example array file layout	687
14.5 Example local array filetype for process 1	687
19.1 Status conversion routines	806

DRAFT

List of Tables

2.1	Deprecated and removed constructs	22
3.1	Predefined MPI datatypes corresponding to Fortran datatypes	31
3.2	Predefined MPI datatypes corresponding to C datatypes	32
3.3	Predefined MPI datatypes corresponding to both C and Fortran datatypes	33
3.4	Predefined MPI datatypes corresponding to C++ datatypes	33
5.1	combiner values returned from MPI_TYPE_GET_ENVELOPE	152
7.1	MPI_COMM_* function behavior (in inter-communication mode)	336
9.1	Error classes (Part 1)	444
9.2	Error classes (Part 2)	445
11.1	List of MPI Functions that can be called at any time within an MPI program, including prior to MPI initialization and following MPI finalization	486
12.1	C types of attribute value argument to MPI_WIN_GET_ATTR and MPI_WIN_SET_ATTR	536
12.2	Error classes in one-sided communication routines	577
14.1	Data access routines	625
14.2	"external32" sizes of predefined datatypes	667
14.3	"external32" sizes of optional datatypes	668
14.4	"external32" sizes of C++ datatypes	668
14.5	I/O error classes	685
15.1	MPI tool information interface verbosity levels	696
15.2	Constants to identify associations of variables	697
15.3	MPI datatypes that can be used by the MPI tool information interface	700
15.4	Scopes for control variables	704
15.5	Hierarchy of safety requirement levels for event callback routines	723
15.6	List of MPI functions that when called from within a callback function may not return MPI_T_ERR_NOT_ACCESSIBLE	724
15.7	Return codes used in functions of the MPI tool information interface	741
17.1	Removed MPI-1 functions and their replacements	751
17.2	Removed MPI-1 datatypes. The indicated routine may be used for changing the lower and upper bound respectively.	752
17.3	Removed MPI-1 constants	752
17.4	Removed MPI-1 callback prototypes and their replacements	752
19.1	Specific Fortran procedure names and related calling conventions	763

19.2 Occurrence of Fortran optimization problems 788

DRAFT

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.

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

13
14 The University of Tennessee and Oak Ridge National Laboratory made the draft avail-
15 able by anonymous FTP mail servers and were instrumental in distributing the document.

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

20 21 MPI-1.2 and MPI-2.0:

22
23 Those who served as primary coordinators in MPI-1.2 and MPI-2.0 are:

- 24
25 • Ewing Lusk, Convener and Meeting Chair
- 26
27 • Steve Huss-Lederman, Editor
- 28
29 • Ewing Lusk, Miscellany
- 30
31 • Bill Saphir, Process Creation and Management
- 32
33 • Marc Snir, One-Sided Communications
- 34
35 • William Gropp and Anthony Skjellum, Extended Collective Operations
- 36
37 • Steve Huss-Lederman, External Interfaces
- 38
39 • Bill Nitzberg, I/O
- 40
41 • Andrew Lumsdaine, Bill Saphir, and Jeffrey M. Squyres, Language Bindings
- 42
43 • Anthony Skjellum and Arkady Kanevsky, Real-Time

44
45 The following list includes some of the active participants who attended MPI-2 Forum
46 meetings and are not mentioned above.
47
48

Greg Astfalk	Robert Babb	Ed Benson	Rajesh Bordawekar	1
Pete Bradley	Peter Brennan	Ron Brightwell	Maciej Brodowicz	2
Eric Brunner	Greg Burns	Margaret Cahir	Pang Chen	3
Ying Chen	Albert Cheng	Yong Cho	Joel Clark	4
Lyndon Clarke	Laurie Costello	Dennis Cottel	Jim Cownie	5
Zhenqian Cui	Suresh Damodaran-Kamal		Raja Daoud	6
Judith Devaney	David DiNucci	Doug Doeffler	Jack Dongarra	7
Terry Dontje	Nathan Doss	Anne Elster	Mark Fallon	8
Karl Feind	Sam Fineberg	Craig Fischberg	Stephen Fleischman	9
Ian Foster	Hubertus Franke	Richard Frost	Al Geist	10
Robert George	David Greenberg	John Hagedorn	Kei Harada	11
Leslie Hart	Shane Hebert	Rolf Hempel	Tom Henderson	12
Alex Ho	Hans-Christian Hoppe	Joefon Jann	Terry Jones	13
Karl Kesselman	Koichi Konishi	Susan Kraus	Steve Kubica	14
Steve Landherr	Mario Lauria	Mark Law	Juan Leon	15
Lloyd Lewins	Ziyang Lu	Bob Madahar	Peter Madams	16
John May	Oliver McBryan	Brian McCandless	Tyce McLarty	17
Thom McMahon	Harish Nag	Nick Nevin	Jarek Nieplocha	18
Ron Oldfield	Peter Ossadnik	Steve Otto	Peter Pacheco	19
Yoonho Park	Perry Partow	Pratap Pattnaik	Elsie Pierce	20
Paul Pierce	Heidi Poxon	Jean-Pierre Prost	Boris Protopopov	21
James Pruyve	Rolf Rabenseifner	Joe Rieken	Peter Rigsbee	22
Tom Robey	Anna Rounbehler	Nobutoshi Sagawa	Arindam Saha	23
Eric Salo	Darren Sanders	Eric Sharakan	Andrew Sherman	24
Fred Shirley	Lance Shuler	A. Gordon Smith	Ian Stockdale	25
David Taylor	Stephen Taylor	Greg Tensa	Rajeev Thakur	26
Marydell Tholburn	Dick Treumann	Simon Tsang	Manuel Ujaldon	27
David Walker	Jerrell Watts	Klaus Wolf	Parkson Wong	28
Dave Wright				29

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	35
Bolt, Beranek, and Newman	36
California Institute of Technology	37
Center for Computing Sciences	38
Convex Computer Corporation	39
Cray Research	40
Digital Equipment Corporation	41
Dolphin Interconnect Solutions, Inc.	42
Edinburgh Parallel Computing Centre	43
General Electric Company	44
German National Research Center for Information Technology	45
Hewlett-Packard	46
Hitachi	47
Hughes Aircraft Company	48

- 1 Intel Corporation
- 2 International Business Machines
- 3 Khoral Research
- 4 Lawrence Livermore National Laboratory
- 5 Los Alamos National Laboratory
- 6 MPI Software Techology, Inc.
- 7 Mississippi State University
- 8 NEC Corporation
- 9 National Aeronautics and Space Administration
- 10 National Energy Research Scientific Computing Center
- 11 National Institute of Standards and Technology
- 12 National Oceanic and Atmospheric Administration
- 13 Oak Ridge National Laboratory
- 14 The Ohio State University
- 15 PALLAS GmbH
- 16 Pacific Northwest National Laboratory
- 17 Pratt & Whitney
- 18 San Diego Supercomputer Center
- 19 Sanders, A Lockheed-Martin Company
- 20 Sandia National Laboratories
- 21 Schlumberger
- 22 Scientific Computing Associates, Inc.
- 23 Silicon Graphics Incorporated
- 24 Sky Computers
- 25 Sun Microsystems Computer Corporation
- 26 Syracuse University
- 27 The MITRE Corporation
- 28 Thinking Machines Corporation
- 29 United States Navy
- 30 University of Colorado
- 31 University of Denver
- 32 University of Houston
- 33 University of Illinois
- 34 University of Maryland
- 35 University of Notre Dame
- 36 University of San Francisco
- 37 University of Stuttgart Computing Center
- 38 University of Wisconsin

39 MPI-2 operated on a very tight budget (in reality, it had no budget when the first
40 meeting was announced). Many institutions helped the MPI-2 effort by supporting the
41 efforts and travel of the members of the MPI Forum. Direct support was given by NSF and
42 DARPA under NSF contract CDA-9115428 for travel by U.S. academic participants and
43 Esprit under project HPC Standards (21111) for European participants.
44

45
46
47
48

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
- William 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:

- William 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.

1	Pavan Balaji	Purushotham V. Bangalore	Brian Barrett
2	Richard Barrett	Christian Bell	Robert Blackmore
3	Gil Bloch	Ron Brightwell	Jeffrey Brown
4	Darius Buntinas	Jonathan Carter	Nathan DeBardeleben
5	Terry Dontje	Gabor Dozsa	Edric Ellis
6	Karl Feind	Edgar Gabriel	Patrick Geoffray
7	David Gingold	Dave Goodell	Erez Haba
8	Robert Harrison	Thomas Herault	Steve Hodson
9	Torsten Hoefler	Joshua Hursey	Yann Kalemkarian
10	Matthew Koop	Quincey Koziol	Sameer Kumar
11	Miron Livny	Kannan Narasimhan	Mark Pagel
12	Avneesh Pant	Steve Poole	Howard Pritchard
13	Craig Rasmussen	Hubert Ritzdorf	Rob Ross
14	Tony Skjellum	Brian Smith	Vinod Tipparaju
15	Jesper Larsson Träff	Keith Underwood	

16
17 The MPI Forum also acknowledges and appreciates the valuable input from people via
18 e-mail and in person.

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

21 Argonne National Laboratory
22 Bull
23 Cisco Systems, Inc.
24 Cray Inc.
25 The HDF Group
26 Hewlett-Packard
27 IBM T.J. Watson Research
28 Indiana University
29 Institut National de Recherche en Informatique et Automatique (Inria)
30 Intel Corporation
31 Lawrence Berkeley National Laboratory
32 Lawrence Livermore National Laboratory
33 Los Alamos National Laboratory
34 Mathworks
35 Mellanox Technologies
36 Microsoft
37 Myricom
38 NEC Laboratories Europe, NEC Europe Ltd.
39 Oak Ridge National Laboratory
40 The Ohio State University
41 Pacific Northwest National Laboratory
42 QLogic Corporation
43 Sandia National Laboratories
44 SiCortex
45 Silicon Graphics Incorporated
46 Sun Microsystems, Inc.
47 University of Alabama at Birmingham
48 University of Houston

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

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.

1	Pavan Balaji	Purushotham V. Bangalore	Brian Barrett
2	Richard Barrett	Christian Bell	Robert Blackmore
3	Gil Bloch	Ron Brightwell	Greg Bronevetsky
4	Jeff Brown	Darius Buntinas	Jonathan Carter
5	Nathan DeBardeleben	Terry Dontje	Gabor Dozsa
6	Edric Ellis	Karl Feind	Edgar Gabriel
7	Patrick Geoffray	Johann George	David Gingold
8	David Goodell	Erez Haba	Robert Harrison
9	Thomas Herault	Marc-André Hermanns	Steve Hodson
10	Joshua Hursey	Yutaka Ishikawa	Bin Jia
11	Hideyuki Jitsumoto	Terry Jones	Yann Kalemkarian
12	Ranier Keller	Matthew Koop	Quincey Koziol
13	Manojkumar Krishnan	Sameer Kumar	Miron Livny
14	Andrew Lumsdaine	Miao Luo	Ewing Lusk
15	Timothy I. Mattox	Kannan Narasimhan	Mark Pagel
16	Avneesh Pant	Steve Poole	Howard Pritchard
17	Craig Rasmussen	Hubert Ritzdorf	Rob Ross
18	Martin Schulz	Pavel Shamis	Galen Shipman
19	Christian Siebert	Anthony Skjellum	Brian Smith
20	Naoki Sueyasu	Vinod Tipparaju	Keith Underwood
21	Rolf Vandevaart	Abhinav Vishnu	Weikuan Yu

22
23 The MPI Forum also acknowledges and appreciates the valuable input from people via
24 e-mail and in person.

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

27 Argonne National Laboratory
28 Auburn University
29 Bull
30 Cisco Systems, Inc.
31 Cray Inc.
32 Forschungszentrum Jülich
33 Fujitsu
34 The HDF Group
35 Hewlett-Packard
36 International Business Machines
37 Indiana University
38 Institut National de Recherche en Informatique et Automatique (Inria)
39 Institute for Advanced Science & Engineering Corporation
40 Intel Corporation
41 Lawrence Berkeley National Laboratory
42 Lawrence Livermore National Laboratory
43 Los Alamos National Laboratory
44 Mathworks
45 Mellanox Technologies
46 Microsoft
47 Myricom
48 NEC Corporation

Oak Ridge National Laboratory	1
The Ohio State University	2
Pacific Northwest National Laboratory	3
QLogic Corporation	4
RunTime Computing Solutions, LLC	5
Sandia National Laboratories	6
SiCortex, Inc.	7
Silicon Graphics Inc.	8
Sun Microsystems, Inc.	9
Tokyo Institute of Technology	10
University of Alabama at Birmingham	11
University of Houston	12
University of Illinois at Urbana-Champaign	13
University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)	14
University of Tennessee, Knoxville	15
University of Tokyo	16
University of Wisconsin	17

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 significant 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

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

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

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

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

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

43
44 Argonne National Laboratory
45 Bull
46 Cisco Systems, Inc.
47 Cray Inc.
48 CSCS

ETH Zurich	1
Fujitsu Ltd.	2
German Research School for Simulation Sciences	3
The HDF Group	4
Hewlett-Packard	5
International Business Machines	6
IBM India Private Ltd	7
Indiana University	8
Institut National de Recherche en Informatique et Automatique (Inria)	9
Institute for Advanced Science & Engineering Corporation	10
Intel Corporation	11
Lawrence Berkeley National Laboratory	12
Lawrence Livermore National Laboratory	13
Los Alamos National Laboratory	14
Mellanox Technologies, Inc.	15
Microsoft Corporation	16
NEC Corporation	17
National Oceanic and Atmospheric Administration, Global Systems Division	18
NVIDIA Corporation	19
Oak Ridge National Laboratory	20
The Ohio State University	21
Oracle America	22
Platform Computing	23
RIKEN AICS	24
RunTime Computing Solutions, LLC	25
Sandia National Laboratories	26
Technical University of Chemnitz	27
Tokyo Institute of Technology	28
University of Alabama at Birmingham	29
University of Chicago	30
University of Houston	31
University of Illinois at Urbana-Champaign	32
University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)	33
University of Tennessee, Knoxville	34
University of Tokyo	35

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

- 1 • Rolf Rabenseifner, Steering Committee, Terms and Definitions, and Fortran Bindings,
2 Deprecated Functions, Annex Change-Log, and Annex Language Bindings
- 3
- 4 • Richard L. Graham, Steering Committee, Meeting Convener
- 5
- 6 • Jeffrey M. Squyres, Language Bindings and MPI-3.1 Secretary
- 7
- 8 • Daniel Holmes, Point-to-Point Communication
- 9
- 10 • George Bosilca, Datatypes and Environmental Management
- 11
- 12 • Torsten Hoefler, Collective Communication and Process Topologies
- 13
- 14 • Pavan Balaji, Groups, Contexts, and Communicators, and External Interfaces
- 15
- 16 • Jeff Hammond, The Info Object
- 17
- 18 • David Solt, Process Creation and Management
- 19
- 20 • Quincey Koziol, I/O
- 21
- 22 • Kathryn Mohror, Tool Support
- 23
- 24 • Rajeev Thakur, One-Sided Communications

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

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

Hari Subramoni	Shinji Sumimoto	Alexander Supalov	1
Bronis R. de Supinski	Sayantana Sur	Masamichi Takagi	2
Keita Teranishi	Rajeev Thakur	Fabian Tillier	3
Yuichi Tsujita	Geoffroy Vallée	Rolf vandeVaart	4
Akshay Venkatesh	Jerome Vienne	Venkat Vishwanath	5
Anh Vo	Huseyin S. Yildiz	Junchao Zhang	6
Xin Zhao			7

The MPI Forum also acknowledges and appreciates the valuable input from people via e-mail and in person. 8

The following institutions supported the MPI-3.1 effort through time and travel support for the people listed above. 9

- Argonne National Laboratory 10
- Auburn University 11
- Cisco Systems, Inc. 12
- Cray 13
- EPCC, The University of Edinburgh 14
- ETH Zurich 15
- Forschungszentrum Jülich 16
- Fujitsu 17
- German Research School for Simulation Sciences 18
- The HDF Group 19
- International Business Machines 20
- Institut National de Recherche en Informatique et Automatique (Inria) 21
- Intel Corporation 22
- Kyushu University 23
- Lawrence Berkeley National Laboratory 24
- Lawrence Livermore National Laboratory 25
- Lenovo 26
- Los Alamos National Laboratory 27
- Mellanox Technologies, Inc. 28
- Microsoft Corporation 29
- NEC Corporation 30
- NVIDIA Corporation 31
- Oak Ridge National Laboratory 32
- The Ohio State University 33
- RIKEN AICS 34
- Sandia National Laboratories 35
- Texas Advanced Computing Center 36
- Tokyo Institute of Technology 37
- University of Alabama at Birmingham 38
- University of Houston 39
- University of Illinois at Urbana-Champaign 40
- University of Oregon 41
- University of Stuttgart, High Performance Computing Center Stuttgart (HLRS) 42
- University of Tennessee, Knoxville 43
- University of Tokyo 44

1 MPI-4.0:

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

3 The editors and organizers of the MPI-4.0 have been:

- 4
- 5 • Martin Schulz, MPI-4.0 Chair, Info Object, External Interfaces
- 6
- 7 • Richard Graham, MPI-4.0 Treasurer
- 8
- 9 • Wesley Bland, MPI-4.0 Secretary, Backward Incompatibilities
- 10
- 11 • William Gropp, MPI-4.0 Editor, Steering Committee, Front Matter, Introduction,
12 One-Sided Communications, and Bibliography
- 13
- 14 • Rolf Rabenseifner, Steering Committee, Process Topologies, Deprecated Functions,
15 Removed Interfaces, Annex Language Bindings Summary, and Annex Change-Log.
- 16
- 17 • Purushotham V. Bangalore, Language Bindings
- 18
- 19 • Claudia Blaas-Schenner, Terms and Conventions
- 20
- 21 • George Bosilca, Datatypes and Environmental Management
- 22
- 23 • Ryan E. Grant, Partitioned Communication
- 24
- 25 • Marc-André Hermanns, Tool Support
- 26
- 27 • Daniel Holmes, Point-to-Point Communication, Sessions
- 28
- 29 • Guillaume Mercier, Groups, Contexts, Communicators, Caching
- 30
- 31 • Howard Pritchard, Process Creation and Management
- 32
- 33 • Anthony Skjellum, Collective Communication, I/O

30 As part of the development of MPI-4.0, a number of working groups were established. In
31 some cases, the work for these groups overlapped with multiple chapters. The following
32 describes the major working groups and the leaders of those groups:

34 **Collective Communication, Topology, Communicators:** Torsten Hoefer, Andrew
35 Lumsdaine, and Anthony Skjellum

36 **Fault Tolerance:** Wesley Bland, Aurélien Bouteiller, and Richard Graham

37 **Hardware-Topologies:** Guillaume Mercier

38 **Hybrid & Accelerator:** Pavan Balaji and James Dinan

39 **Large Counts:** Jeff Hammond

40 **Persistence:** Anthony Skjellum

41 **Point to Point Communication:** Daniel Holmes and Richard Graham

42 **Remote Memory Access:** William Gropp and Rajeev Thakur

43 **Semantic Terms:** Purushotham V. Bangalore and Rolf Rabenseifner

Sessions: Daniel Holmes and Howard Pritchard

1

Tools: Kathryn Mohror and Marc-André Hermanns

2

3

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

4

5

6

Julien Adam	Abdelhalim Amer	Charles Archer	7
Ammar Ahmad Awan	Pavan Balaji	Purushotham V. Bangalore	8
Mohammadreza Bayatpour	Jean-Baptiste Besnard	Claudia Blaas-Schenner	9
Wesley Bland	Gil Bloch	George Bosilca	10
Aurelien Bouteiller	Ben Bratu	Alexander Calvert	11
Nicholas Chaimov	Sourav Chakraborty	Steffen Christgau	12
Ching-Hsiang Chu	Mikhail Chuvelev	James Clark	13
Carsten Clauss	Isaias Alberto Compres Urena		14
Giuseppe Congiu	Brandon Cook	James Custer	15
Anna Daly	Hoang-Vu Dang	James Dinan	16
Matthew Dosanjh	Murali Emani	Christian Engelmann	17
Noah Evans	Ana Gainaru	Esthela Gallardo	18
Marc Gamell Balmana	Balazs Gerofi	Salvatore Di Girolamo	19
Brice Goglin	Manjunath Gorentla Venkata	Richard Graham	20
Ryan E. Grant	Stanley Graves	William Gropp	21
Siegmar Gross	Taylor Groves	Yanfei Guo	22
Khaled Hamidouche	Jeff Hammond	Marc-André Hermanns	23
Nathan Hjelm	Torsten Hoefler	Daniel Holmes	24
Atsushi Hori	Josh Hursey	Ilya Ivanov	25
Julien Jaeger	Emmanuel Jeannot	Sylvain Jaeger	26
Jithin Jose	Krishna Kandalla	Takahiro Kawashima	27
Chulho Kim	Michael Knobloch	Alice Koniges	28
Sameer Kumar	Kim Kyunghun	Ignacio Laguna Peralta	29
Stefan Lankes	Tonglin Li	Xioyi Lu	30
Kavitha Madhu	Alexey Malhanov	Ryan Marshall	31
William Marts	Guillaume Mercier	Ali Mohammed	32
Kathryn Mohror	Takeshi Nanri	Thomas Naughton	33
Christoph Niethammer	Takafumi Nose	Lena Oden	34
Steve Oyanagi	Guillaume Papauré	Ivy Peng	35
Antonio Peña	Simon Pickartz	Artem Polyakov	36
Sreeram Potluri	Howard Pritchard	Martina Prugger	37
Marc Pérache	Rolf Rabenseifner	Nicholas Radcliffe	38
Ken Raffenetti	Craig Rasmussen	Soren Rasmussen	39
Hubert Ritzdorf	Sergio Rivas-Gomez	Davide Rossetti	40
Martin Ruefenacht	Amit Ruhela	Whit Schonbein	41
Joseph Schuchart	Martin Schulz	Sangmin Seo	42
Sameh Sharkawi	Sameer Shende	Min Si	43
Anthony Skjellum	Brian Smith	David Solt	44
Jeffrey M. Squyres	Srinivas Sridharan	Hari Subramoni	45
Nawrin Sultana	Shinji Sumimoto	Sayantana Sur	46

47

48

1 Hugo Taboada Keita Teranishi Rajeev Thakur
2 Keith Underwood Geoffroy Vallee Akshay Venkatesh
3 Jerome Vienne Anh Vo Justin Wozniak
4 Junchao Zhang Dong Zhong Hui Zhou
5
6
7

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

10 The following institutions supported the MPI-4.0 effort through time and travel support
11 for the people listed above.

12 ATOS
13 Argonne National Laboratory
14 Arm
15 Auburn University
16 Barcelona Supercomputing Center
17 CEA
18 Cisco Systems Inc.
19 Cray Inc.
20 EPCC, The University of Edinburgh
21 ETH Zürich
22 Fujitsu
23 Fulda University of Applied Sciences
24 German Research School for Simulation Sciences
25 Hewlett Packard Enterprise
26 International Business Machines
27 Institut National de Recherche en Informatique et Automatique (Inria)
28 Intel Corporation
29 Jülich Supercomputing Center, Forschungszentrum Jülich
30 KTH Royal Institute of Technology
31 Kyushu University
32 Lawrence Berkeley National Laboratory
33 Lawrence Livermore National Laboratory
34 Lenovo
35 Los Alamos National Laboratory
36 Mellanox Technologies, Inc.
37 Microsoft Corporation
38 NEC Corporation
39 NVIDIA Corporation
40 Oak Ridge National Laboratory
41 PAR-TEC
42 Paratools, Inc.
43 RIKEN AICS (R-CCS as of 2017)
44 RWTH Aachen University
45 Rutgers University
46 Sandia National Laboratories
47 Silicon Graphics, Inc.
48 Technical University of Munich

The HDF Group	1
The Ohio State University	2
Texas Advanced Computing Center	3
Tokyo Institute of Technology	4
University of Alabama at Birmingham	5
University of Basel, Switzerland	6
University of Houston	7
University of Illinois at Urbana-Champaign and the National Center for Supercomputing Applications	8
University of Innsbruck	10
University of Oregon	11
University of Potsdam	12
University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)	13
University of Tennessee, Chattanooga	14
University of Tennessee, Knoxville	15
University of Texas at El Paso	16
University of Tokyo	17
VSC Research Center, TU Wien	18

MPI-4.1:

MPI-4.1 is a minor update to the MPI Standard.

The editors and organizers of the MPI-4.1 have been: **REVIEW AND CORRECT**

- Martin Schulz, MPI-4.1 Chair, Info Object, External Interfaces
- Richard Graham, MPI-4.1 Treasurer
- Wesley Bland, MPI-4.1 Secretary, Backward Incompatibilities
- William Gropp, MPI-4.1 Editor, Steering Committee, Front Matter, Introduction, One-Sided Communications, and Bibliography
- Rolf Rabenseifner, Steering Committee, Process Topologies, Deprecated Functions, Removed Interfaces, Annex Language Bindings Summary, and Annex Change-Log.
- Purushotham V. Bangalore, Language Bindings
- Claudia Blaas-Schenner, Terms and Conventions
- George Bosilca, Datatypes and Environmental Management
- Ryan E. Grant, Partitioned Communication
- Marc-André Hermans, Tool Support
- Daniel Holmes, Point-to-Point Communication, Sessions
- Guillaume Mercier, Groups, Contexts, Communicators, Caching
- Howard Pritchard, Process Creation and Management
- Anthony Skjellum, Collective Communication, I/O

1 As part of the development of MPI-4.1, a number of working groups were established or
2 continued from MPI-4.0. In some cases, the work for these groups overlapped with
3 multiple chapters. The following describes the major working groups and the leaders of
4 those groups: **REVIEW AND CORRECT**

5
6
7 **Collective Communication, Topology, Communicators:** Torsten Hoeffler, Andrew
8 Lumsdaine, and Anthony Skjellum

9 **Fault Tolerance:** Wesley Bland, Aurélien Bouteiller, and Richard Graham

10
11 **Hardware-Topologies:** Guillaume Mercier

12
13 **Hybrid & Accelerator:** Pavan Balaji and James Dinan

14
15 **Large Counts:** Jeff Hammond

16
17 **Persistence:** Anthony Skjellum

18
19 **Point to Point Communication:** Daniel Holmes and Richard Graham

20
21 **Remote Memory Access:** William Gropp and Rajeev Thakur

22
23 **Semantic Terms:** Purushotham V. Bangalore and Rolf Rabenseifner

24
25 **Sessions:** Daniel Holmes and Howard Pritchard

26
27 **Tools:** Kathryn Mohror and Marc-André Hermanns

28
29 The following list includes some of the active participants who attended MPI Forum
30 meetings or participated in the e-mail discussions.

31
32 will be generated from list

33
34 The MPI Forum also acknowledges and appreciates the valuable input from people via
35 e-mail and in person.

36
37 The following institutions supported the MPI-4.0 effort through time and travel support
38 for the people listed above.

39
40 Will generate from list

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 that, 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.

- 1 • Define an interface that can be implemented on many vendor’s platforms, with no
2 significant changes in the underlying communication and system software.
- 3
- 4 • Semantics of the interface should be language independent.
- 5
- 6 • The interface should be designed to allow for thread safety.
- 7

8 1.2 Background of MPI-1.0

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

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

24 A preliminary draft proposal, known as MPI-1, was put forward by Dongarra, Hempel,
25 Hey, and Walker in November 1992, and a revised version was completed in February
26 1993 [19]. MPI-1 embodied the main features that were identified at the Williamsburg
27 workshop as being necessary in a message passing standard. Since MPI-1 was primarily
28 intended to promote discussion and “get the ball rolling,” it focused mainly on point-to-point
29 communications. MPI-1 brought to the forefront a number of important standardization
30 issues, but did not include any collective communication routines and was not thread-safe.

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

41 42 1.3 Background of MPI-1.1, MPI-1.2, and MPI-2.0

43
44 Beginning in March 1995, the MPI Forum began meeting to consider corrections and exten-
45 sions to the original MPI Standard document [23]. The first product of these deliberations
46 was Version 1.1 of the MPI specification, released in June of 1995 [24] (see
47 <http://www.mpi-forum.org> for official MPI document releases). At that time, effort focused
48 in five areas.

1. Further corrections and clarifications for the MPI-1.1 document. 1
2. Additions to MPI-1.1 that do not significantly change its types of functionality (new datatype constructors, language interoperability, etc.). 2
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.” 3
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. 4
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). 5

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. 6

This structure makes it easy for users and implementors to understand what level of MPI compliance a given implementation has: 7

- 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. 8
- MPI-2 compliance will mean compliance with all of MPI-2.1. 9
- The MPI Journal of Development is not part of the MPI Standard. 10

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. 11

1.4 Background of MPI-1.3 and MPI-2.1 12

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. 13

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 14

1 decided to combine the existing and future MPI documents to one document for each ver-
2 sion of the MPI standard. For technical and historical reasons, this series was started with
3 MPI-1.3. Additional Ballots 3 and 4 solved old questions from the errata list started in 1995
4 up to new questions from the last years. After all documents (MPI-1.1, MPI-2, Errata for
5 MPI-1.1 (Oct. 12, 1998), and MPI-2.1 Ballots 1–4) were combined into one draft document,
6 for each chapter, a chapter author and review team were defined. They cleaned up the
7 document to achieve a consistent MPI-2.1 document. The final MPI-2.1 standard document
8 was finished in June 2008, and finally released with a second vote in September 2008 in the
9 meeting at Dublin, just before EuroPVM/MPI’08.

11 1.5 Background of MPI-2.2

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

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

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

26 1.6 Background of MPI-3.0

28 MPI-3.0 is a major update to the MPI standard. The updates include the extension of
29 collective operations to include nonblocking versions, extensions to the one-sided operations,
30 and a new Fortran 2008 binding. In addition, the deprecated C++ bindings have been
31 removed, as well as many of the deprecated routines and MPI objects (such as the MPI_UB
32 datatype). Any valid MPI-2.2 program not using any of these removed MPI procedures or
33 objects is a valid MPI-3.0 program.

36 1.7 Background of MPI-3.1

38 MPI-3.1 is a minor update to the MPI standard. Most of the updates are corrections
39 and clarifications to the standard, especially for the Fortran bindings. New functions added
40 include routines to manipulate MPI_Aint values in a portable manner, nonblocking collective
41 I/O routines, and routines to get the index value by name for MPI_T performance and
42 control variables. A general index was also added. Any valid MPI-3.0 program is a valid
43 MPI-3.1 program.

45 1.8 Background of MPI-4.0

47 MPI-4.0 is a major update to the MPI standard. The largest changes are the addition of
48 large-count versions of many routines to address the limitations of using an int or INTEGER

for the count parameter, persistent collectives, partitioned communications, an alternative way to initialize MPI, application info assertions, and improvements to the definitions of error handling. In addition, there are a number of smaller improvements and corrections. Any valid MPI-3.1 is a valid MPI-4.0 program with the exception of semantic changes listed in Chapter 18.

1.9 Background of 2022 Draft Specification

The 2022 draft specification is expected to become the MPI-4.1 specification once all features have been merged.

1.10 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.

1.11 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.

1.12 What Is Included in the Standard?

The standard includes:

- Point-to-point communication,
- Partitioned communication,

- 1 • Datatypes,
- 2
- 3 • Collective operations,
- 4
- 5 • Process groups,
- 6
- 7 • Communication contexts,
- 8
- 9 • Process topologies,
- 10
- 11 • Environmental management and inquiry,
- 12
- 13 • The Info object,
- 14
- 15 • Process initialization, creation, and management,
- 16
- 17 • One-sided communication,
- 18
- 19 • External interfaces,
- 20
- 21 • Parallel file I/O,
- 22
- 23 • Tool support, and
- 24
- 25 • Language bindings for Fortran and C.

26 1.13 Organization of This Document

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

- 29 • Chapter 2, [MPI Terms and Conventions](#), explains notational terms and conventions
30 used throughout the MPI document.
- 31 • Chapter 3, [Point-to-Point Communication](#), defines the basic, pairwise communication
32 subset of MPI. *Send* and *receive* are found here, along with many associated functions
33 designed to make basic communication powerful and efficient.
- 34 • Chapter 4, [Partitioned Point-to-Point Communication](#), defines a method of perform-
35 ing partitioned communication in MPI. Partitioned communication allows multiple
36 contributions of data to be made, potentially, from multiple actors (e.g., threads or
37 tasks) in an MPI process to a single message.
- 38 • Chapter 5, [Datatypes](#), defines a method to describe any data layout, e.g., an array of
39 structures in the memory, which can be used as message send or receive buffer.
- 40 • Chapter 6, [Collective Communication](#), defines process-group collective communica-
41 tion operations. Well known examples of this are barrier and broadcast over a group
42 of processes (not necessarily all the processes). With MPI-2, the semantics of collec-
43 tive communication was extended to include inter-communicators. It also adds two
44 new collective operations. MPI-3 adds nonblocking collective operations. MPI-4 adds
45 persistent nonblocking collective operations.
- 46
- 47
- 48

- Chapter 7, [Groups, Contexts, Communicators, and Caching](#), shows how groups of processes are formed and manipulated, how unique communication contexts are obtained, and how the two are bound together into a *communicator*. 1
2
3
4
- Chapter 8, [Process Topologies](#), explains a set of utility functions meant to assist in the mapping of process groups (a linearly ordered set) to richer topological structures such as multi-dimensional grids. 5
6
7
- Chapter 9, [MPI Environmental Management](#), explains how the programmer can manage and make inquiries of the current MPI environment. These functions are needed for the writing of correct, robust programs, and are especially important for the construction of highly-portable message-passing programs. 8
9
10
11
12
- Chapter 10, [The Info Object](#), defines an opaque object, that is used as input in several MPI routines. 13
14
15
- Chapter 11, [Process Initialization, Creation, and Management](#), defines several approaches to MPI initialization, process creation, and process management while placing minimal restrictions on the execution environment. MPI-4 adds a new Sessions Model. 16
17
18
19
20
- Chapter 12, [One-Sided Communications](#), defines communication routines that can be completed by a single process. These include shared-memory operations (put/get) and remote accumulate operations. 21
22
23
24
- Chapter 13, [External Interfaces](#), defines routines designed to allow developers to layer on top of MPI. This includes generalized requests, routines that decode MPI opaque objects, and threads. 25
26
27
- Chapter 14, [I/O](#), defines MPI support for parallel I/O. 28
29
- Chapter 15, [Tool Support](#), covers interfaces that allow debuggers, performance analyzers, and other tools to obtain data about the operation of MPI processes. This chapter includes Section 15.2 ([Profiling Interface](#)), which was a chapter in previous versions of MPI. 30
31
32
33
- Chapter 16, [Deprecated Interfaces](#), describes routines that are kept for reference. However usage of these functions is discouraged, as they may be deleted in future versions of the standard. 34
35
36
37
- Chapter 17, [Removed Interfaces](#), describes routines and constructs that have been removed from MPI. These were deprecated in MPI-2, and the MPI Forum decided to remove these from the MPI-3 standard. 38
39
40
41
- Chapter 18, [Semantic Changes and Warnings](#), describes semantic changes from previous versions of MPI. 42
43
- Chapter 19, [Language Bindings](#), discusses Fortran issues, and describes language interoperability aspects between C and Fortran. 44
45
46

The Appendices are: 47
48

- 1 • Annex A, [Language Bindings Summary](#), gives specific syntax in C and Fortran, for
2 all MPI functions, constants, and types.
- 3
- 4 • Annex B, [Change-Log](#), summarizes some changes since the previous version of the
5 standard.
- 6
- 7 • Several Index pages show the locations of [general terms and definitions](#), [examples, con-](#)
8 [stants and predefined handles](#), [declarations of C and Fortran types](#), [callback routine](#)
9 [prototypes](#), and all [MPI functions](#).

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

14 A separate document consists of ideas that were discussed in the MPI Forum during the
15 MPI-2 development and deemed to have value, but were not included in the MPI Standard.
16 They are part of the “Journal of Development” (JOD), which was created to capture these
17 ideas and discussions. The JOD is available at <https://www.mpi-forum.org/docs>.

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 and the Fortran `mpi_f08` module, all routines associated with a particular type of MPI object should be of the form `MPI_Class_action_subset` or, if no subset exists, of the form `MPI_Class_action`. In the Fortran `mpi` module and `mpif.h` file, all routines associated with a particular type of MPI object should be of the form `MPI_CLASS_ACTION_SUBSET` or, if no subset exists, of the form `MPI_CLASS_ACTION`.
2. If the routine is not associated with a class, the name should be of the form `MPI_Action_subset` or `MPI_ACTION_SUBSET` in C and Fortran.
3. The names of certain actions have been standardized. In particular, **Create** creates a new object, **Get** retrieves information about an object, **Set** sets this information, **Delete** deletes information, **Is** asks whether or not an object has a certain property.

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

2.3 Procedure Specification

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

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

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

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

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

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

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

2.4 Semantic Terms

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

2.4.1 MPI Operations

MPI operation: An MPI operation is a sequence of steps performed by the MPI library to establish and enable data transfer and/or synchronization. It consists of four stages: initialization, starting, completion, and freeing, and it is implemented as a set of one or more MPI procedures, see Section 2.4.2.

Initialization hands over the argument list to the operation but not the content of the data buffers, if any. The specification of an operation may state that array arguments must not be changed until the operation is freed.

Starting hands over control of the data buffers, if any, to the associated operation.

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

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

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

Freeing returns control of the rest of the argument list (e.g., the data buffer address and array arguments).

MPI operations are available in one or more of these forms: blocking, nonblocking, and persistent.



Figure 2.1: State transition diagram for blocking operations



Figure 2.2: State transition diagram for nonblocking operations

Blocking operation: For a **blocking operation**, all four stages are combined in a single procedure call (as shown in Figure 2.1 and defined in Section 2.4.2).

Nonblocking operation: For a **nonblocking operation**, the initialization and starting stages are combined into a single nonblocking procedure call and the completion and freeing stages are combined into a separate, single procedure call, which can be blocking or nonblocking (as shown in Figure 2.2 and defined in Section 2.4.2).

Persistent operation: For a **persistent operation**, there is a separate procedure for each of the four stages (as shown in Figure 2.3 and defined in Section 2.4.2). Each of these procedures may be blocking or nonblocking.

For a partitioned send operation, an additional call to activate each partition of the send buffer (see Section 4.2.1) is required to finish the starting stage. For a partitioned receive operation, before the operation is complete the user is allowed to access a partition of the output buffer after verifying that it has arrived (see Section 4.2.2).

Additionally, an MPI operation can be collective or noncollective.

Collective operation: Collective operations are defined as operations that involve a group or groups of MPI processes. For collective operations the completion stage may or may not finish before all processes in the group have started the operation.

Collective MPI operations are also available as blocking, nonblocking, or persistent operations.

Noncollective operation: Noncollective operations are defined as operations that are not collective.

2.4.2 MPI Procedures

All MPI procedures can either be *local* or *nonlocal*—defined as follows:

Nonlocal procedure: An MPI procedure is **nonlocal** if returning may require, during its execution, some specific semantically-related MPI procedure to be called on another MPI process.

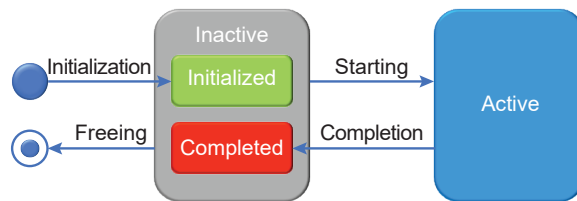


Figure 2.3: State transition diagram for persistent operations

Local procedure: An MPI procedure is **local** if it is not *nonlocal*.

An MPI operation is implemented as a set of one or more MPI procedures. An MPI **operation-related procedure** implements at least a part of a stage of an MPI operation as described in Section 2.4.1. An MPI operation-related procedure may also implement one or more stages of one or several MPI operations. In certain cases, more than one MPI operation-related procedure may be needed to implement a single stage.

There are also other MPI procedures that do not implement any stage of any MPI operation.

The semantics of MPI operation-related procedures are described using two orthogonal (independent) concepts: completeness (depends on which stages are included) and locality. Such procedures can be either incomplete, or completing, or freeing, or completing and freeing based on the status of the associated operation at the time the procedure returns. Also, all such procedures can be described as either blocking or nonblocking, but these latter two terms refer to combinations of the completeness and locality concepts. Additionally, all MPI operation-related procedures can be collective or noncollective.

The following are properties of MPI operation-related procedures:

Initialization procedure: An MPI procedure is an **initialization procedure** if return from the procedure indicates that the associated operation has completed its initialization stage, which implies that the user has handed over control of the argument list (but not contents of the data buffers) to MPI. The user is still allowed to read or modify the contents of the data buffers. If an initializing procedure is not also the freeing procedure of the associated operation (see below) then the user is not permitted to deallocate the data buffers or to modify the array arguments.

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

Initiation procedure: An MPI procedure is an **initiation procedure** if return from the procedure indicates that both the initialization and the starting stage have completed, which implies control of the entire argument list is handed over to MPI.

Completing procedure: An MPI procedure is called **completing** if return from the procedure indicates that at least one associated operation has finished its completion stage, which implies that the user can rely on the content of the output data buffers

and modify the content of input and output data buffers of such operation(s). If a completing procedure is not also a freeing procedure (see below) then the user is not permitted to deallocate the data buffers or to modify the array arguments.

Incomplete procedure: An MPI procedure is called **incomplete** if it is not a completing procedure.

Freeing procedure: An MPI procedure is **freeing** if return from the procedure indicates that at least one associated operation has finished its freeing stage, which implies that the user can reuse all parameters specified when initializing such associated operation(s).

Nonblocking procedure: An MPI procedure is **nonblocking** if it is incomplete and local.

Blocking procedure: An MPI procedure is **blocking** if it is not nonblocking.

Advice to users. Note that for operation-related MPI procedures, in most cases incomplete procedures are local and completing procedures are nonlocal. Exceptions are noted where such procedures are defined. In many cases an additional prefix letter **I** as an abbreviation of the words **incomplete** and **immediate** marks nonblocking procedures in the procedure name.

Some categorization examples are listed below.

Nonblocking procedures:

- incomplete and local: MPI_ISEND, MPI_IRecv, MPI_IBCAST, MPI_IMPROBE, MPI_SEND_INIT, MPI_RECV_INIT, ...

Blocking procedures:

- completing and nonlocal: MPI_SEND, MPI_RECV, MPI_BCAST, ...
- incomplete and nonlocal: MPI_MPROBE, MPI_BCAST_INIT, ..., MPI_FILE_{READ|WRITE}_{AT_ALL|ALL|ORDERED}_BEGIN.
- completing and local: MPI_BSEND, MPI_RSEND, MPI_MRECV.

MPI procedures that are not MPI operation-related:

- MPI_COMM_RANK, MPI_WTIME, MPI_PROBE, MPI_Iprobe, ...

(End of advice to users.)

Collective procedure: An MPI procedure is **collective** if all processes in a group or groups of MPI processes need to invoke the procedure.

Initialization procedures of collective operations over the same process group must be executed in the same order by all members of the process group.

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

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

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

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

2.4.3 MPI Datatypes

For datatypes, the following terms are defined:

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

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

portable: A datatype is portable if it is a predefined datatype, or it is derived from 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 Datatypes

2.5.1 Opaque Objects

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

arguments to access these objects. In addition to their use by MPI calls for object access, handles can participate in assignments and comparisons.

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

```

10 TYPE, BIND(C) :: MPI_Comm
11     INTEGER    :: MPI_VAL
12 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 that 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 than copies of its components; a call to `MPI_COMM_GROUP` may return a reference to the group associated with the communicator, rather than a copy of this group. In such cases, the implementation must maintain reference counts, and allocate and deallocate objects in such a way that the visible effect is as if the objects were copied. (*End of advice to implementors.*)

2.5.2 Array Arguments

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

1 considered valid entries. When a NULL argument is desired for an array of statuses, one
 2 uses MPI_STATUSES_IGNORE.
 3

4 2.5.3 State

5 MPI procedures use at various places arguments with *state* types. The values of such a
 6 datatype are all identified by names, and no operation is defined on them. For example,
 7 the MPI_TYPE_CREATE_SUBARRAY routine has a state argument *order* with values
 8 MPI_ORDER_C and MPI_ORDER_FORTRAN.
 9

10 2.5.4 Named Constants

11 MPI procedures sometimes assign a special meaning to a special value of a basic type argu-
 12 ment; e.g., *tag* is an integer-valued argument of point-to-point communication operations,
 13 with a special wild-card value, MPI_ANY_TAG. Such arguments will have a range of regular
 14 values, which is a proper subrange of the range of values of the corresponding basic type;
 15 special values (such as MPI_ANY_TAG) will be outside the regular range. The range of regu-
 16 lar values, such as *tag*, can be queried using environmental inquiry functions, see Chapter 9.
 17 The range of other values, such as *source*, depends on values given by other MPI routines
 18 (in the case of *source* it is the communicator size).
 19

20 MPI also provides predefined named constant handles, such as MPI_COMM_WORLD.
 21

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

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

34 MPI_MAX_PROCESSOR_NAME
 35 MPI_MAX_LIBRARY_VERSION_STRING
 36 MPI_MAX_ERROR_STRING
 37 MPI_MAX_DATAREP_STRING
 38 MPI_MAX_INFO_KEY
 39 MPI_MAX_INFO_VAL
 40 MPI_MAX_OBJECT_NAME
 41 MPI_MAX_PORT_NAME
 42 MPI_VERSION
 43 MPI_SUBVERSION
 44 MPI_F_STATUS_SIZE (C only)
 45 MPI_STATUS_SIZE (Fortran only)
 46 MPI_ADDRESS_KIND (Fortran only)
 47 MPI_COUNT_KIND (Fortran only)
 48 MPI_INTEGER_KIND (Fortran only)

MPI_OFFSET_KIND	(Fortran only)	1
MPI_SUBARRAYS_SUPPORTED	(Fortran only)	2
MPI_ASYNC_PROTECTS_NONBLOCKING	(Fortran only)	3

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

MPI_BOTTOM	7
MPI_STATUS_IGNORE	8
MPI_STATUSES_IGNORE	9
MPI_ERRCODES_IGNORE	10
MPI_IN_PLACE	11
MPI_ARGV_NULL	12
MPI_ARGVS_NULL	13
MPI_UNWEIGHTED	14
MPI_WEIGHTS_EMPTY	15

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 with TS 29113 syntax `TYPE(*), DIMENSION(..)`; for C, we use `void*`.

Advice to implementors. Implementors can freely choose how to implement choice arguments in the `mpi` module, e.g., with a nonstandard compiler-dependent method that has the quality of the call mechanism in the implicit Fortran interfaces, or with the method defined for the `mpi_f08` module. See details in Section 19.1.1. (*End of advice to implementors.*)

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

1 be passed directly to another language without conversion. There is the MPI constant
2 MPI_BOTTOM to indicate the start of the address range. For retrieving absolute addresses
3 or any calculation with absolute addresses, one should use the routines and functions pro-
4 vided in Section 5.1.5. Section 5.1.12 provides additional rules for the correct use of absolute
5 addresses. For expressions with relative displacements or other usage without absolute ad-
6 dresses, intrinsic operators (e.g., +, -, *) can be used.

7
8 *Rationale.* Byte displacement values need to be large enough to encode any value
9 used for expressing absolute or relative memory addresses. Prior to MPI-4.0, some
10 MPI routines used `int` in C and `INTEGER` in Fortran as the type for *byte displacement*
11 arguments. To avoid breaking backward compatibility, this version of the standard
12 continues to support `int` in C as well as `INTEGER` in Fortran in such routines. In
13 addition, this version of the standard supports using `MPI_Aint` in C (via separate
14 “_c” suffixed procedures) as well as `INTEGER(KIND=MPI_ADDRESS_KIND)` in Fortran (via
15 polymorphic interfaces in newer MPI Fortran bindings (USE `mpi_f08`)) in such routines.
16 See Section 19.2 for a full explanation. (*End of rationale.*)

17 18 2.5.7 File Offsets

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

25 26 27 2.5.8 Counts

28 As described above, MPI defines types (e.g., `MPI_Aint`) to address locations within memory
29 and other types (e.g., `MPI_Offset`) to address locations within files. In addition, some MPI
30 procedures use *count* arguments that represent a number of MPI datatypes on which to
31 operate. Furthermore, *timestamps* in the context of the MPI Tool Information Interface
32 are a count of clock ticks elapsed since some time in the past. At times, one needs a
33 single type that can be used to address locations within either memory or files as well as
34 express *count* values, and that type is `MPI_Count` in C and `INTEGER(KIND=MPI_COUNT_KIND)`
35 in Fortran. These types must have the same width and encode values in the same manner
36 such that count values in one language may be passed directly to another language without
37 conversion. The size of the `MPI_Count` type is determined by the MPI implementation
38 with the restriction that it must be minimally capable of encoding any value that may
39 be stored in a variable of type `int`, `MPI_Aint`, or `MPI_Offset` in C and of type `INTEGER`,
40 `INTEGER(KIND=MPI_ADDRESS_KIND)`, or `INTEGER(KIND=MPI_OFFSET_KIND)` in Fortran. Even
41 though the `MPI_Count` type is large enough to encode address locations, the `MPI_Count` type
42 shall not be used to represent an *absolute address*.

43
44 *Rationale.* Count values need to be large enough to encode any value used for
45 expressing element counts, strides, offsets, indexes, displacements, typemaps in mem-
46 ory, typemaps in file views, etc. Prior to MPI-4.0, many MPI routines used `int` in C
47 and `INTEGER` in Fortran as the type for *count* arguments. To avoid breaking back-
48 ward compatibility, this version of the standard continues to support `int` in C as

well as INTEGER in Fortran in such routines. In addition, this version of the standard supports using MPI_Count in C (via separate “_c” suffixed procedures) as well as INTEGER(KIND=MPI_COUNT_KIND) in Fortran (via polymorphic interfaces in newer MPI Fortran bindings (USE mpi_f08)) in such routines. See Section 19.2 for a full explanation. (*End of rationale.*)

The phrase **large count** refers to the use of MPI_Count and INTEGER(KIND=MPI_COUNT_KIND) parameter types.

There are cases where MPI_UNDEFINED can be returned in a **large count** OUT parameter. Per Table A.1.1 (page 819), the MPI_UNDEFINED constant is defined to be a C int (or unnamed enum) and a Fortran INTEGER. Implementations shall therefore choose the underlying types for MPI_Count and INTEGER(KIND=MPI_COUNT_KIND) such that they can be compared to MPI_UNDEFINED.

Advice to implementors. The comparison of MPI_UNDEFINED to an MPI_Count or INTEGER(KIND=MPI_COUNT_KIND) may need to be via a casting operation. (*End of advice to implementors.*)

2.6 Language Binding

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

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

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

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

2.6.1 Deprecated and Removed Interfaces

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

Table 2.1: Deprecated and removed constructs

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 ²	MPI-2.0		COMM_COPY_ATTR_FUNCTION ²
MPI_Delete_function ²	MPI-2.0		MPI_Comm_delete_attr_function ²
DELETE_FUNCTION ²	MPI-2.0		COMM_DELETE_ATTR_FUNCTION ²
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-4.0		no direct replacement
MPI_INFO_GET	MPI-4.0		MPI_INFO_GET_STRING
MPI_INFO_GET_VALUelen	MPI-4.0		MPI_INFO_GET_STRING
MPI_T_ERR_INVALID_ITEM	MPI-4.0		MPI_T_ERR_INVALID_INDEX
MPI_SIZEOF	MPI-4.0		storage_size() ⁵ or c_sizeof()

¹ Predefined datatype.
² Callback prototype definition.
³ Predefined callback routine.
⁴ Constant.
⁵ Fortran intrinsic. `storage_size()` returns the size in bits instead of bytes; see Section 16.3.
Other entries are regular MPI routines.

also provided, even though the new functions are equivalent to the old functions. The old names are deprecated.

Some of the previously deprecated constructs are now removed, as documented in Chapter 17. 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.

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 with TS 29113 and later if the `mpi_f08` module is used.

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

All MPI Fortran subroutines have a return code in the last argument. With `USE mpi_f08`, this last argument is declared as `OPTIONAL`, except for user-defined callback functions (e.g., `COMM_COPY_ATTR_FUNCTION`) and their predefined callbacks (e.g., `MPI_COMM_NULL_COPY_FN`). A few MPI operations that are functions do not have the return code argument. The return code value for successful completion is `MPI_SUCCESS`. Other error codes are implementation dependent; see the error codes in Chapter 9 and Annex A.

Constants representing the maximum length of a string are one smaller in Fortran than in C as discussed in Section 19.3.9.

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

Array arguments are indexed from one.

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

The support for large count and displacement in Fortran is only available when using newer MPI Fortran bindings (`USE mpi_f08`). For better readability, all Fortran large count procedure declarations are marked with a comment “!(_c)”.

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 nonzero 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 19.3.4, and no others, as macros in C.

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

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

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 11.6.

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 **transmission failures** in the communication system. If the MPI implementation is built on an unreliable underlying mechanism, then it is the job of the implementor of the MPI subsystem to insulate the user from this unreliability, and to reflect only unrecoverable transmission failures. Whenever possible, such failures will be reflected as errors in the relevant communication call.

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

Of course, MPI programs may still be erroneous. A **program error** can occur when an MPI call is made with an incorrect argument (nonexisting 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 themselves. Also, the user may provide user-defined error-handling routines, which will be invoked whenever an MPI call returns abnormally. The MPI error handling facilities are described in Section 9.3.

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; some faults (e.g., memory faults) may corrupt the state of the MPI library and its outputs; finally some errors may be “catastrophic” and may prevent MPI from returning control to the caller. On the other hand, some 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` when using the World Model (see Section 11.2). When `MPI_COMM_SELF` is not initialized (i.e., before `MPI_INIT` / `MPI_INIT_THREAD`, after `MPI_FINALIZE`, or when using the Sessions Model exclusively) the error raises the **initial error handler** (set during the launch operation, see 11.8.4). The Sessions Model is described in Section 11.3.

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 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

1 matching rules, and provide automatic type conversion in such situations. It will be helpful
 2 to generate warnings for such nonconforming behavior.

3 MPI defines a way for users to create new error codes as defined in Section 9.5.
 4

5 2.9 Implementation Issues 6

7 There are a number of areas where an MPI implementation may interact with the operating
 8 environment and system. While MPI does not mandate that any services (such as signal
 9 handling) be provided, it does strongly suggest the behavior to be provided if those services
 10 are available. This is an important point in achieving portability across platforms that
 11 provide the same set of services.
 12

13 2.9.1 Independence of Basic Runtime Routines 14

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

19 Note that this in no way prevents the creation of library routines that provide parallel
 20 services whose operation is collective. However, the following program is expected to com-
 21 plete in an ISO C environment regardless of the size of `MPI_COMM_WORLD` (assuming that
 22 `printf` is available at the executing nodes).
 23

```
24 int commworld_rank;  

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

26 MPI_Comm_rank(MPI_COMM_WORLD, &commworld_rank);  

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

28 MPI_Finalize();
```

29 The corresponding Fortran programs are also expected to complete.

30 An example of what is *not* required is any particular ordering of the action of these
 31 routines when called by several tasks. For example, MPI makes neither requirements nor
 32 recommendations for the output from the following program (again assuming that I/O is
 33 available at the executing nodes).
 34

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

36 printf("Output from MPI process where commworld_rank=%d\n",  

37     commworld_rank);
```

38 In addition, calls that fail because of resource exhaustion or other error are not con-
 39 sidered a violation of the requirements here (however, they are required to complete, just
 40 not to complete successfully).
 41

42 2.9.2 Interaction with Signals 43

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

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

2.10 Examples

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

DRAFT

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

DRAFT

Chapter 3

Point-to-Point Communication

3.1 Introduction

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

Example 3.1. A simple ‘hello world’ example usage of point-to-point communication.

```
#include "mpi.h"
int main(int argc, char *argv[])
{
    char message[20];
    int myrank;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    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 Example 3.1, process zero ($\text{myrank} = 0$) 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

1 (myrank = 1) receives this message with the *receive* operation `MPI_RECV`. The message to
 2 be received is selected according to the value of its *envelope*, and the *message data* is stored
 3 into the *receive buffer*. In the example above, the receive buffer consists of the storage
 4 containing the string `message` in the memory of process one. The first three parameters
 5 of the receive operation specify the location, size and type of the receive buffer. The next
 6 three parameters are used for selecting the incoming message. The last parameter is used
 7 to return information on the message just received.

8 The next sections describe the blocking send and receive operations. We discuss send,
 9 receive, blocking communication semantics, type matching requirements, type conversion in
 10 heterogeneous environments, and more general communication modes. Nonblocking com-
 11 munication is addressed next, followed by probing and cancelling a message, channel-like
 12 constructs and send-receive operations, ending with a description of the “dummy” process,
 13 `MPI_PROC_NULL`.

15 3.2 Blocking Send and Receive Operations

17 3.2.1 Blocking Send

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

21 `MPI_SEND(buf, count, datatype, dest, tag, comm)`

23	IN	<code>buf</code>	initial address of send buffer (choice)
24	IN	<code>count</code>	number of elements in send buffer (non-negative integer)
25			
26	IN	<code>datatype</code>	datatype of each send buffer element (handle)
27	IN	<code>dest</code>	rank of destination (integer)
28	IN	<code>tag</code>	message tag (integer)
29	IN	<code>comm</code>	communicator (handle)
30			
31			

33 C binding

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

36 `int MPI_Send_c(const void *buf, MPI_Count count, MPI_Datatype datatype,`
 37 `int dest, int tag, MPI_Comm comm)`

39 Fortran 2008 binding

40 `MPI_Send(buf, count, datatype, dest, tag, comm, ierror)`

41 `TYPE(*), DIMENSION(..), INTENT(IN) :: buf`

42 `INTEGER, INTENT(IN) :: count, dest, tag`

43 `TYPE(MPI_Datatype), INTENT(IN) :: datatype`

44 `TYPE(MPI_Comm), INTENT(IN) :: comm`

45 `INTEGER, OPTIONAL, INTENT(OUT) :: ierror`

46 `MPI_Send(buf, count, datatype, dest, tag, comm, ierror) !(_c)`

47 `TYPE(*), DIMENSION(..), INTENT(IN) :: buf`

48

Table 3.1: Predefined MPI datatypes corresponding to Fortran datatypes

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	

```

INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
INTEGER, INTENT(IN) :: dest, tag
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

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

```

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

3.2.2 Message Data

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

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

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

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 datatypes¹: `MPI_DOUBLE_COMPLEX` for double precision complex in Fortran declared to

¹These types, such as `DOUBLE COMPLEX` and `INTEGER*4`, are not specified by any Fortran standard but are

Table 3.2: Predefined MPI datatypes corresponding to C datatypes

MPI datatype	C datatype
MPI_CHAR	char (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 (treated as integral value)
MPI_UNSIGNED_CHAR	unsigned char (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 (defined in <stddef.h>) (treated as printable character)
MPI_C_BOOL	_Bool
MPI_INT8_T	int8_t
MPI_INT16_T	int16_t
MPI_INT32_T	int32_t
MPI_INT64_T	int64_t
MPI_UINT8_T	uint8_t
MPI_UINT16_T	uint16_t
MPI_UINT32_T	uint32_t
MPI_UINT64_T	uint64_t
MPI_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	

be of type DOUBLE COMPLEX; MPI_REAL2, MPI_REAL4, MPI_REAL8, and MPI_REAL16 for Fortran reals, declared to be of type REAL*2, REAL*4, REAL*8, and REAL*16, respectively; MPI_INTEGER1, MPI_INTEGER2, MPI_INTEGER4, and MPI_INTEGER8 for Fortran integers, declared to be of type INTEGER*1, INTEGER*2, INTEGER*4, and INTEGER*8, respectively; MPI_COMPLEX4, MPI_COMPLEX8, MPI_COMPLEX16, and

extensions commonly accepted by Fortran compilers.

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

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 3.4: Predefined MPI datatypes corresponding to C++ datatypes

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>

MPI_COMPLEX32 for complex numbers in Fortran declared to be of type COMPLEX*4, COMPLEX*8, COMPLEX*16, and COMPLEX*32, respectively; etc.

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

The datatypes MPI_AINT, MPI_OFFSET, and MPI_COUNT correspond to the MPI-defined C types MPI_Aint, MPI_Offset, and MPI_Count and their Fortran equivalents INTEGER(KIND=MPI_ADDRESS_KIND), INTEGER(KIND=MPI_OFFSET_KIND), and INTEGER(KIND=MPI_COUNT_KIND). This is described in Table 3.3. All predefined datatype handles are available in all language bindings. See Sections 19.3.6 and 19.3.10 on page 807 and 814 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.

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 procedure.

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 $0, \dots, \text{UB}$, 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 9. 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 7; 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 $0, \dots, n - 1 \cup \{\text{MPI_PROC_NULL}\}$, 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 7.)

When using the World Model (see Section 11.2), 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 World Model for MPI initialization, and 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 7. 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** procedure is given below.

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

OUT	<code>buf</code>	initial address of receive buffer (choice)
IN	<code>count</code>	number of elements in receive buffer (non-negative integer)
IN	<code>datatype</code>	datatype of each receive buffer element (handle)

IN	source	rank of source or MPI_ANY_SOURCE (integer)	1
IN	tag	message tag or MPI_ANY_TAG (integer)	2
IN	comm	communicator (handle)	3
OUT	status	status object (status)	4

C binding

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

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

Fortran 2008 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
```

```
MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror) !(_c)
  TYPE(*), DIMENSION(..) :: buf
  INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  INTEGER, INTENT(IN) :: source, tag
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Status) :: status
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran 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

1 procedure will return an error code. A quality implementation will also ensure that
2 no memory that is outside the receive buffer will ever be overwritten.

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

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

18 The message tag is specified by the **tag** argument of the receive operation. The
19 argument **source**, if different from MPI_ANY_SOURCE, is specified as a rank within the
20 process group associated with that same communicator (remote process group, for inter-
21 communicators). Thus, the range of valid values for the **source** argument is $\{0, \dots, n-1\} \cup$
22 $\{\text{MPI_ANY_SOURCE}\} \cup \{\text{MPI_PROC_NULL}\}$, where n is the number of processes in this group.

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

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

31
32 *Advice to implementors.* Message context and other communicator information can
33 be implemented as an additional tag field. It differs from the regular message tag
34 in that wild card matching is not allowed on this field, and that value setting for
35 this field is controlled by communicator manipulation functions. (*End of advice to*
36 *implementors.*)

37
38 The use of **dest** = MPI_PROC_NULL or **source** = MPI_PROC_NULL to define a “dummy”
39 destination or source in any send or receive call is described in Section 3.10.

40 41 3.2.5 Return Status

42 The source or tag of a received message may not be known if wildcard values were used
43 in the receive operation. Also, if multiple requests are completed by a single MPI function
44 (see Section 3.7.5), a distinct error code may need to be returned for each request. The
45 information is returned by the **status** argument of MPI_RECV. The type of **status** is MPI-
46 defined. Status variables need to be explicitly allocated by the user, that is, they are not
47 system objects.
48

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 19.3.5.

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

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

In general, message-passing calls do not modify the value of the error code field of `status` variables. This field may be updated only by the functions in Section 3.7.5 that 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	<code>status</code>	return status of receive operation (<code>status</code>)
IN	<code>datatype</code>	<code>datatype</code> of each receive buffer entry (<code>handle</code>)
OUT	<code>count</code>	number of received entries (<code>integer</code>)

C binding

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

```

1 int MPI_Get_count_c(const MPI_Status *status, MPI_Datatype datatype,
2                   MPI_Count *count)
3

```

Fortran 2008 binding

```

4 MPI_Get_count(status, datatype, count, ierror)
5     TYPE(MPI_Status), INTENT(IN) :: status
6     TYPE(MPI_Datatype), INTENT(IN) :: datatype
7     INTEGER, INTENT(OUT) :: count
8     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10 MPI_Get_count(status, datatype, count, ierror) !(_c)
11     TYPE(MPI_Status), INTENT(IN) :: status
12     TYPE(MPI_Datatype), INTENT(IN) :: datatype
13     INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: count
14     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15

```

Fortran binding

```

16 MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)
17     INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
18

```

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 5.1.11.

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` procedures described in this section.

3.2.6 Passing `MPI_STATUS_IGNORE` for Status

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

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

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

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

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

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.

3.2.7 Blocking Send-Receive

The **send-receive** operations combine in one operation 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 procedures described in Chapter 8 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 (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 (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)
```

```
int MPI_Sendrecv_c(const void *sendbuf, MPI_Count sendcount,
                   MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf,
                   MPI_Count recvcount, MPI_Datatype recvtype, int source,
                   int recvtag, MPI_Comm comm, MPI_Status *status)
```

Fortran 2008 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
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount,
             recvtype, source, recvtag, comm, status, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
INTEGER, INTENT(IN) :: dest, sendtag, source, recvtag
TYPE(*), DIMENSION(..) :: recvbuf
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Status) :: status
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

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

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

```

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

```

INOUT	buf	initial address of send and receive buffer (choice)
IN	count	number of elements in send and receive buffer (non-negative integer)
IN	datatype	type of elements in send and receive buffer (handle)
IN	dest	rank of destination (integer)
IN	sendtag	send message tag (integer)
IN	source	rank of source or MPI_ANY_SOURCE (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)

```

```

1 int MPI_Sendrecv_replace_c(void *buf, MPI_Count count, MPI_Datatype datatype,
2     int dest, int sendtag, int source, int recvtag, MPI_Comm comm,
3     MPI_Status *status)
4

```

Fortran 2008 binding

```

5 MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
6     comm, status, ierror)
7     TYPE(*), DIMENSION(..) :: buf
8     INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag
9     TYPE(MPI_Datatype), INTENT(IN) :: datatype
10    TYPE(MPI_Comm), INTENT(IN) :: comm
11    TYPE(MPI_Status) :: status
12    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13

```

```

14 MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
15     comm, status, ierror) !(_c)
16    TYPE(*), DIMENSION(..) :: buf
17    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
18    TYPE(MPI_Datatype), INTENT(IN) :: datatype
19    INTEGER, INTENT(IN) :: dest, sendtag, source, recvtag
20    TYPE(MPI_Comm), INTENT(IN) :: comm
21    TYPE(MPI_Status) :: status
22    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23

```

Fortran binding

```

24 MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
25     COMM, STATUS, IERROR)
26    <type> BUF(*)
27    INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM,
28     STATUS(MPI_STATUS_SIZE), IERROR
29

```

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

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

3.3 Datatype Matching and Data Conversion

3.3.1 Type Matching Rules

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

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

Type matching has to be observed at each of these three phases: The type of each variable in the sender buffer has to match the type specified for that entry by the send

operation; the type specified by the send operation has to match the type specified by the receive operation; and the type of each variable in the receive buffer has to match the type specified for that entry by the receive operation. A program that fails to observe these three rules is *erroneous*.

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

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

The type of a variable in a host program matches the type specified in the 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 5.2. The type MPI_BYTE allows one to transfer the binary value of a byte in memory unchanged.

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

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

The following examples illustrate the first two cases.

Example 3.2. Sender and receiver specify matching types.

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

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

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

```
! ----- THIS EXAMPLE IS ERRONEOUS -----
```

```

1 CALL MPI_COMM_RANK(comm, rank, ierr)
2 IF (rank .EQ. 0) THEN
3   CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
4 ELSE IF (rank .EQ. 1) THEN
5   CALL MPI_RECV(b(1), 40, MPI_BYTE, 0, tag, comm, status, ierr)
6 END IF

```

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

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

```

12 CALL MPI_COMM_RANK(comm, rank, ierr)
13 IF (rank .EQ. 0) THEN
14   CALL MPI_SEND(a(1), 40, MPI_BYTE, 1, tag, comm, ierr)
15 ELSE IF (rank .EQ. 1) THEN
16   CALL MPI_RECV(b(1), 60, MPI_BYTE, 0, tag, comm, status, ierr)
17 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 3.5. Transfer of Fortran `CHARACTER`s.

```

38 CHARACTER*10 a
39 CHARACTER*10 b
40
41 CALL MPI_COMM_RANK(comm, rank, ierr)
42 IF (rank .EQ. 0) THEN
43   CALL MPI_SEND(a, 5, MPI_CHARACTER, 1, tag, comm, ierr)
44 ELSE IF (rank .EQ. 1) THEN
45   CALL MPI_RECV(b(6:10), 5, MPI_CHARACTER, 0, tag, comm, status, ierr)
46 END IF

```

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

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

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

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

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

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.)

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

3 Consider the three examples, 3.2–3.4. The first program is correct, assuming that **a** and
4 **b** are REAL arrays of size ≥ 10 . If the sender and receiver execute in different environments,
5 then the ten real values that are fetched from the send buffer will be converted to the
6 representation for reals on the receiver site before they are stored in the receive buffer.
7 While the number of real elements fetched from the send buffer equal the number of real
8 elements stored in the receive buffer, the number of bytes stored need not equal the number
9 of bytes loaded. For example, the sender may use a four byte representation and the receiver
10 an eight byte representation for reals.

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

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

19 Data representation conversion also applies to the *envelope* of a message: source, des-
20 tination and tag are all integers that may need to be converted.

21
22 *Advice to implementors.* The current definition does not require messages to carry
23 data type information. Both sender and receiver provide complete data type infor-
24 mation. In a heterogeneous environment, one can either use a machine independent
25 encoding such as XDR, or have the receiver convert from the sender representation
26 to its own, or even have the sender do the conversion.

27 Additional type information might be added to messages in order to allow the sys-
28 tem to detect mismatches between datatype at sender and receiver. This might be
29 particularly useful in a slower but safer debug mode. (*End of advice to implementors.*)
30

31 MPI requires support for inter-language communication, e.g., if messages are sent using
32 an MPI procedure from the MPI C language interface and received using an MPI procedure
33 from one of the MPI Fortran language interfaces. The behavior is defined in Section 19.3.
34

35 3.4 Communication Modes

36
37 The send call described in Section 3.2.1 is *blocking*: it does not return until the *message*
38 *data* and *envelope* have been safely stored away so that the sender is free to modify the
39 send buffer. The message might be copied directly into the matching receive buffer, or it
40 might be copied into a temporary system buffer.

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

47 The send call described in Section 3.2.1 uses the **standard** communication mode. In
48 this mode, it is up to MPI to decide whether outgoing messages will be buffered. MPI may

buffer outgoing messages. In such a case, the send call may complete before a matching receive is invoked. On the other hand, buffer space may be unavailable, or MPI may choose not to buffer outgoing messages, for performance reasons. In this case, the send call will not complete until a matching receive has been posted, and the data has been moved to the receiver.

Thus, a *standard mode send* can be *started* whether or not a matching receive has been posted. It may *complete* before a matching receive is posted. The standard mode send is *nonlocal*: 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 *nonlocal*.

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 protocol that is otherwise required and results in improved performance. The completion of the send operation does not depend on the status of a matching receive, and merely indicates that the send buffer can be reused. A send operation that uses the ready mode has the same semantics as a standard send operation, or a synchronous send operation; it is merely that the sender provides additional information to the system (namely that a matching receive is already posted), that can save some overhead. In a correct program, therefore, a ready send could be replaced by a standard send with no effect on the behavior of the program other than performance.

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

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

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

C binding

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

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

Fortran 2008 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(buf, count, datatype, dest, tag, comm, ierror) !(_c)
  TYPE(*), DIMENSION(..), INTENT(IN) :: buf
  INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  INTEGER, INTENT(IN) :: dest, tag
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

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

Send in buffered mode.

According to the definitions in Section 2.4.2, MPI_BSEND is a completing procedure and the user can re-use all resources given as arguments, including the *message data buffer*. It is also a local procedure because it returns immediately without depending on the execution of any MPI procedure in any other MPI process.

Advice to users. This is one of the exceptions in which a completing and therefore blocking operation-related procedure is local. (*End of advice to users.*)

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

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

C binding

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

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

Fortran 2008 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
```

```
MPI_Ssend(buf, count, datatype, dest, tag, comm, ierror) !(_c)
  TYPE(*), DIMENSION(..), INTENT(IN) :: buf
  INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  INTEGER, INTENT(IN) :: dest, tag
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

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

Send in synchronous mode.

```

1 MPI_RSEND(buf, count, datatype, dest, tag, comm)
2     IN      buf                initial address of send buffer (choice)
3
4     IN      count              number of elements in send buffer (non-negative
5                                integer)
6
7     IN      datatype           datatype of each send buffer element (handle)
8
9     IN      dest                rank of destination (integer)
10
11    IN      tag                  message tag (integer)
12
13    IN      comm                 communicator (handle)

```

C binding

```

13 int MPI_Rsend(const void *buf, int count, MPI_Datatype datatype, int dest,
14               int tag, MPI_Comm comm)
15

```

```

16 int MPI_Rsend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,
17                 int dest, int tag, MPI_Comm comm)
18

```

Fortran 2008 binding

```

19 MPI_Rsend(buf, count, datatype, dest, tag, comm, ierror)
20

```

```

21   TYPE(*), DIMENSION(..), INTENT(IN) :: buf
22   INTEGER, INTENT(IN) :: count, dest, tag
23   TYPE(MPI_Datatype), INTENT(IN) :: datatype
24   TYPE(MPI_Comm), INTENT(IN) :: comm
25   INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

26 MPI_Rsend(buf, count, datatype, dest, tag, comm, ierror) !(_c)
27

```

```

28   TYPE(*), DIMENSION(..), INTENT(IN) :: buf
29   INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
30   TYPE(MPI_Datatype), INTENT(IN) :: datatype
31   INTEGER, INTENT(IN) :: dest, tag
32   TYPE(MPI_Comm), INTENT(IN) :: comm
33   INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

34 MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
35

```

```

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

```

38 Send in ready mode.

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

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

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

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

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

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

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

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

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

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

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

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

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 **nonovertaking**: 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 3.6. An example of nonovertaking 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 receive operations have been initiated, then at least one of these two operations will complete, independently of other actions in the system: the send operation will complete, unless the receive is satisfied by another message, and completes; the receive operation will complete, unless the message sent is consumed by another matching receive that was posted at the same destination process.

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

```

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 3.8. 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 3.9. An errant attempt to exchange messages.

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

```

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

```

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

Example 3.10. An exchange that relies on buffering.

```

13  ! ----- THIS EXAMPLE IS ERRONEOUS -----
14  CALL MPI_COMM_RANK(comm, rank, ierr)
15  IF (rank .EQ. 0) THEN
16      CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
17      CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
18  ELSE IF (rank .EQ. 1) THEN
19      CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
20      CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
21  END IF

```

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

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

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

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

Nonblocking message-passing operations, as described in Section 3.7, can be used to avoid the need for buffering outgoing messages. This prevents deadlocks due to lack

of buffer space, and improves performance, by allowing overlap of computation and communication, and avoiding the overheads of allocating buffers and copying messages into buffers. (*End of advice to users.*)

3.6 Buffer Allocation and Usage

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

`MPI_BUFFER_ATTACH(buffer, size)`

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

C binding

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

`int MPI_Buffer_attach_c(void *buffer, MPI_Count size)`

Fortran 2008 binding

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

`MPI_Buffer_attach(buffer, size, ierror) !(_c)`
 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: size
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran 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 19.1.12).

`MPI_BUFFER_DETACH(buffer_addr, size)`

OUT	buffer_addr	initial buffer address (choice)
OUT	size	buffer size, in bytes (integer)

C binding

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

```
1 int MPI_Buffer_detach_c(void *buffer_addr, MPI_Count *size)
```

3 Fortran 2008 binding

```
4 MPI_Buffer_detach(buffer_addr, size, ierror)
5     USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
6     TYPE(C_PTR), INTENT(OUT) :: buffer_addr
7     INTEGER, INTENT(OUT) :: size
8     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10 MPI_Buffer_detach(buffer_addr, size, ierror) !(_c)
11     USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
12     TYPE(C_PTR), INTENT(OUT) :: buffer_addr
13     INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
14     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

15 Fortran binding

```
16 MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)
17     <type> BUFFER_ADDR(*)
18     INTEGER SIZE, IERROR
```

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

23 If the size of the detached buffer cannot be represented in `size`, it is set to
24 `MPI_UNDEFINED`.

25 **Example 3.11.** Calls to attach and detach buffers.

```
27 #define BUFFSIZE 10000
28 int size;
29 char *buff;
30 MPI_Buffer_attach(malloc(BUFFSIZE), BUFFSIZE);
31 /* a buffer of 10000 bytes can now be used by MPI_Bsend */
32 MPI_Buffer_detach(&buff, &size);
33 /* Buffer size reduced to zero */
34 MPI_Buffer_attach(buff, size);
35 /* Buffer of 10000 bytes available again */
```

36 *Advice to users.* Even though the C functions `MPI_Buffer_attach` and
37 `MPI_Buffer_detach` both have a first argument of type `void*`, these arguments are
38 used differently: A pointer to the buffer is passed to `MPI_Buffer_attach`; the address
39 of the pointer is passed to `MPI_Buffer_detach`, so that this call can return the pointer
40 value. In Fortran with the `mpi` module or `mpif.h`, the type of the `buffer_addr` argument
41 is wrongly defined and the argument is therefore unused. In Fortran with the `mpi_f08`
42 module, the address of the buffer is returned as `TYPE(C_PTR)`, see also Example 9.1
43 about the use of `C_PTR` pointers. (*End of advice to users.*)

44 *Rationale.* Both arguments are defined to be of type `void*` (rather than `void*` and
45 `void**`, respectively), so as to avoid complex type casts. E.g., in the last example,
46 `&buff`, which is of type `char**`, can be passed as argument to `MPI_Buffer_detach`
47
48

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 5.2 and the nonblocking communication functions described in Section 3.7.

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

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

- Traverse sequentially the PME queue from head towards the tail, deleting all entries for communications that have completed, up to the first entry with an uncompleted request; update queue head to point to that entry.
- Compute the number, n , of bytes needed to store an entry for the new message. An upper bound on n can be computed as follows: A call to the function `MPI_PACK_SIZE(count, datatype, comm, size)`, with the `count`, `datatype` and `comm` arguments used in the `MPI_BSEND` call, returns an upper bound on the amount of space needed to buffer the message data (see Section 5.2). The MPI constant `MPI_BSEND_OVERHEAD` provides an upper bound on the additional space consumed by the entry (e.g., for pointers or *envelope* information).

- 1 • Find the next contiguous empty space of n bytes in buffer (space following queue tail,
2 or space at start of buffer if queue tail is too close to end of buffer). If space is not
3 found then raise buffer overflow error.
- 4
- 5 • Append to end of PME queue in contiguous space the new entry that contains request
6 handle, next pointer and packed message data; `MPI_PACK` is used to pack data.
- 7
- 8 • Post nonblocking send (standard mode) for packed data.
- 9
- Return

11 3.7 Nonblocking Communication

13 **Nonblocking communication** is important both for reasons of correctness and perfor-
14 mance. For complex communication patterns, the use of only blocking communication
15 (without buffering) is difficult because the programmer must ensure that each send is
16 matched with a receive in an order that avoids *deadlock*. For communication patterns that
17 are determined only at run time, this is even more difficult. Nonblocking communication
18 can be used to avoid this problem, allowing programmers to express complex and possibly
19 dynamic communication patterns without needing to ensure that all sends and receives
20 are issued in an order that prevents deadlock (see Section 3.5 and the discussion of “safe”
21 programs). Nonblocking communication also allows for the *overlap* of communication with
22 different communication operations, e.g., to prevent the *serialization* of such operations,
23 and for the *overlap* of communication with computation. Whether an implementation is
24 able to accomplish an effective (from a performance standpoint) overlap of operations de-
25 pends on the implementation itself and the system on which the implementation is running.
26 Using nonblocking operations *permits* an implementation to overlap communication with
27 computation, but does not require it to do so.

28 A nonblocking **send start** call *initiates* the send operation, but does not complete it.
29 The send start call can return before the message was copied out of the send buffer. A
30 separate **send complete** call is needed to complete the communication, i.e., to verify that
31 the data has been copied out of the send buffer. With suitable hardware, the transfer of data
32 out of the sender memory may proceed concurrently with computations done at the sender
33 after the send was initiated and before it completed. Similarly, a nonblocking **receive start**
34 call *initiates* the receive operation, but does not complete it. The call can return before a
35 message is stored into the receive buffer. A separate **receive complete** call is needed to
36 complete the receive operation and verify that the data has been received into the receive
37 buffer. With suitable hardware, the transfer of data into the receiver memory may proceed
38 concurrently with computations done after the receive was initiated and before it completed.
39 The use of nonblocking receives may also avoid system buffering and memory-to-memory
40 copying, as information is provided early on the location of the receive buffer.

41 Nonblocking send start calls can use the same four modes as blocking sends: *standard*,
42 *buffered*, *synchronous*, and *ready*. These carry the same meaning. Sends of all modes, *ready*
43 excepted, can be started whether a matching receive has been posted or not; a nonblocking
44 **ready** send can be started only if a matching receive is posted. In all cases, the send start
45 call is *local*: it returns immediately, irrespective of the status of other processes. If the call
46 causes some system resource to be exhausted, then it will fail and return an error code.
47 Quality implementations of MPI should ensure that this happens only in “pathological”
48

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 *nonlocal*. Note that a synchronous, nonblocking send may complete, if matched by a nonblocking receive, before the receive complete call occurs. (It can complete as soon as the sender “knows” the transfer will complete, but before the receiver “knows” the transfer will complete.)

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

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

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

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

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

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

3.7.1 Communication Request Objects

Nonblocking communications use opaque **request** objects to identify communication operations and match the operation that initiates the communication with the operation that terminates it. These are system objects that are accessed via a handle. A request object identifies various properties of a communication operation, such as the send mode, the communication buffer that is associated with it, its context, the tag and destination arguments

to be used for a send, or the tag and source arguments to be used for a receive. In addition, this object stores information about the status of the pending communication operation.

3.7.2 Communication Initiation

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

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 (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)
```

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

Fortran 2008 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
```

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

Fortran 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 (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)

```

```

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

```

Fortran 2008 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

```

```

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

```

Fortran 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.

```

1 MPI_ISSEND(buf, count, datatype, dest, tag, comm, request)
2   IN      buf          initial address of send buffer (choice)
3
4   IN      count       number of elements in send buffer (non-negative
5                          integer)
6
7   IN      datatype    datatype of each send buffer element (handle)
8
9   IN      dest        rank of destination (integer)
10
11  IN      tag          message tag (integer)
12
13  IN      comm         communicator (handle)
14
15  OUT     request      communication request (handle)

```

C binding

```

14 int MPI_Issend(const void *buf, int count, MPI_Datatype datatype, int dest,
15               int tag, MPI_Comm comm, MPI_Request *request)
16

```

```

17 int MPI_Issend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,
18                 int dest, int tag, MPI_Comm comm, MPI_Request *request)
19

```

Fortran 2008 binding

```

20 MPI_Issend(buf, count, datatype, dest, tag, comm, request, ierror)
21   TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
22   INTEGER, INTENT(IN) :: count, dest, tag
23   TYPE(MPI_Datatype), INTENT(IN) :: datatype
24   TYPE(MPI_Comm), INTENT(IN) :: comm
25   TYPE(MPI_Request), INTENT(OUT) :: request
26   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27

```

```

28 MPI_Issend(buf, count, datatype, dest, tag, comm, request, ierror) !(_c)
29   TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
30   INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
31   TYPE(MPI_Datatype), INTENT(IN) :: datatype
32   INTEGER, INTENT(IN) :: dest, tag
33   TYPE(MPI_Comm), INTENT(IN) :: comm
34   TYPE(MPI_Request), INTENT(OUT) :: request
35   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36

```

Fortran binding

```

37 MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
38

```

```

39   <type> BUF(*)

```

```

40   INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

```

```

41   Start a synchronous mode nonblocking send.
42
43
44
45
46
47
48

```

MPI_IRSEND(buf, count, datatype, dest, tag, comm, request)	1
IN buf	2
	3
IN count	4
	5
IN datatype	6
IN dest	7
	8
IN tag	9
IN comm	10
OUT request	11
	12
	13
C binding	14
int MPI_Irsend(const void *buf, int count, MPI_Datatype datatype, int dest,	15
int tag, MPI_Comm comm, MPI_Request *request)	16
int MPI_Irsend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,	17
int dest, int tag, MPI_Comm comm, MPI_Request *request)	18
	19
Fortran 2008 binding	20
MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror)	21
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf	22
INTEGER, INTENT(IN) :: count, dest, tag	23
TYPE(MPI_Datatype), INTENT(IN) :: datatype	24
TYPE(MPI_Comm), INTENT(IN) :: comm	25
TYPE(MPI_Request), INTENT(OUT) :: request	26
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	27
MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror) !(_c)	28
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf	29
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count	30
TYPE(MPI_Datatype), INTENT(IN) :: datatype	31
INTEGER, INTENT(IN) :: dest, tag	32
TYPE(MPI_Comm), INTENT(IN) :: comm	33
TYPE(MPI_Request), INTENT(OUT) :: request	34
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	35
	36
Fortran binding	37
MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	38
<type> BUF(*)	39
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR	40
	41
Start a ready mode nonblocking send.	42
	43
	44
	45
	46
	47
	48

```

1 MPI_Irecv(buf, count, datatype, source, tag, comm, request)
2   OUT    buf                initial address of receive buffer (choice)
3
4   IN     count              number of elements in receive buffer (non-negative
5                               integer)
6
7   IN     datatype           datatype of each receive buffer element (handle)
8
9   IN     source              rank of source or MPI_ANY_SOURCE (integer)
10
11  IN     tag                  message tag or MPI_ANY_TAG (integer)
12
13  IN     comm                 communicator (handle)
14
15  OUT    request              communication request (handle)

```

C binding

```

14 int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
15             MPI_Comm comm, MPI_Request *request)
16

```

```

17 int MPI_Irecv_c(void *buf, MPI_Count count, MPI_Datatype datatype, int source,
18               int tag, MPI_Comm comm, MPI_Request *request)
19

```

Fortran 2008 binding

```

20 MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror)
21   TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
22   INTEGER, INTENT(IN) :: count, source, tag
23   TYPE(MPI_Datatype), INTENT(IN) :: datatype
24   TYPE(MPI_Comm), INTENT(IN) :: comm
25   TYPE(MPI_Request), INTENT(OUT) :: request
26   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27

```

```

28 MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror) !(_c)
29   TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
30   INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
31   TYPE(MPI_Datatype), INTENT(IN) :: datatype
32   INTEGER, INTENT(IN) :: source, tag
33   TYPE(MPI_Comm), INTENT(IN) :: comm
34   TYPE(MPI_Request), INTENT(OUT) :: request
35   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36

```

Fortran binding

```

37 MPI_Irecv(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
38

```

```

39   <type> BUF(*)

```

```

40   INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR

```

```

41   Start a nonblocking receive.
42
43
44
45
46
47
48

```


MPI_ISENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype,			1
source, recvtag, comm, request)			2
			3
IN sendbuf	initial address of send buffer (choice)		4
IN sendcount	number of elements in send buffer (non-negative integer)		5
			6
IN sendtype	datatype of each send buffer element (handle)		7
IN dest	rank of destination (integer)		8
			9
IN sendtag	send tag (integer)		10
OUT recvbuf	initial address of receive buffer (choice)		11
IN recvcount	number of elements in receive buffer (non-negative integer)		12
			13
IN recvtype	datatype of each receive buffer element (handle)		14
IN source	rank of source or MPI_ANY_SOURCE (integer)		15
IN recvtag	receive tag or MPI_ANY_TAG (integer)		16
			17
IN comm	communicator (handle)		18
OUT request	communication request (handle)		19
			20
			21
			22
C binding			23
int MPI_Isendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,			24
int dest, int sendtag, void *recvbuf, int recvcount,			25
MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm,			26
MPI_Request *request)			27
int MPI_Isendrecv_c(const void *sendbuf, MPI_Count sendcount,			28
MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf,			29
MPI_Count recvcount, MPI_Datatype recvtype, int source,			30
int recvtag, MPI_Comm comm, MPI_Request *request)			31
			32
Fortran 2008 binding			33
MPI_Isendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount,			34
recvtype, source, recvtag, comm, request, ierror)			35
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf			36
INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source, recvtag			37
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype			38
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf			39
TYPE(MPI_Comm), INTENT(IN) :: comm			40
TYPE(MPI_Request), INTENT(OUT) :: request			41
INTEGER, OPTIONAL, INTENT(OUT) :: ierror			42
MPI_Isendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount,			43
recvtype, source, recvtag, comm, request, ierror) !(_c)			44
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf			45
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount			46
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype			47
INTEGER, INTENT(IN) :: dest, sendtag, source, recvtag			48

```

1     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
2     TYPE(MPI_Comm), INTENT(IN) :: comm
3     TYPE(MPI_Request), INTENT(OUT) :: request
4     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

6 MPI_ISENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF, RECVCOUNT,
7             RECVTYPE, SOURCE, RECVTAG, COMM, REQUEST, IERROR)
8     <type> SENDBUF(*), RECVBUF(*)
9     INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE, SOURCE,
10            RECVTAG, COMM, REQUEST, IERROR

```

Initiate a nonblocking communication request for a *send and receive* operation.

```

14 MPI_ISENDRECV_REPLACE(buf, count, datatype, dest, sendtag, source, recvtag, comm,
15                       request)

```

17	INOUT	buf	initial address of send and receive buffer (choice)
18	IN	count	number of elements in send and receive buffer (non-negative integer)
19			
20			
21	IN	datatype	type of elements in send and receive buffer (handle)
22	IN	dest	rank of destination (integer)
23			
24	IN	sendtag	send message tag (integer)
25	IN	source	rank of source or MPI_ANY_SOURCE (integer)
26	IN	recvtag	receive message tag or MPI_ANY_TAG (integer)
27			
28	IN	comm	communicator (handle)
29	OUT	request	communication request (handle)

C binding

```

32 int MPI_Isendrecv_replace(void *buf, int count, MPI_Datatype datatype,
33                          int dest, int sendtag, int source, int recvtag, MPI_Comm comm,
34                          MPI_Request *request)

```

```

35 int MPI_Isendrecv_replace_c(void *buf, MPI_Count count, MPI_Datatype datatype,
36                          int dest, int sendtag, int source, int recvtag, MPI_Comm comm,
37                          MPI_Request *request)

```

Fortran 2008 binding

```

40 MPI_Isendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
41                       comm, request, ierror)
42     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
43     INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag
44     TYPE(MPI_Datatype), INTENT(IN) :: datatype
45     TYPE(MPI_Comm), INTENT(IN) :: comm
46     TYPE(MPI_Request), INTENT(OUT) :: request
47     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

MPI_Isendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
                      comm, request, ierror) !(_c)
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
INTEGER, INTENT(IN) :: dest, sendtag, source, recvtag
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

MPI_ISENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
                     COMM, REQUEST, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, REQUEST,
IERROR

```

Initiate a nonblocking communication request for a *send and receive* operation. The same buffer is used both for the send and for the receive, so that the message sent is replaced by the message received.

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

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

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

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

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 communication request* and the handle to it are **inactive**

if the request is not associated with any ongoing communication (see Section 3.9). A handle is **active** if it is neither *null* nor *inactive*. An **empty** status is a status that 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)
```

Fortran 2008 binding

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

Fortran 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 communication request*, it is marked *inactive*. Any other type of request is deallocated and the request handle is set to `MPI_REQUEST_NULL`. **When the operation represented by the request is sufficiently enabled, then a call to `MPI_WAIT` is a local procedure call, otherwise it is a nonlocal procedure call.**

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

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

Advice to users. Successful return of `MPI_WAIT` after a `MPI_IBSEND` implies that 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)
```

Fortran 2008 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
```

Fortran 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 communication 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* procedure.

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 procedure returns with `flag = true` and *empty* status.

The procedures `MPI_WAIT` and `MPI_TEST` can be used to complete any request-based nonblocking or persistent operation.

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 3.12. Simple usage of nonblocking operations and MPI_WAIT.

```

1  CALL MPI_COMM_RANK(comm, rank, ierr)
2
3  CALL MPI_COMM_RANK(comm, rank, ierr)
4  IF (rank .EQ. 0) THEN
5      CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)
6      ! **** do some computation to mask latency ****
7      CALL MPI_WAIT(request, status, ierr)
8  ELSE IF (rank .EQ. 1) THEN
9      CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr)
10     ! **** do some computation to mask latency ****
11     CALL MPI_WAIT(request, status, ierr)
12 END IF

```

A request object can be *freed* using the following MPI procedure.

MPI_REQUEST_FREE(request)

INOUT request communication request (handle)

C binding

int MPI_Request_free(MPI_Request *request)

Fortran 2008 binding

```

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

```

Fortran binding

```

MPI_REQUEST_FREE(REQUEST, IERROR)
  INTEGER REQUEST, IERROR

```

MPI_REQUEST_FREE is a *local* procedure. Upon successful return, MPI_REQUEST_FREE sets request to MPI_REQUEST_NULL. For an *inactive* request representing any type of MPI operation, MPI_REQUEST_FREE shall do the *freeing stage* of the associated operation during its execution.

For a request representing a *nonblocking* point-to-point or a persistent point-to-point operation, it is permitted (although strongly discouraged) to call MPI_REQUEST_FREE when the request is *active*. In this special case, MPI_REQUEST_FREE will only mark the request for freeing and MPI will actually do the *freeing stage* of the associated operation later.

The use of this procedure for generalized requests is described in Section 13.2.

Calling MPI_REQUEST_FREE with an *active* request representing any other type of MPI operation (e.g., any partitioned operation (see Chapter 4), any collective operation (see Chapter 6), any I/O operation (see Chapter 14), or any request-based RMA operation (see Chapter 12)) is *erroneous*.

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

Advice to users. Once a request is freed by a call to MPI_REQUEST_FREE, it is not

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

Example 3.13. An example using `MPI_REQUEST_FREE`.

```

CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
IF (rank .EQ. 0) THEN
  DO i=1,n
    CALL MPI_ISEND(outval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)
    CALL MPI_REQUEST_FREE(req, ierr)
    CALL MPI_Irecv(ival, 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(ival, 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(ival, 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 **nonovertaking** requirement of Section 3.5 is extended to nonblocking communication, with this definition of order being used.

Example 3.14. Message ordering for nonblocking operations.

```

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

```

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

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

Example 3.15. An illustration of progress semantics.

```
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_WAIT SOME` or `MPI_TEST SOME` can be used to *complete* all enabled operations in a list.

`MPI_WAITANY(count, array_of_requests, index, status)`

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

C binding

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

Fortran 2008 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
```

Fortran 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 communication 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` with an array containing multiple entries has the same effect as the execution of `MPI_WAIT` with the array entry indicated by the output value of *index* (unless the output 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 (non-negative integer)
INOUT	array_of_requests	array of requests (array of handles)
OUT	index	index of operation that completed or <code>MPI_UNDEFINED</code> 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)
```

Fortran 2008 binding

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

```

1     INTEGER, INTENT(OUT) :: index
2     LOGICAL, INTENT(OUT) :: flag
3     TYPE(MPI_Status) :: status
4     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

6 MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)
7     INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE), IERROR
8     LOGICAL FLAG
9

```

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

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

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

```

27 MPI_WAITALL(count, array_of_requests, array_of_statuses)
28
29     IN        count                list length (non-negative integer)
30     INOUT    array_of_requests    array of requests (array of handles)
31     OUT      array_of_statuses    array of status objects (array of status)
32
33

```

C binding

```

35 int MPI_Waitall(int count, MPI_Request array_of_requests[],
36                MPI_Status array_of_statuses[])
37

```

Fortran 2008 binding

```

38 MPI_Waitall(count, array_of_requests, array_of_statuses, ierror)
39     INTEGER, INTENT(IN) :: count
40     TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
41     TYPE(MPI_Status) :: array_of_statuses(*)
42     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43

```

Fortran binding

```

44 MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)
45     INTEGER COUNT, ARRAY_OF_REQUESTS(*), ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *),
46     IERROR
47
48

```

Blocks until all communication operations associated with *active* handles in the list *complete*, and returns 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` has the same effect as the execution of `MPI_WAIT` for each of the array elements 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	list length (non-negative integer)
INOUT	array_of_requests	array of requests (array of handles)
OUT	flag	true if all of the operations are complete (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[])
```

Fortran 2008 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
```

Fortran 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* procedure.

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

`MPI_WAITSSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)`

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

C binding

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

Fortran 2008 binding

```
MPI_Waitssome(incount, array_of_requests, outcount, array_of_indices,
              array_of_statuses, ierror)
```

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

Fortran binding

```
MPI_WAITSSOME(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_statuses 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_WAITSSOME 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_TESTSSOME(`incount`, `array_of_requests`, `outcount`, `array_of_indices`, `array_of_statuses`)

IN	<code>incount</code>	length of <code>array_of_requests</code> (non-negative integer)
INOUT	<code>array_of_requests</code>	array of requests (array of handles)
OUT	<code>outcount</code>	number of completed requests (integer)
OUT	<code>array_of_indices</code>	array of indices of operations that completed (array of integers)
OUT	<code>array_of_statuses</code>	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[])
```

Fortran 2008 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(incount)
INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
TYPE(MPI_Status) :: array_of_statuses(*)
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_TESTSSOME(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_WAITSSOME, 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`

1 = MPI_UNDEFINED.

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

8 Errors that occur during the execution of MPI_TEST SOME are handled as for
 9 MPI_WAIT SOME.

10
 11 *Advice to users.* The use of MPI_TEST SOME is likely to be more efficient than the use
 12 of MPI_TEST ANY. The former returns information on all *completed* communications,
 13 with the latter, a new call is required for each communication that completes.

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

19
 20 *Advice to implementors.* MPI_TEST SOME should *complete* as many pending com-
 21 munications as possible. (*End of advice to implementors.*)

22
 23 **Example 3.16.** Client-server code (starvation can occur).

```

24 CALL MPI_COMM_SIZE(comm, size, ierr)
25 CALL MPI_COMM_RANK(comm, rank, ierr)
26 IF (rank .GT. 0) THEN          ! client code
27   DO WHILE(.TRUE.)
28     CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
29     CALL MPI_WAIT(request, status, ierr)
30   END DO
31 ELSE                          ! rank=0 -- server code
32   DO i=1,size-1
33     CALL MPI_Irecv(a(1,i), n, MPI_REAL, i, tag, &
34                  comm, request_list(i), ierr)
35   END DO
36   DO WHILE(.TRUE.)
37     CALL MPI_WAITANY(size-1, request_list, index, status, ierr)
38     CALL DO_SERVICE(a(1,index)) ! handle one message
39     CALL MPI_Irecv(a(1, index), n, MPI_REAL, index, tag, &
40                  comm, request_list(index), ierr)
41   END DO
42 END IF

```

43 **Example 3.17.** Same code, using MPI_WAIT SOME.

```

44 CALL MPI_COMM_SIZE(comm, size, ierr)
45 CALL MPI_COMM_RANK(comm, rank, ierr)
46 IF (rank .GT. 0) THEN          ! client code
47   DO WHILE(.TRUE.)
48

```

```

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_WAIT_SOME(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)

Fortran 2008 binding

```

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

```

Fortran binding

```

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

```

Sets flag = true if the operation is *complete*, and, if so, returns in status the request status. However, unlike test or wait, it does not deallocate or *inactivate* the request; a

subsequent call to test, wait or free should be executed with that request. It sets `flag = false` if the operation is not *complete*.

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

The *progress* rule for `MPI_TEST`, as described in Section 3.7.4, also applies to `MPI_REQUEST_GET_STATUS`.

3.8 Probe and Cancel

The `MPI_PROBE`, `MPI_IPROBE`, `MPI_MPROBE`, and `MPI_IMPROBE` procedures 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` procedure 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. *Cancelling* a send-recv request by calling `MPI_CANCEL` is not allowed.

3.8.1 Probe

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

IN	source	rank of source or <code>MPI_ANY_SOURCE</code> (integer)
IN	tag	message tag or <code>MPI_ANY_TAG</code> (integer)
IN	comm	communicator (handle)
OUT	flag	true if there is a matching message that can be received (logical)
OUT	status	status object (status)

C binding

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

Fortran 2008 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
```

Fortran binding

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


LOGICAL FLAG

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

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

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

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

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

A probe with `MPI_PROC_NULL` as source returns `flag = true`, and the status object returns `source = MPI_PROC_NULL`, `tag = MPI_ANY_TAG`, and `count = 0`; see Section 3.10.

MPI_PROBE(source, tag, comm, status)

IN	<code>source</code>	rank of source or <code>MPI_ANY_SOURCE</code> (integer)
IN	<code>tag</code>	message tag or <code>MPI_ANY_TAG</code> (integer)
IN	<code>comm</code>	communicator (handle)
OUT	<code>status</code>	status object (status)

C binding

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

Fortran 2008 binding

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

Fortran binding

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

1 MPI_PROBE behaves like MPI_IPROBE except that it is a *nonlocal* call that returns
2 only after a matching message has been found.

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

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

12 **Example 3.18.** Use probe to wait for an incoming message.

```
14 CALL MPI_COMM_RANK(comm, rank, ierr)
15 IF (rank .EQ. 0) THEN
16     CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
17 ELSE IF (rank .EQ. 1) THEN
18     CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
19 ELSE IF (rank .EQ. 2) THEN
20     DO i=1,2
21         CALL MPI_PROBE(MPI_ANY_SOURCE, 0, comm, status, ierr)
22         IF (status(MPI_SOURCE) .EQ. 0) THEN
23             100 CALL MPI_RECV(i, 1, MPI_INTEGER, 0, 0, comm, status, ierr)
24             ELSE
25             200 CALL MPI_RECV(x, 1, MPI_REAL, 1, 0, comm, status, ierr)
26         END IF
27     END DO
28 END IF
```

29 Each message is received with the right type.

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

```
31 ! ----- THIS EXAMPLE IS ERRONEOUS -----
32 CALL MPI_COMM_RANK(comm, rank, ierr)
33 IF (rank .EQ. 0) THEN
34     CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
35 ELSE IF (rank .EQ. 1) THEN
36     CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
37 ELSE IF (rank .EQ. 2) THEN
38     DO i=1,2
39         CALL MPI_PROBE(MPI_ANY_SOURCE, 0, comm, status, ierr)
40         IF (status(MPI_SOURCE) .EQ. 0) THEN
41             100 CALL MPI_RECV(i, 1, MPI_INTEGER, MPI_ANY_SOURCE, &
42                 0, comm, status, ierr)
43             ELSE
44             200 CALL MPI_RECV(x, 1, MPI_REAL, MPI_ANY_SOURCE, &
45                 0, comm, status, ierr)
46         END IF
47     END DO
48 END IF
```

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

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

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

3.8.2 Matching Probe

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

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

`MPI_improbe(source, tag, comm, flag, message, status)`

IN	source	rank of source or <code>MPI_ANY_SOURCE</code> (integer)
IN	tag	message tag or <code>MPI_ANY_TAG</code> (integer)
IN	comm	communicator (handle)
OUT	flag	true if there is a matching message that can be received (logical)

```

1      OUT      message      returned message (handle)
2
3      OUT      status      status object (status)

```

C binding

```

5      int MPI_Improbe(int source, int tag, MPI_Comm comm, int *flag,
6                      MPI_Message *message, MPI_Status *status)
7

```

Fortran 2008 binding

```

9      MPI_Improbe(source, tag, comm, flag, message, status, ierror)
10     INTEGER, INTENT(IN) :: source, tag
11     TYPE(MPI_Comm), INTENT(IN) :: comm
12     LOGICAL, INTENT(OUT) :: flag
13     TYPE(MPI_Message), INTENT(OUT) :: message
14     TYPE(MPI_Status) :: status
15     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

17     MPI_IMPROBE(SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS, IERROR)
18     INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR
19     LOGICAL FLAG
20

```

MPI_IMPROBE returns `flag = true` if there is a message that can be received and that matches the pattern specified by the arguments `source`, `tag`, and `comm`. The call matches the same message that would have been received by a call to `MPI_RECV` with the same argument values for `source`, `tag`, `comm`, and `status` executed at the same point in the program and returns in `status` the same value that would have been returned by `MPI_RECV`. In addition, it returns in `message` a **message handle** to the matched message. Otherwise, the call returns `flag = false`, and leaves `status` and `message` undefined.

MPI_IMPROBE is a *local* procedure. According to the definitions in Section 2.4.2 and in contrast to `MPI_IPROBE`, it is a *nonblocking* procedure because it is the *initialization* of a *matched receive* operation.

A *matched receive* (`MPI_MRECV` or `MPI_IMRECV`) executed with the *message handle* will receive the message that was matched by the *matching probe*. Unlike `MPI_IPROBE`, no other probe or receive operation may match the message returned by `MPI_IMPROBE`. Each *message handle* 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 mode send* operation that is matched with `MPI_IMPROBE` or `MPI_MPROBE` will *complete* successfully only if both a *matching receive* is posted with `MPI_MRECV` or `MPI_IMRECV`, and the *matching receive* operation has *started* to receive the message sent by the *synchronous mode send*.

There is a special **predefined message handle**: `MPI_MESSAGE_NO_PROC`, which is a message that 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 `source = MPI_PROC_NULL` returns `flag = true`, `message =`

MPI_MESSAGE_NO_PROC, and the status object returns `source = MPI_PROC_NULL`, `tag = MPI_ANY_TAG`, and `count = 0`; see Section 3.10. It is not necessary to call `MPI_MRECV` or `MPI_IMRECV` with `MPI_MESSAGE_NO_PROC`, but it is not *erroneous* to do so.

Rationale. `MPI_MESSAGE_NO_PROC` 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 <code>MPI_ANY_SOURCE</code> (integer)
IN	tag	message tag or <code>MPI_ANY_TAG</code> (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)
```

Fortran 2008 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
```

Fortran 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`.

According to the definitions in Section 2.4.2, `MPI_MPROBE` is *incomplete*. It is also a *nonlocal* procedure.

Advice to users. This is one of the exceptions in which *incomplete* procedures are *nonlocal*. (*End of advice to users.*)

3.8.3 Matched Receives

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

```

1 MPI_MRECV(buf, count, datatype, message, status)
2   OUT    buf                initial address of receive buffer (choice)
3
4   IN     count              number of elements in receive buffer (non-negative
5                               integer)
6
7   IN     datatype           datatype of each receive buffer element (handle)
8
9   INOUT  message            message (handle)
10
11  OUT    status              status object (status)

```

C binding

```

12 int MPI_Mrecv(void *buf, int count, MPI_Datatype datatype,
13              MPI_Message *message, MPI_Status *status)
14
15 int MPI_Mrecv_c(void *buf, MPI_Count count, MPI_Datatype datatype,
16                MPI_Message *message, MPI_Status *status)

```

Fortran 2008 binding

```

17 MPI_Mrecv(buf, count, datatype, message, status, ierror)
18   TYPE(*), DIMENSION(..) :: buf
19   INTEGER, INTENT(IN) :: count
20   TYPE(MPI_Datatype), INTENT(IN) :: datatype
21   TYPE(MPI_Message), INTENT(INOUT) :: message
22   TYPE(MPI_Status) :: status
23   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
25 MPI_Mrecv(buf, count, datatype, message, status, ierror) !(_c)
26   TYPE(*), DIMENSION(..) :: buf
27   INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
28   TYPE(MPI_Datatype), INTENT(IN) :: datatype
29   TYPE(MPI_Message), INTENT(INOUT) :: message
30   TYPE(MPI_Status) :: status
31   INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

32 MPI_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR)
33   <type> BUF(*)
34   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. This is consistent with the status object produced by a call to MPI_RECV or to MPI_PROBE with source = MPI_PROC_NULL (see Section 3.10). A call to MPI_MRECV with MPI_MESSAGE_NULL is *erroneous*.

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

OUT	buf	initial address of receive buffer (choice)
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)
```

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

Fortran 2008 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
```

```
MPI_Imrecv(buf, count, datatype, message, request, ierror) !(_c)
```

```
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
INTEGER(KIND=MPI_COUNT_KIND), 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
```

Fortran 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.

1 If `MPI_IMRECV` is called with `MPI_MESSAGE_NO_PROC` as the message argument, the
 2 call returns immediately with a request object that, when completed, will yield a status
 3 object set to `source = MPI_PROC_NULL`, `tag = MPI_ANY_TAG`, and `count = 0`, as if a receive
 4 from `MPI_PROC_NULL` was issued (see Section 3.10). A call to `MPI_IMRECV` with
 5 `MPI_MESSAGE_NULL` is *erroneous*.

6
 7 *Advice to implementors.* If reception of a matched message is started with
 8 `MPI_IMRECV`, then it is possible to *cancel* the returned request with `MPI_CANCEL`. If
 9 `MPI_CANCEL` succeeds, the matched message must be found by a subsequent message
 10 probe (`MPI_PROBE`, `MPI_IPROBE`, `MPI_MPROBE`, or `MPI_IMPROBE`), received by a
 11 subsequent receive operation or *cancelled* by the sender. See Section 3.8.4 for details
 12 about `MPI_CANCEL`. The *cancellation* of operations initiated with `MPI_IMRECV` may
 13 fail. (*End of advice to implementors.*)

14 3.8.4 Cancel

15
 16
 17
 18 `MPI_CANCEL(request)`

19 IN request communication request (handle)

21 C binding

22 `int MPI_Cancel(MPI_Request *request)`

23 Fortran 2008 binding

24 `MPI_Cancel(request, ierror)`

25 TYPE(MPI_Request), INTENT(IN) :: request

26 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

27 Fortran binding

28 `MPI_CANCEL(REQUEST, IERROR)`

29 INTEGER REQUEST, IERROR

30
 31
 32 A call to `MPI_CANCEL` marks for *cancellation* a pending, *nonblocking* communica-
 33 tion operation (send or receive). *Cancelling* a send request by calling `MPI_CANCEL` is
 34 deprecated. The *cancel* call is *local*. It returns *immediately*, possibly before the communi-
 35 cation is actually *cancelled*. It is still necessary to call `MPI_REQUEST_FREE`, `MPI_WAIT` or
 36 `MPI_TEST` (or any of the derived procedures) with the *cancelled* request as argument after
 37 the call to `MPI_CANCEL`. If a communication is marked for *cancellation*, then a `MPI_WAIT`
 38 call for that communication is guaranteed to return, irrespective of the activities of other
 39 processes (i.e., `MPI_WAIT` behaves as a *local* function); similarly if `MPI_TEST` is repeatedly
 40 called in a busy wait loop for a *cancelled* communication, then `MPI_TEST` will eventually
 41 be successful.

42
 43 `MPI_CANCEL` can be used to *cancel* a communication that uses a *persistent commu-*
 44 *nication request* (see Section 3.9), in the same way it is used for nonpersistent requests.
 45 *Cancelling* a persistent send request by calling `MPI_CANCEL` is deprecated. A successful
 46 *cancellation* *cancels* the *active* communication, but not the request itself. After the call to
 47 `MPI_CANCEL` and the subsequent call to `MPI_WAIT` or `MPI_TEST`, the request becomes
 48 *inactive* and can be activated for a new communication.

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

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

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

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

`MPI_TEST_CANCELLED(status, flag)`

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

C binding

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

Fortran 2008 binding

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

Fortran binding

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

Returns `flag = true` if the communication associated with the status object was *cancelled* 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*

1 of this send may require communication with the intended receiver in order to free
 2 allocated buffers. On some systems this may require an interrupt to the intended
 3 receiver. Note that, while communication may be needed to implement
 4 `MPI_CANCEL`, this is still a *local* procedure, since its completion does not depend on
 5 the code executed by other processes. If processing is required on another process,
 6 this should be transparent to the application (hence the need for an interrupt and an
 7 interrupt handler). (*End of advice to implementors.*)
 8

9 3.9 Persistent Communication Requests

11 Often a communication with the same argument list (with the exception of the buffer con-
 12 tents) is repeatedly executed within the inner loop of a parallel computation. In such a
 13 situation, it may be possible to optimize the communication by binding the list of commu-
 14 nication arguments to a *persistent communication request* once and then repeatedly using
 15 the request to *start* and *complete* operations. In the case of point-to-point communication,
 16 the *persistent communication request* thus created can be thought of as a communication
 17 port or a “half-channel.” It does not provide the full functionality of a conventional channel,
 18 since there is no binding of the send port to the receive port. This construct allows reduction
 19 of the overhead for communication between the process and communication controller, but
 20 not of the overhead for communication between one communication controller and another.
 21 It is not necessary that messages sent with a persistent point-to-point request be received
 22 by a receive operation using a persistent point-to-point request, or vice versa.

23 There are also persistent collective communication operations defined in Section 6.13
 24 and Section 8.8. The remainder of this section covers the point-to-point persistent *ini-*
 25 *tialization* operations and the start routines, which are used for persistent point-to-point,
 26 partitioned point-to-point, and persistent collective communication operations.
 27

28 A point-to-point **persistent communication request** is created using one of the five
 29 following calls. These point-to-point persistent *initialization* calls involve no communica-
 30 tion.
 31

32 `MPI_SEND_INIT(buf, count, datatype, dest, tag, comm, request)`

33	IN	<code>buf</code>	initial address of send buffer (choice)
34	IN	<code>count</code>	number of elements sent (non-negative integer)
35	IN	<code>datatype</code>	type of each element (handle)
36	IN	<code>dest</code>	rank of destination (integer)
37	IN	<code>tag</code>	message tag (integer)
38	IN	<code>comm</code>	communicator (handle)
39	OUT	<code>request</code>	communication request (handle)

43 C binding

44 `int MPI_Send_init(const void *buf, int count, MPI_Datatype datatype, int dest,`
 45 `int tag, MPI_Comm comm, MPI_Request *request)`
 46

47 `int MPI_Send_init_c(const void *buf, MPI_Count count, MPI_Datatype datatype,`
 48 `int dest, int tag, MPI_Comm comm, MPI_Request *request)`

Fortran 2008 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

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

```

Fortran 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.

```

```

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 (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)

```

```

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

```

Fortran 2008 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

```

```

1     TYPE(MPI_Request), INTENT(OUT) :: request
2     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
4 MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror) !(_c)
5     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
6     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
7     TYPE(MPI_Datatype), INTENT(IN) :: datatype
8     INTEGER, INTENT(IN) :: dest, tag
9     TYPE(MPI_Comm), INTENT(IN) :: comm
10    TYPE(MPI_Request), INTENT(OUT) :: request
11    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

12 MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
13     <type> BUF(*)
14     INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
15
16     Creates a persistent communication request for a buffered mode send operation.
17
18
19

```

```

20 MPI_SSEND_INIT(buf, count, datatype, dest, tag, comm, request)
21     IN      buf          initial address of send buffer (choice)
22     IN      count       number of elements sent (non-negative integer)
23     IN      datatype    type of each element (handle)
24     IN      dest        rank of destination (integer)
25     IN      tag         message tag (integer)
26     IN      comm        communicator (handle)
27     IN      request     communication request (handle)
28
29
30

```

C binding

```

31 int MPI_Ssend_init(const void *buf, int count, MPI_Datatype datatype, int dest,
32                  int tag, MPI_Comm comm, MPI_Request *request)
33
34 int MPI_Ssend_init_c(const void *buf, MPI_Count count, MPI_Datatype datatype,
35                    int dest, int tag, MPI_Comm comm, MPI_Request *request)
36

```

Fortran 2008 binding

```

37 MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror)
38     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
39     INTEGER, INTENT(IN) :: count, dest, tag
40     TYPE(MPI_Datatype), INTENT(IN) :: datatype
41     TYPE(MPI_Comm), INTENT(IN) :: comm
42     TYPE(MPI_Request), INTENT(OUT) :: request
43     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45 MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror) !(_c)
46     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
47     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
48

```

```

TYPE(MPI_Datatype), INTENT(IN) :: datatype
INTEGER, INTENT(IN) :: dest, tag
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran 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 request* for a *synchronous mode send* operation.

```

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

```

C binding

```

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

```

```

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

```

Fortran 2008 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

```

```

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

```

Fortran binding

```

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

```

Creates a *persistent communication request* for a *ready mode send* operation.

```

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

```

9	OUT	buf	initial address of receive buffer (choice)
10			
11	IN	count	number of elements received (non-negative integer)
12			
13	IN	datatype	type of each element (handle)
14			
15	IN	source	rank of source or MPI_ANY_SOURCE (integer)
16			
17	IN	tag	message tag or MPI_ANY_TAG (integer)
18			
19	IN	comm	communicator (handle)
20			
21	OUT	request	communication request (handle)

C binding

```

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

```

```

24 int MPI_Recv_init_c(void *buf, MPI_Count count, MPI_Datatype datatype,
25                    int source, int tag, MPI_Comm comm, MPI_Request *request)

```

Fortran 2008 binding

```

26 MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror)
27   TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
28   INTEGER, INTENT(IN) :: count, source, tag
29   TYPE(MPI_Datatype), INTENT(IN) :: datatype
30   TYPE(MPI_Comm), INTENT(IN) :: comm
31   TYPE(MPI_Request), INTENT(OUT) :: request
32   INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

33 MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror) !(_c)
34   TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
35   INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
36   TYPE(MPI_Datatype), INTENT(IN) :: datatype
37   INTEGER, INTENT(IN) :: source, tag
38   TYPE(MPI_Comm), INTENT(IN) :: comm
39   TYPE(MPI_Request), INTENT(OUT) :: request
40   INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

41 MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
42   <type> BUF(*)
43   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 that uses a *persistent communication request* is *started* by the function `MPI_START`.

`MPI_START(request)`

INOUT request communication request (handle)

C binding

`int MPI_Start(MPI_Request *request)`

Fortran 2008 binding

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

Fortran binding

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

The argument, `request`, is a handle returned by any of the *initialization* procedures for persistent point-to-point communication (the previous five procedures), or for partitioned point-to-point communication (see Section 4), or for persistent collective communication (see Sections 6.13 and 8.8). The associated request should be *inactive*. The request becomes *active* once the call is made.

If the request is for a *ready mode send* operation, then a matching receive operation 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 (non-negative integer)
 INOUT array_of_requests array of requests (array of handles)

C binding

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

Fortran 2008 binding

`MPI_Startall(count, array_of_requests, ierror)`
 INTEGER, INTENT(IN) :: count

```

1     TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
2     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

4 MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)
5     INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR
6

```

The execution of `MPI_STARTALL` has the same effect as the execution of `MPI_START` for each of the array elements in some arbitrary order. `MPI_STARTALL` with an array of length one is equivalent to `MPI_START`.

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

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

Create (Start Complete)* Free

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

A send operation *started* with `MPI_START` can be matched with any receive operation and, likewise, a receive operation *started* with `MPI_START` can receive messages generated by any send operation.

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

3.10 Null Processes

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

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

possible, and the status object returns `source = MPI_PROC_NULL`, `tag = MPI_ANY_TAG` and `count = 0`. A matching probe (cf. Section 3.8.2) with `source = MPI_PROC_NULL` returns `flag = true`, `message = MPI_MESSAGE_NO_PROC`, and the status object returns `source = MPI_PROC_NULL`, `tag = MPI_ANY_TAG`, and `count = 0`.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

DRAFT

DRAFT

Chapter 4

Partitioned Point-to-Point Communication

4.1 Introduction

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

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

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

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

4.2 Semantics of Partitioned Point-to-Point Communication

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

Persistent communications use opaque `MPI_REQUEST` objects as described in Section 3. Partitioned communication uses these same semantics for `MPI_REQUEST` objects.

Partitioned communication provides fine-grained transfers on either or both sides of a send-receive operation described by requests. Persistent communication semantics are ideal for partitioned communication: they provide `MPI_PSEND_INIT` and `MPI_PRECV_INIT` functions that allow partitioned communication setup to occur prior to message transfers. Partitioned communication initialization functions are local. The partitioned communication initialization includes inputs on the number of user-visible partitions on the send-side and receive-side, which may differ. Valid partitioned communication operations must have one or more partitions specified.

Once an `MPI_PSEND_INIT` call has been made, the user may start the operation with a call to a starting procedure and complete the operation with a number of `MPI_PREADY` calls equal to the requested number of send partitions followed by a call to a completing procedure. A call to `MPI_PREADY` notifies the MPI library that a specified portion of the data buffer (a specific partition) is ready to be sent. Notification of partial completion can be done via fine-grained `MPI_PARRIVED` calls at the receiver before a final `MPI_TEST/MPI_WAIT` on the request itself; the latter represents overall operation completion upon success. A full set of methods for starting and completing partitioned communication is given in the following sections.

Advice to users. Having a large number of receiver-side partitions can increase overheads as the completion mechanism may need to work with finer-grained notifications. Using a small number of receiver-side partitions *may* provide higher performance.

A large number of sender-side partitions may be aggregated by an MPI implementation, making performance concerns of a large number of sender-side partitions potentially less impactful than receiver-side granularity. (*End of advice to users.*)

Advice to implementors. It is expected that an MPI implementation will attempt to balance latency and aggregation for data transfers for the requested partition counts on the sender-side and receiver-side to allow optimization for different hardware. A high quality implementation may perform significant optimizations to enhance performance in this way; they may, for example, resize the data transfers of the partitions to combine partitions in fractional partition sizes (e.g., 2.5 partitions in a single data transfer). (*End of advice to implementors.*)

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

Example 4.1. Simple partitioned communication example.

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

```

1  double message[PARTITIONS*COUNT];
2  MPI_Count partitions = PARTITIONS;
3  int source = 0, dest = 1, tag = 1, flag = 0;
4  int myrank, i;
5  int provided;
6  MPI_Request request;
7  MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZED, &provided);
8  if (provided < MPI_THREAD_SERIALIZED)
9      MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
10 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
11 if (myrank == 0)
12 {
13     MPI_Psend_init(message, partitions, COUNT, MPI_DOUBLE, dest, tag,
14                   MPI_COMM_WORLD, MPI_INFO_NULL, &request);
15     MPI_Start(&request);
16     for(i = 0; i < partitions; ++i)
17     {
18         /* compute and fill partition #i, then mark ready: */
19         MPI_Pready(i, request);
20     }
21     while(!flag)
22     {
23         /* do useful work #1 */
24         MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
25         /* do useful work #2 */
26     }
27     MPI_Request_free(&request);
28 }
29 else if (myrank == 1)
30 {
31     MPI_Precv_init(message, partitions, COUNT, MPI_DOUBLE, source, tag,
32                   MPI_COMM_WORLD, MPI_INFO_NULL, &request);
33     MPI_Start(&request);
34     while(!flag)
35     {
36         /* do useful work #1 */
37         MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
38         /* do useful work #2 */
39     }
40     MPI_Request_free(&request);
41 }
42 MPI_Finalize();
43 return 0;
44 }

```

Rationale. Partitioned communication is designed to provide opportunities for MPI implementations to optimize data transfers. MPI is free to choose how many transfers to do within a partitioned communication send independent of how many partitions are reported as ready to MPI through MPI_PREADY calls. Aggregation of partitions is permitted but not required. Ordering of partitions is permitted but not required. A naive implementation can simply wait for the entire message buffer to be marked ready before any transfer(s) occur and could wait until the completion function is

1 called on a request before transferring data. However, this modality of communication
 2 gives MPI implementations far more flexibility in data movement than nonpartitioned
 3 communications. (*End of rationale.*)
 4

5 4.2.1 Communication Initialization and Starting with Partitioning

6 Initialization of partitioned communication operations use the initialization calls described
 7 below. Subsequent to initialization, MPI_START/MPI_STARTALL are used as the first
 8 indication to MPI that a message transfer will occur. For send-side operations, neither
 9 initializing nor starting the operation enables transfer of any part of the user buffer. Freeing
 10 or canceling a partitioned communication request that is active (i.e., initialized and started)
 11 and not completed is erroneous. After the partitioned communication operation is started,
 12 individual partitions of a message are indicated as ready to be sent by MPI via the
 13 MPI_PREADY function, described below.
 14

15
 16 MPI_PSEND_INIT(buf, partitions, count, datatype, dest, tag, comm, info, request)

17	IN	buf	initial address of send buffer (choice)
18	IN	partitions	number of partitions (non-negative integer)
19	IN	count	number of elements sent per partition (non-negative integer)
20	IN	datatype	type of each element (handle)
21	IN	dest	rank of destination (integer)
22	IN	tag	message tag (integer)
23	IN	comm	communicator (handle)
24	IN	info	info argument (handle)
25	OUT	request	communication request (handle)

31 C binding

32
 33 int MPI_Psend_init(const void *buf, int partitions, MPI_Count count,
 34 MPI_Datatype datatype, int dest, int tag, MPI_Comm comm,
 35 MPI_Info info, MPI_Request *request)

36 Fortran 2008 binding

37 MPI_Psend_init(buf, partitions, count, datatype, dest, tag, comm, info,
 38 request, ierror)
 39 TYPE(*), DIMENSION(..), INTENT(IN) :: buf
 40 INTEGER, INTENT(IN) :: partitions, dest, tag
 41 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
 42 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 43 TYPE(MPI_Comm), INTENT(IN) :: comm
 44 TYPE(MPI_Info), INTENT(IN) :: info
 45 TYPE(MPI_Request), INTENT(OUT) :: request
 46 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 47
 48

Fortran binding

```

MPI_PSEND_INIT(BUF, PARTITIONS, COUNT, DATATYPE, DEST, TAG, COMM, INFO,
               REQUEST, IERROR)
<type> BUF(*)
INTEGER PARTITIONS, DATATYPE, DEST, TAG, COMM, INFO, REQUEST, IERROR
INTEGER(KIND=MPI_COUNT_KIND) COUNT

```

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

Advice to implementors. Unlike MPI_SEND_INIT, MPI_PSEND_INIT can be matched as early as the initialization call. Also, unlike MPI_SEND_INIT, MPI_PSEND_INIT takes an info argument. (*End of advice to implementors.*)

```

MPI_PRECV_INIT(buf, partitions, count, datatype, source, tag, comm, info, request)

```

IN	buf	initial address of recv buffer (choice)
IN	partitions	number of partitions (non-negative integer)
IN	count	number of elements received per partition (non-negative integer)
IN	datatype	type of each element (handle)
IN	source	rank of source (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)

C binding

```

int MPI_Precv_init(void *buf, int partitions, MPI_Count count,
                  MPI_Datatype datatype, int source, int tag, MPI_Comm comm,
                  MPI_Info info, MPI_Request *request)

```

Fortran 2008 binding

```

MPI_Precv_init(buf, partitions, count, datatype, source, tag, comm, info,
               request, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: buf
INTEGER, INTENT(IN) :: partitions, source, tag

```

```

1     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
2     TYPE(MPI_Datatype), INTENT(IN) :: datatype
3     TYPE(MPI_Comm), INTENT(IN) :: comm
4     TYPE(MPI_Info), INTENT(IN) :: info
5     TYPE(MPI_Request), INTENT(OUT) :: request
6     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

8     MPI_PRECV_INIT(BUF, PARTITIONS, COUNT, DATATYPE, SOURCE, TAG, COMM, INFO,
9                   REQUEST, IERROR)
10
11    <type> BUF(*)
12    INTEGER PARTITIONS, DATATYPE, SOURCE, TAG, COMM, INFO, REQUEST, IERROR
13    INTEGER(KIND=MPI_COUNT_KIND) COUNT

```

15 *Rationale.* The info argument is provided in order to support per-operation implementation-defined info keys. (*End of rationale.*)

18 MPI_PRECV_INIT creates a partitioned communication receive request and binds to it all the arguments of a partitioned receive operation. This operation can only match with partitioned communication initialization operations, therefore the MPI library is required to match MPI_PRECV_INIT calls only with a corresponding MPI_PSEND_INIT call. Matching follows the same MPI matching rules as for point-to-point communication (see Chapter 3) with communicator, tag, and source dictating message matching. In the event that the communicator, tag, and source do not uniquely identify a message, the order in which partitioned communication initialization calls are made is the order in which they will eventually match. Partitioned communication initialization calls are local. That is, MPI_PRECV_INIT may return before the operation completes. It is erroneous to provide a partitions value ≤ 0 . Wildcards for source and tag are not allowed.

30 *Advice to implementors.* Unlike MPI_RECV_INIT, MPI_PRECV_INIT may communicate. Also unlike MPI_RECV_INIT, MPI_PRECV_INIT takes an info argument. (*End of advice to implementors.*)

```

35 MPI_PREADY(partition, request)

```

36	IN	partition	partition to mark ready for transfer (non-negative integer)
37			
38			
39	INOUT	request	partitioned communication request (handle)
40			

C binding

```

42 int MPI_Pready(int partition, MPI_Request request)

```

Fortran 2008 binding

```

44 MPI_Pready(partition, request, ierror)
45     INTEGER, INTENT(IN) :: partition
46     TYPE(MPI_Request), INTENT(IN) :: request
47     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48

```


Fortran binding

```
MPI_PREADY(PARTITION, REQUEST, IERROR)
    INTEGER PARTITION, REQUEST, IERROR
```

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

```
MPI_PREADY_RANGE(partition_low, partition_high, request)
```

IN	partition_low	lowest partition ready for transfer (non-negative integer)
IN	partition_high	highest partition ready for transfer (non-negative integer)
INOUT	request	partitioned communication request (handle)

C binding

```
int MPI_Pready_range(int partition_low, int partition_high,
    MPI_Request request)
```

Fortran 2008 binding

```
MPI_Pready_range(partition_low, partition_high, request, ierror)
    INTEGER, INTENT(IN) :: partition_low, partition_high
    TYPE(MPI_Request), INTENT(IN) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_PREADY_RANGE(PARTITION_LOW, PARTITION_HIGH, REQUEST, IERROR)
    INTEGER PARTITION_LOW, PARTITION_HIGH, REQUEST, IERROR
```

A call to MPI_PREADY_RANGE has the same effect as calls to MPI_PREADY, executed for $i = \text{partition_low}, \dots, \text{partition_high}$, in some arbitrary order. Calls to MPI_PREADY_RANGE follow the same rules as those for MPI_PREADY calls.

```
MPI_PREADY_LIST(length, array_of_partitions, request)
```

IN	length	list length (integer)
IN	array_of_partitions	array of partitions (array of non-negative integers)
INOUT	request	partitioned communication request (handle)

C binding

```
int MPI_Pready_list(int length, const int array_of_partitions[],
```

1 MPI_Request request)

2

3 Fortran 2008 binding

4

MPI_Pready_list(length, array_of_partitions, request, ierror)
 5 INTEGER, INTENT(IN) :: length, array_of_partitions(length)
 6 TYPE(MPI_Request), INTENT(IN) :: request
 7 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

8

9 Fortran binding

10

MPI_PREADY_LIST(LENGTH, ARRAY_OF_PARTITIONS, REQUEST, IERROR)
 11 INTEGER LENGTH, ARRAY_OF_PARTITIONS(*), REQUEST, IERROR

12

A call to MPI_PREADY_LIST has the same effect as calls to
 13 MPI_PREADY, executed for the partitions specified in the range *array_of_partitions*[0]
 14 ..., *array_of_partitions*[*count* - 1] of the *array_of_partitions*, executed in some arbitrary
 15 order. Calls to MPI_PREADY_LIST follow the same rules as those for MPI_PREADY calls.

16

17

18 4.2.2 Communication Completion under Partitioning

19

The functions MPI_WAIT and MPI_TEST (and variants) are used to complete a partitioned
 20 communication operation. The completion of a partitioned send operation indicates that
 21 the sender is now free to call MPI_START/MPI_STARTALL to restart the operation and
 22 subsequently MPI_PREADY, MPI_PREADY_RANGE or MPI_PREADY_LIST. Alternatively,
 23 the user can safely free the partitioned communication request after the completion of the
 24 partitioned operation. For the sending process, completion of the partitioned send operation
 25 does not indicate that the partitions of the message have all been received.

26

The completion of a partitioned receive operation through MPI_WAIT or MPI_TEST
 27 indicates that the receive buffer contains all of the partitions. A function for probing the
 28 partial reception of the receive buffer is provided by MPI_PARRIVED. The MPI_PARRIVED
 29 function can be used to determine if the message data for the indicated partition has been
 30 received into the receive buffer. Upon success, the receiver becomes free to access the
 31 indicated partition (as well as any others that previously completed for that operation).

32

33

MPI_PARRIVED(request, partition, flag)

34

35	IN	request	partitioned communication request (handle)
36	IN	partition	partition to be tested (non-negative integer)
37	OUT	flag	true if operation completed on the specified partition, false if not (logical)

38

39

40

41 C binding

42

int MPI_Parrived(MPI_Request request, int partition, int *flag)

43

44 Fortran 2008 binding

45

MPI_Parrived(request, partition, flag, ierror)
 46 TYPE(MPI_Request), INTENT(IN) :: request
 47 INTEGER, INTENT(IN) :: partition
 48 LOGICAL, INTENT(OUT) :: flag
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

```

MPI_PARRIVED(REQUEST, PARTITION, FLAG, IERROR)
    INTEGER REQUEST, PARTITION, IERROR
    LOGICAL FLAG

```

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

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

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

4.2.3 Semantics of Communications in Partitioned Mode

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

Interpretation of count and datatype for partitioned communication. Partitioned communication uses the `count` and `datatype` arguments in the partitioned communication initialization functions to describe a single partition. The argument `partitions` specifies how many equal partitions of a number (`count`) of objects of `datatypes` make up the entire buffer to be transferred in the partitioned communication. As partitioned communication describes many partitions, using absolute displacements in `datatypes` (e.g., `MPI_BOTTOM`) is not supported. Partitions are contiguous in memory, there is no padding in between them. Once a partitioned send operation is started, each partition must be marked as ready using `MPI_PREADY` and the operation must be completed using a completion function, such as `MPI_TEST` or `MPI_WAIT`.

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

4.3 Partitioned Communication Examples

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

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

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

Example 4.2. Equal partitioning on send-side and receive-side using threads.

```

14 #include <stdlib.h>
15 #include "mpi.h"
16 #define NUM_THREADS 8
17 #define PARTITIONS 8
18 #define PARTLENGTH 16
19 int main(int argc, char *argv[]) /* same send/rcv partitioning */
20 {
21     double message[PARTITIONS*PARTLENGTH];
22     int partitions = PARTITIONS;
23     int partlength = PARTLENGTH;
24     int count = 1, source = 0, dest = 1, tag = 1, flag = 0;
25     int myrank;
26     int provided;
27     MPI_Request request;
28     MPI_Info info = MPI_INFO_NULL;
29     MPI_Datatype xfer_type;
30     MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
31     if (provided < MPI_THREAD_MULTIPLE)
32         MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
33     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
34     MPI_Type_contiguous(partlength, MPI_DOUBLE, &xfer_type);
35     MPI_Type_commit(&xfer_type);
36     if (myrank == 0)
37     {
38         MPI_Psend_init(message, partitions, count, xfer_type, dest, tag,
39                        MPI_COMM_WORLD, info, &request);
40         MPI_Start(&request);
41
42         #pragma omp parallel for shared(request) num_threads(NUM_THREADS)
43         for (int i=0; i<partitions; i++)
44         {
45             /* compute and fill partition #i, then mark ready: */
46             MPI_Pready(i, request);
47         }
48         while(!flag)
49         {
50             /* Do useful work */
51             MPI_Test(&request, &flag, MPI_STATUS_IGNORE);

```

```

    /* Do useful work */
  }
  MPI_Request_free(&request);
}
else if (myrank == 1)
{
  MPI_Precv_init(message, partitions, count, xfer_type, source, tag,
                MPI_COMM_WORLD, info, &request);
  MPI_Start(&request);
  while(!flag)
  {
    /* Do useful work */
    MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
    /* Do useful work */
  }
  MPI_Request_free(&request);
}
MPI_Finalize();
return 0;
}

```

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

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

Example 4.3. Parallel task producers for partitioned communication using threads.

```

#include <stdlib.h>
#include "mpi.h"
#define NUM_THREADS 8
#define NUM_TASKS 64
#define PARTITIONS NUM_TASKS
#define PARTLENGTH 16
#define MESSAGE_LENGTH PARTITIONS*PARTLENGTH
int main(int argc, char *argv[]) /* send-side partitioning */
{
  double message[MESSAGE_LENGTH];
  int send_partitions = PARTITIONS,
      send_partlength = PARTLENGTH,
      recv_partitions = 1,
      recv_partlength = PARTITIONS*PARTLENGTH;
  int count = 1, source = 0, dest = 1, tag = 1, flag = 0;
  int myrank;
  int provided;
  MPI_Request request;
  MPI_Info info = MPI_INFO_NULL;
  MPI_Datatype send_type;
  MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
  if (provided < MPI_THREAD_MULTIPLE)
    MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Type_contiguous(send_partlength, MPI_DOUBLE, &send_type);

```

```

1  MPI_Type_commit(&send_type);
2
3  if (myrank == 0)
4  {
5      MPI_Psend_init(message, send_partitions, count, send_type, dest, tag,
6                    MPI_COMM_WORLD, info, &request);
7      MPI_Start(&request);
8
9      #pragma omp parallel shared(request) num_threads(NUM_THREADS)
10     {
11         #pragma omp single
12         {
13             /* single thread creates 64 tasks to be executed by 8 threads */
14             for (int partition_num=0;partition_num<NUM_TASKS;partition_num++)
15             {
16                 #pragma omp task firstprivate(partition_num)
17                 {
18                     /* compute and fill partition #partition_num, then mark
19                     ready: */
20                     /* buffer is filled in arbitrary order from each task */
21                     MPI_Pready(partition_num, request);
22                 } /*end task*/
23             } /* end for */
24         } /* end single */
25     } /* end parallel */
26     while(!flag)
27     {
28         /* Do useful work */
29         MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
30         /* Do useful work */
31     }
32     MPI_Request_free(&request);
33 }
34 else if (myrank == 1)
35 {
36     MPI_Precv_init(message, recv_partitions, recv_partlength, MPI_DOUBLE,
37                  source, tag, MPI_COMM_WORLD, info, &request);
38
39     MPI_Start(&request);
40     while(!flag)
41     {
42         /* Do useful work */
43         MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
44         /* Do useful work */
45     }
46     MPI_Request_free(&request);
47 }
48 MPI_Finalize();
49 return 0;
50 }

```

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

This example demonstrates receive-side partial completion notification using more than one partition per receive-side thread. It uses a naive flag based method to test for multiple completed partitions per thread. Note that this means that some threads may be busy polling

for completion of assigned partitions when partitions are available to work on that were not assigned to the polling threads in this example. More advanced work stealing methods could be employed for greater efficiency. Like previous examples, it also demonstrates send-side production of input to part of an overall buffer. This example also uses different send-side and receive-side partitioning.

Example 4.4. Partitioned communication receive-side partial completion.

```

1  #include <stdlib.h>
2  #include "mpi.h"
3  #define NUM_THREADS 64
4  #define PARTITIONS NUM_THREADS
5  #define PARTLENGTH 16
6  #define MESSAGE_LENGTH PARTITIONS*PARTLENGTH
7  int main(int argc, char *argv[]) /* send-side partitioning */
8  {
9      double message[MESSAGE_LENGTH];
10     int send_partitions = PARTITIONS,
11         send_partlength = PARTLENGTH,
12         recv_partitions = PARTITIONS*2,
13         recv_partlength = PARTLENGTH/2;
14     int source = 0, dest = 1, tag = 1, flag = 0;
15     int myrank;
16     int provided;
17     MPI_Request request;
18     MPI_Info info = MPI_INFO_NULL;
19     MPI_Datatype send_type;
20     MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
21     if (provided < MPI_THREAD_MULTIPLE)
22         MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
23     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
24     MPI_Type_contiguous(send_partlength, MPI_DOUBLE, &send_type);
25     MPI_Type_commit(&send_type);
26
27     if (myrank == 0)
28     {
29         MPI_Psend_init(message, send_partitions, 1, send_type, dest, tag,
30             MPI_COMM_WORLD, info, &request);
31         MPI_Start(&request);
32         #pragma omp parallel for shared(request) \
33             firstprivate(send_partitions) \
34             num_threads(NUM_THREADS)
35         for (int i=0; i<send_partitions; i++)
36         {
37             /* compute and fill partition #i, then mark ready: */
38             MPI_Pready(i, request);
39         }
40         while(!flag)
41         {
42             /* Do useful work */
43             MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
44             /* Do useful work */
45         }
46     }
47
48

```

```
1
2     }
3     MPI_Request_free(&request);
4 }
5 else if (myrank == 1)
6 {
7     MPI_Precv_init(message, recv_partitions, recv_partlength,
8                   MPI_DOUBLE, source, tag, MPI_COMM_WORLD, info,
9                   &request);
10    MPI_Start(&request);
11    #pragma omp parallel for shared(request) \
12                          firstprivate(recv_partitions) \
13                          num_threads(NUM_THREADS)
14    for (int j=0; j<recv_partitions; j+=2)
15    {
16        int part_flag = 0;
17        int part1_complete = 0;
18        int part2_complete = 0;
19        while(part1_complete == 0 || part2_complete == 0)
20        {
21            /* test partition #j and #j+1 */
22            MPI_Parrived(request, j, &part_flag);
23            if(part_flag && part1_complete == 0)
24            {
25                part1_complete++;
26                /* Do work using partition j data */
27            }
28            MPI_Parrived(request, j+1, &part_flag);
29            if(part_flag && part2_complete == 0)
30            {
31                part2_complete++;
32                /* Do work using partition j+1 */
33            }
34        }
35    }
36    while(!flag)
37    {
38        /* Do useful work */
39        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
40        /* Do useful work */
41    }
42    MPI_Request_free(&request);
43 }
44 MPI_Finalize();
45 return 0;
46 }
47
48
```


Chapter 5

Datatypes

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

5.1 Derived Datatypes

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

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

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

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

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

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

map. The sequence of basic datatypes (displacements ignored) is the **type signature** of the datatype.

Let

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

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

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

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 $buf + disp_i$ 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 nonnegative integers. If the value is zero, no elements are generated in the type map and there is no effect on datatype bounds or extent.

We can use a handle to a general datatype as an argument in a send or receive operation, instead of a basic datatype argument. The operation `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 5.1.11, 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 $\{(int, 0)\}$, 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

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

then

$$\begin{aligned} lb(Typemap) &= \min_j disp_j, \\ ub(Typemap) &= \max_j (disp_j + \text{sizeof}(type_j)) + \epsilon, \text{ and} \\ extent(Typemap) &= ub(Typemap) - lb(Typemap). \end{aligned} \tag{5.1}$$

If $type_j$ requires alignment to a byte address that is a multiple of k_j , then ϵ is the least nonnegative 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 5.1 Section 5.1.

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

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

5.1.1 Type Constructors with Explicit Addresses

In Fortran, the functions MPI_TYPE_CREATE_HVECTOR, MPI_TYPE_CREATE_HINDEXED, MPI_TYPE_CREATE_HINDEXED_BLOCK, MPI_TYPE_CREATE_STRUCT, and MPI_GET_ADDRESS accept arguments of type INTEGER(KIND=MPI_ADDRESS_KIND), wherever arguments of type MPI_Aint are used in C. For Fortran compilers that do not support the Fortran 90 KIND notation, and where addresses are 64 bits whereas default INTEGERS are 32 bits, these arguments will be of type INTEGER*8 (assuming the Fortran compiler accepts the common extension of INTEGER*8 for eight-byte integers).

For the large count versions of three datatype constructors with explicit addresses, MPI_TYPE_CREATE_HINDEXED, MPI_TYPE_CREATE_HINDEXED_BLOCK, and MPI_TYPE_CREATE_STRUCT, absolute addresses shall not be used to specify byte displacements since the parameter is of type MPI_COUNT instead of type MPI_AINT.

5.1.2 Datatype Constructors

Contiguous. The simplest datatype constructor is MPI_TYPE_CONTIGUOUS, which allows replication of a datatype into contiguous locations.

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)

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

C binding

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

```

1 int MPI_Type_contiguous_c(MPI_Count count, MPI_Datatype oldtype,
2     MPI_Datatype *newtype)
3

```

Fortran 2008 binding

```

4 MPI_Type_contiguous(count, oldtype, newtype, ierror)
5     INTEGER, INTENT(IN) :: count
6     TYPE(MPI_Datatype), INTENT(IN) :: oldtype
7     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
8     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9

```

```

10 MPI_Type_contiguous(count, oldtype, newtype, ierror) !(_c)
11     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
12     TYPE(MPI_Datatype), INTENT(IN) :: oldtype
13     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
14     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15

```

Fortran binding

```

16 MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
17     INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
18

```

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

Example 5.2. Let `oldtype` have type map $\{(double, 0), (char, 8)\}$, with extent 16, and let `count` = 3. The type map of the datatype returned by `newtype` is

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

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

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

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

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

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

MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype)				1
IN	count	number of blocks (non-negative integer)		2
IN	blocklength	number of elements in each block (non-negative integer)		3
				4
IN	stride	number of elements between start of each block (integer)		5
				6
IN	oldtype	old datatype (handle)		7
				8
OUT	newtype	new datatype (handle)		9
				10

C binding

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

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

Fortran 2008 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
```

```
MPI_Type_vector(count, blocklength, stride, oldtype, newtype, ierror) !(_c)
  INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count, blocklength, stride
  TYPE(MPI_Datatype), INTENT(IN) :: oldtype
  TYPE(MPI_Datatype), INTENT(OUT) :: newtype
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

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

Example 5.3. Assume, again, that oldtype has type map $\{(double, 0), (char, 8)\}$, with extent 16. A call to MPI_TYPE_VECTOR(2, 3, 4, oldtype, newtype) will create the datatype with type map,

$$\{(double, 0), (char, 8), (double, 16), (char, 24), (double, 32), (char, 40), \\ (double, 64), (char, 72), (double, 80), (char, 88), (double, 96), (char, 104)\}.$$

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

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

In general, assume that `oldtype` has type map,

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

with extent ex . Let `bl` be the `blocklength`. The newly created datatype has a type map with `count · bl · n` entries:

$$\begin{aligned} &\{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1}), \\ &(type_0, disp_0 + ex), \dots, (type_{n-1}, disp_{n-1} + ex), \dots, \\ &(type_0, disp_0 + (bl - 1) \cdot ex), \dots, (type_{n-1}, disp_{n-1} + (bl - 1) \cdot ex), \\ &(type_0, disp_0 + stride \cdot ex), \dots, (type_{n-1}, disp_{n-1} + stride \cdot ex), \dots, \\ &(type_0, disp_0 + (stride + bl - 1) \cdot ex), \dots, (type_{n-1}, disp_{n-1} + (stride + bl - 1) \cdot ex), \dots, \\ &(type_0, disp_0 + stride \cdot (count - 1) \cdot ex), \dots, \\ &(type_{n-1}, disp_{n-1} + stride \cdot (count - 1) \cdot ex), \dots, \\ &(type_0, disp_0 + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots, \\ &(type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex)\}. \end{aligned}$$

A call to `MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)` is equivalent to a call to `MPI_TYPE_VECTOR(count, 1, 1, oldtype, newtype)`, or to a call to `MPI_TYPE_VECTOR(1, count, n, oldtype, newtype)`, where `n` is an arbitrary integer value.

Hvector. The function `MPI_TYPE_CREATE_HVECTOR` is identical to `MPI_TYPE_VECTOR`, except that `stride` is given in bytes, rather than in elements. The use for both types of vector constructors is illustrated in Section 5.1.14. (H stands for “heterogeneous”).

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

IN	count	number of blocks (non-negative integer)
IN	blocklength	number of elements in each block (non-negative integer)
IN	stride	number of 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)
```

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

Fortran 2008 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

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

```

Fortran 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,

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

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

$$\begin{aligned}
&\{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1}), \\
&(type_0, disp_0 + ex), \dots, (type_{n-1}, disp_{n-1} + ex), \dots, \\
&(type_0, disp_0 + (bl - 1) \cdot ex), \dots, (type_{n-1}, disp_{n-1} + (bl - 1) \cdot ex), \\
&(type_0, disp_0 + stride), \dots, (type_{n-1}, disp_{n-1} + stride), \dots, \\
&(type_0, disp_0 + stride + (bl - 1) \cdot ex), \dots, \\
&(type_{n-1}, disp_{n-1} + stride + (bl - 1) \cdot ex), \dots, \\
&(type_0, disp_0 + stride \cdot (count - 1)), \dots, (type_{n-1}, disp_{n-1} + stride \cdot (count - 1)), \dots, \\
&(type_0, disp_0 + stride \cdot (count - 1) + (bl - 1) \cdot ex), \dots, \\
&(type_{n-1}, disp_{n-1} + stride \cdot (count - 1) + (bl - 1) \cdot ex)\}.
\end{aligned}$$

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.

```

1 MPI_TYPE_INDEXED(count, array_of_blocklengths, array_of_displacements, oldtype,
2     newtype)
3
4 IN     count           number of blocks—also number of entries in
5     array_of_displacements and array_of_blocklengths
6     (non-negative integer)
7
8 IN     array_of_blocklengths  number of elements per block (array of non-negative
9     integers)
10
11 IN    array_of_displacements  displacement for each block, in multiples of oldtype
12     (array of integers)
13
14 IN    oldtype            old datatype (handle)
15
16 OUT   newtype           new datatype (handle)

```

C binding

```

17 int MPI_Type_indexed(int count, const int array_of_blocklengths[],
18     const int array_of_displacements[], MPI_Datatype oldtype,
19     MPI_Datatype *newtype)
20
21 int MPI_Type_indexed_c(MPI_Count count,
22     const MPI_Count array_of_blocklengths[],
23     const MPI_Count array_of_displacements[], MPI_Datatype oldtype,
24     MPI_Datatype *newtype)

```

Fortran 2008 binding

```

25 MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements, oldtype,
26     newtype, ierror)
27     INTEGER, INTENT(IN) :: count, array_of_blocklengths(count),
28     array_of_displacements(count)
29     TYPE(MPI_Datatype), INTENT(IN) :: oldtype
30     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
31     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33 MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements, oldtype,
34     newtype, ierror) !(_c)
35     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count,
36     array_of_blocklengths(count), array_of_displacements(count)
37     TYPE(MPI_Datatype), INTENT(IN) :: oldtype
38     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
39     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

40 MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS, OLDTYPE,
41     NEWTYPE, IERROR)
42     INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
43     OLDTYPE, NEWTYPE, IERROR

```

Example 5.5. Let `oldtype` have type map $\{(double, 0), (char, 8)\}$, with extent 16. Let $B = (3, 1)$ and let $D = (4, 0)$. A call to `MPI_TYPE_INDEXED(2, B, D, oldtype, newtype)` returns

a datatype with type map,

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

$$(\text{double}, 0), (\text{char}, 8)\}.$$

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

In general, assume that `oldtype` has type map,

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

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

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

$$(type_0, disp_0 + (D[0] + B[0] - 1) \cdot ex), \dots,$$

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

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

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

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

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

$$D[j] = j \cdot \text{stride}, \quad j = 0, \dots, \text{count} - 1,$$

and

$$B[j] = \text{blocklength}, \quad j = 0, \dots, \text{count} - 1.$$

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.

```

1 MPI_TYPE_CREATE_HINDEXED(count, array_of_blocklengths, array_of_displacements,
2     oldtype, newtype)
3
4 IN     count           number of blocks—also number of entries in
5     array_of_displacements and array_of_blocklengths
6     (non-negative integer)
7
8 IN     array_of_blocklengths  number of elements in each block (array of
9     non-negative integers)
10
11 IN     array_of_displacements  byte displacement of each block (array of integers)
12
13 IN     oldtype             old datatype (handle)
14
15 OUT    newtype            new datatype (handle)

```

C binding

```

16 int MPI_Type_create_hindexed(int count, const int array_of_blocklengths[],
17     const MPI_Aint array_of_displacements[], MPI_Datatype oldtype,
18     MPI_Datatype *newtype)
19
20 int MPI_Type_create_hindexed_c(MPI_Count count,
21     const MPI_Count array_of_blocklengths[],
22     const MPI_Count array_of_displacements[], MPI_Datatype oldtype,
23     MPI_Datatype *newtype)

```

Fortran 2008 binding

```

24 MPI_Type_create_hindexed(count, array_of_blocklengths, array_of_displacements,
25     oldtype, newtype, ierror)
26     INTEGER, INTENT(IN) :: count, array_of_blocklengths(count)
27     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: array_of_displacements(count)
28     TYPE(MPI_Datatype), INTENT(IN) :: oldtype
29     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
30     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32 MPI_Type_create_hindexed(count, array_of_blocklengths, array_of_displacements,
33     oldtype, newtype, ierror) !(_c)
34     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count,
35     array_of_blocklengths(count), array_of_displacements(count)
36     TYPE(MPI_Datatype), INTENT(IN) :: oldtype
37     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
38     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

39 MPI_TYPE_CREATE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
40     OLDTYPE, NEWTYPE, IERROR)
41
42     INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), OLDTYPE, NEWTYPE, IERROR
43     INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)

```

Assume that oldtype has type map,

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

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

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

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

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

IN	count	number of blocks—also number of entries in <code>array_of_displacements</code> (non-negative integer)
IN	blocklength	number of elements in each block (non-negative integer)
IN	array_of_displacements	array of displacements, in multiples of <code>oldtype</code> (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)
```

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

Fortran 2008 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
```

```

1     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
2     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

3 MPI_Type_create_indexed_block(count, blocklength, array_of_displacements,
4                             oldtype, newtype, ierror) !(_c)
5     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count, blocklength,
6     array_of_displacements(count)
7     TYPE(MPI_Datatype), INTENT(IN) :: oldtype
8     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
9     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10

```

11 Fortran binding

```

12 MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,
13                               OLDDTYPE, NEWTYPE, IERROR)
14     INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDDTYPE, NEWTYPE,
15     IERROR
16

```

17
18 **Hindexed_block.** The function `MPI_TYPE_CREATE_HINDEXED_BLOCK` is identical to
19 `MPI_TYPE_CREATE_INDEXED_BLOCK`, except that block displacements in
20 `array_of_displacements` are specified in bytes, rather than in multiples of the `oldtype` extent.

```

21
22 MPI_TYPE_CREATE_HINDEXED_BLOCK(count, blocklength, array_of_displacements,
23                               oldtype, newtype)
24
25     IN     count           number of blocks—also number of entries in
26                               array_of_displacements (non-negative integer)
27
28     IN     blocklength    number of elements in each block (non-negative
29                               integer)
30
31     IN     array_of_displacements  byte displacement of each block (array of integers)
32
33     IN     oldtype        old datatype (handle)
34
35     OUT    newtype        new datatype (handle)
36

```

34 C binding

```

35 int MPI_Type_create_hindexed_block(int count, int blocklength,
36                                   const MPI_Aint array_of_displacements[], MPI_Datatype oldtype,
37                                   MPI_Datatype *newtype)
38
39 int MPI_Type_create_hindexed_block_c(MPI_Count count, MPI_Count blocklength,
40                                     const MPI_Count array_of_displacements[], MPI_Datatype oldtype,
41                                     MPI_Datatype *newtype)
42

```

42 Fortran 2008 binding

```

43 MPI_Type_create_hindexed_block(count, blocklength, array_of_displacements,
44                               oldtype, newtype, ierror)
45     INTEGER, INTENT(IN) :: count, blocklength
46     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: array_of_displacements(count)
47     TYPE(MPI_Datatype), INTENT(IN) :: oldtype
48

```

```

TYPE(MPI_Datatype), INTENT(OUT) :: newtype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Type_create_hindexed_block(count, blocklength, array_of_displacements,
                               oldtype, newtype, ierror) !(_c)
INTEGER(KIND=MPI_COUNT_KIND), 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

```

Fortran 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 (non-negative 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	type 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)
int MPI_Type_create_struct_c(MPI_Count count,
                             const MPI_Count array_of_blocklengths[],
                             const MPI_Count array_of_displacements[],
                             const MPI_Datatype array_of_types[], MPI_Datatype *newtype)

```

Fortran 2008 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)

```

```

1     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: array_of_displacements(count)
2     TYPE(MPI_Datatype), INTENT(IN) :: array_of_types(count)
3     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
4     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
6 MPI_Type_create_struct(count, array_of_blocklengths, array_of_displacements,
7     array_of_types, newtype, ierror) !(_c)
8     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count,
9     array_of_blocklengths(count), array_of_displacements(count)
10    TYPE(MPI_Datatype), INTENT(IN) :: array_of_types(count)
11    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
12    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

14 MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
15     ARRAY_OF_TYPES, NEWTYPE, IERROR)
16     INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_TYPES(*), NEWTYPE, IERROR
17     INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
18

```

Example 5.6. Let `type1` have type map,

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

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

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

That is, two copies of `MPI_FLOAT` starting at 0, followed by one copy of `type1` starting at 16, followed by three copies of `MPI_CHAR`, starting at 26. In this example, 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,

$$\text{typemap}_i = \{(type_0^i, disp_0^i), \dots, (type_{n_i-1}^i, disp_{n_i-1}^i)\},$$

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

$$\begin{aligned} & \{(type_0^0, disp_0^0 + D[0]), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0]), \dots, \\ & (type_0^0, disp_0^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0]-1) \cdot ex_0), \dots, \\ & (type_0^{c-1}, disp_0^{c-1} + D[c-1]), \dots, (type_{n_{c-1}-1}^{c-1}, disp_{n_{c-1}-1}^{c-1} + D[c-1]), \dots, \\ & (type_0^{c-1}, disp_0^{c-1} + D[c-1] + (B[c-1] - 1) \cdot ex_{c-1}), \dots, \\ & (type_{n_{c-1}-1}^{c-1}, disp_{n_{c-1}-1}^{c-1} + D[c-1] + (B[c-1]-1) \cdot ex_{c-1})\}. \end{aligned}$$

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`.

5.1.3 Subarray Datatype Constructor

MPI_TYPE_CREATE_SUBARRAY(ndims, array_of_sizes, array_of_subsizes, array_of_starts,
order, oldtype, newtype)

IN	ndims	number of array dimensions (positive integer)	1
IN	array_of_sizes	number of elements of type oldtype in each dimension of the full array (array of positive integers)	2
IN	array_of_subsizes	number of elements of type oldtype in each dimension of the subarray (array of positive integers)	3
IN	array_of_starts	starting coordinates of the subarray in each dimension (array of non-negative integers)	4
IN	order	array storage order flag (state)	5
IN	oldtype	old datatype (handle)	6
OUT	newtype	new datatype (handle)	7

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)
```

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

Fortran 2008 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
```

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

Fortran binding

```
MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES, ARRAY_OF_SUBSIZES,
                          ARRAY_OF_STARTS, ORDER, OLDTYPE, NEWTYPE, IERROR)
```

1 INTEGER NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*), ARRAY_OF_STARTS(*),
2 ORDER, OLDTYPE, NEWTYPE, IERROR

3
4 The subarray type constructor creates an MPI datatype describing an n -dimensional
5 subarray of an n -dimensional array. The subarray may be situated anywhere within the
6 full array, and may be of any nonzero size up to the size of the larger array as long as it
7 is confined within this array. This type constructor facilitates creating filetypes to access
8 arrays distributed in blocks among processes to a single file that contains the global array,
9 see MPI I/O, especially Section 14.1.1.

10 This type constructor can handle arrays with an arbitrary number of dimensions and
11 works for both C and Fortran ordered matrices (i.e., row-major or column-major). Note
12 that a C program may use Fortran order and a Fortran program may use C order.

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

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

19 The `array_of_starts` contains the starting coordinates of each dimension of the subarray.
20 Arrays are assumed to be indexed starting from zero. For any dimension i , it is erroneous to
21 specify `array_of_starts[i] < 0` or `array_of_starts[i] > (array_of_sizes[i] - array_of_subsizes[i])`.

22 *Advice to users.* In a Fortran program with arrays indexed starting from 1, if the
23 starting coordinate of a particular dimension of the subarray is n , then the entry in
24 `array_of_starts` for that dimension is $n-1$. (*End of advice to users.*)

25
26 The `order` argument specifies the storage order for the subarray as well as the full array.
27 It must be set to one of the following:

28 MPI_ORDER_C The ordering used by C arrays, (i.e., row-major
29 order).
30 MPI_ORDER_FORTRAN The ordering used by Fortran arrays, (i.e., column-
31 major order).

32
33 A `ndims`-dimensional subarray (`newtype`) with no extra padding can be defined by the
34 function `Subarray()` as follows:

35 newtype = Subarray(`ndims`, {`size`₀, `size`₁, ..., `size`_{`ndims`-1}},
36 {`subsize`₀, `subsize`₁, ..., `subsize`_{`ndims`-1}},
37 {`start`₀, `start`₁, ..., `start`_{`ndims`-1}}, `oldtype`)

38
39 Let the typemap of `oldtype` have the form:

40
41 {(type₀, disp₀), (type₁, disp₁), ..., (type_{`n`-1}, disp_{`n`-1})}

42
43 where `typei` is a predefined MPI datatype, and let `ex` be the extent of `oldtype`. Then we define
44 the `Subarray()` function recursively using the following three equations. Equation 5.2 defines
45 the base step. Equation 5.3 defines the recursion step when `order = MPI_ORDER_FORTRAN`,
46 and Equation 5.4 defines the recursion step when `order = MPI_ORDER_C`. These equations
47 use the conceptual datatypes `lb_marker` and `ub_marker`; see Section 5.1.6 for details.

$$\begin{aligned}
& \text{Subarray}(1, \{size_0\}, \{subsize_0\}, \{start_0\}, \\
& \quad \{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\}) \\
& = \{(\text{lb_marker}, 0), \\
& \quad (type_0, disp_0 + start_0 \times ex), \dots, (type_{n-1}, disp_{n-1} + start_0 \times ex), \\
& \quad (type_0, disp_0 + (start_0 + 1) \times ex), \dots, (type_{n-1}, \\
& \quad \quad disp_{n-1} + (start_0 + 1) \times ex), \dots \\
& \quad (type_0, disp_0 + (start_0 + subsize_0 - 1) \times ex), \dots, \\
& \quad \quad (type_{n-1}, disp_{n-1} + (start_0 + subsize_0 - 1) \times ex), \\
& \quad (\text{ub_marker}, size_0 \times ex)\}
\end{aligned} \tag{5.2}$$

$$\begin{aligned}
& \text{Subarray}(ndims, \{size_0, size_1, \dots, size_{ndims-1}\}, \\
& \quad \{subsize_0, subsize_1, \dots, subsize_{ndims-1}\}, \\
& \quad \{start_0, start_1, \dots, start_{ndims-1}\}, \text{oldtype}) \\
& = \text{Subarray}(ndims - 1, \{size_1, size_2, \dots, size_{ndims-1}\}, \\
& \quad \{subsize_1, subsize_2, \dots, subsize_{ndims-1}\}, \\
& \quad \{start_1, start_2, \dots, start_{ndims-1}\}, \\
& \quad \text{Subarray}(1, \{size_0\}, \{subsize_0\}, \{start_0\}, \text{oldtype}))
\end{aligned} \tag{5.3}$$

$$\begin{aligned}
& \text{Subarray}(ndims, \{size_0, size_1, \dots, size_{ndims-1}\}, \\
& \quad \{subsize_0, subsize_1, \dots, subsize_{ndims-1}\}, \\
& \quad \{start_0, start_1, \dots, start_{ndims-1}\}, \text{oldtype}) \\
& = \text{Subarray}(ndims - 1, \{size_0, size_1, \dots, size_{ndims-2}\}, \\
& \quad \{subsize_0, subsize_1, \dots, subsize_{ndims-2}\}, \\
& \quad \{start_0, start_1, \dots, start_{ndims-2}\}, \\
& \quad \text{Subarray}(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, \text{oldtype}))
\end{aligned} \tag{5.4}$$

For an example use of `MPI_TYPE_CREATE_SUBARRAY` in the context of I/O see Section 14.9.2.

5.1.4 Distributed Array Datatype Constructor

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

Advice to users. One can create an HPF-like file view using this type constructor as follows. Complementary filetypes are created by having every process of a group call this constructor with identical arguments (with the exception of `rank` which should be set appropriately). These filetypes (along with identical `disp` and `etype`) are then used to define the view (via `MPI_FILE_SET_VIEW`), see MPI I/O, especially Section 14.1.1 and Section 14.3. Using this view, a collective data access operation (with identical offsets) will yield an HPF-like distribution pattern. (*End of advice to users.*)

```

1 MPI_TYPE_CREATE_DARRAY(size, rank, ndims, array_of_gsizes, array_of_distribs,
2     array_of_dargs, array_of_psizes, order, oldtype, newtype)
3
4     IN     size                size of process group (positive integer)
5     IN     rank                rank in process group (non-negative integer)
6     IN     ndims              number of array dimensions as well as process grid
7     dimensions (positive integer)
8     IN     array_of_gsizes    number of elements of type oldtype in each dimension
9     of global array (array of positive integers)
10
11    IN     array_of_distribs  distribution of array in each dimension (array of
12    states)
13
14    IN     array_of_dargs     distribution argument in each dimension (array of
15    positive integers)
16
17    IN     array_of_psizes    size of process grid in each dimension (array of
18    positive integers)
19
20    IN     order              array storage order flag (state)
21
22    IN     oldtype            old datatype (handle)
23
24    OUT    newtype            new datatype (handle)

```

C binding

```

25 int MPI_Type_create_darray(int size, int rank, int ndims,
26     const int array_of_gsizes[], const int array_of_distribs[],
27     const int array_of_dargs[], const int array_of_psizes[],
28     int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
29
30 int MPI_Type_create_darray_c(int size, int rank, int ndims,
31     const MPI_Count array_of_gsizes[], const int array_of_distribs[],
32     const int array_of_dargs[], const int array_of_psizes[],
33     int order, MPI_Datatype oldtype, MPI_Datatype *newtype)

```

Fortran 2008 binding

```

34 MPI_Type_create_darray(size, rank, ndims, array_of_gsizes, array_of_distribs,
35     array_of_dargs, array_of_psizes, order, oldtype, newtype, ierror)
36     INTEGER, INTENT(IN) :: size, rank, ndims, array_of_gsizes(ndims),
37     array_of_distribs(ndims), array_of_dargs(ndims),
38     array_of_psizes(ndims), order
39     TYPE(MPI_Datatype), INTENT(IN) :: oldtype
40     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
41     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43 MPI_Type_create_darray(size, rank, ndims, array_of_gsizes, array_of_distribs,
44     array_of_dargs, array_of_psizes, order, oldtype, newtype, ierror)
45     !(_c)
46     INTEGER, INTENT(IN) :: size, rank, ndims, array_of_distribs(ndims),
47     array_of_dargs(ndims), array_of_psizes(ndims), order
48     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: array_of_gsizes(ndims)
49     TYPE(MPI_Datatype), INTENT(IN) :: oldtype

```

```

TYPE(MPI_Datatype), INTENT(OUT) :: newtype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran 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_psize`s should be set to 1 (see Example 5.7). For a call to MPI_TYPE_CREATE_DARRAY to be correct, the equation $\prod_{i=0}^{ndims-1} array_of_psizes[i] = size$ must be satisfied. The ordering of processes in the process grid is assumed to be row-major, as in the case of virtual Cartesian process topologies.

Advice to users. For both Fortran and C arrays, the ordering of processes in the process grid is assumed to be row-major. This is consistent with the ordering used in virtual Cartesian process topologies in MPI. To create such virtual process topologies, or to find the coordinates of a process in the process grid, etc., users may use the corresponding process topology functions, see Chapter 8. (*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_psize`s[i] < `array_of_gsize`s[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

$$(\text{array_of_gsizes}[i] + \text{array_of_psizes}[i] - 1) / \text{array_of_psizes}[i].$$

1 If `array_of_dargs[i]` is not `MPI_DISTRIBUTE_DFLT_DARG`, then `MPI_DISTRIBUTE_BLOCK` and
 2 `MPI_DISTRIBUTE_CYCLIC` are equivalent.

3 `MPI_DISTRIBUTE_NONE` is equivalent to `MPI_DISTRIBUTE_CYCLIC` with `array_of_dargs[i]`
 4 set to `array_of_gsizes[i]`.

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

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

```
10 oldtypes[0] = oldtype;
11 for (i = 0; i < ndims; i++) {
12     oldtypes[i+1] = cyclic(array_of_dargs[i],
13                          array_of_gsizes[i],
14                          r[i],
15                          array_of_psizes[i],
16                          oldtypes[i]);
17 }
18 newtype = oldtypes[ndims];
```

19 For `MPI_ORDER_C`, the code is:

```
20 oldtypes[0] = oldtype;
21 for (i = 0; i < ndims; i++) {
22     oldtypes[i+1] = cyclic(array_of_dargs[ndims - i - 1],
23                          array_of_gsizes[ndims - i - 1],
24                          r[ndims - i - 1],
25                          array_of_psizes[ndims - i - 1],
26                          oldtypes[i]);
27 }
28 newtype = oldtypes[ndims];
```

29 where `r[i]` is the position of the process (with rank `rank`) in the process grid at dimension
 30 `i`. The values of `r[i]` are given by the following code fragment:

```
31 t_rank = rank;
32 t_size = 1;
33 for (i = 0; i < ndims; i++)
34     t_size *= array_of_psizes[i];
35 for (i = 0; i < ndims; i++) {
36     t_size = t_size / array_of_psizes[i];
37     r[i] = t_rank / t_size;
38     t_rank = t_rank % t_size;
39 }
40
```

Let the typemap of `oldtype` have the form:

$$\{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\}$$

43 where `typei` is a predefined MPI datatype, and let `ex` be the extent of `oldtype`. The following
 44 function uses the conceptual datatypes `lb_marker` and `ub_marker`, see Section 5.1.6 for details.

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

```
46 cyclic(darg, gsize, r, psize, oldtype)
47     = {(lb_marker, 0),
48
```

```

(type0, disp0 + r × darg × ex), ..., 1
      (typen-1, dispn-1 + r × darg × ex), 2
(type0, disp0 + (r × darg + 1) × ex), ..., 3
      (typen-1, dispn-1 + (r × darg + 1) × ex), 4
... 5
... 6
(type0, disp0 + ((r + 1) × darg - 1) × ex), ..., 7
      (typen-1, dispn-1 + ((r + 1) × darg - 1) × ex), 8
... 9
... 10
(type0, disp0 + r × darg × ex + psize × darg × ex), ..., 11
      (typen-1, dispn-1 + r × darg × ex + psize × darg × ex), 12
(type0, disp0 + (r × darg + 1) × ex + psize × darg × ex), ..., 13
      (typen-1, dispn-1 + (r × darg + 1) × ex + psize × darg × ex), 14
... 15
... 16
(type0, disp0 + ((r + 1) × darg - 1) × ex + psize × darg × ex), ..., 17
      (typen-1, dispn-1 + ((r + 1) × darg - 1) × ex + psize × darg × ex), 18
      ⋮ 19
      ⋮ 20
(type0, disp0 + r × darg × ex + psize × darg × ex × (count - 1)), ..., 21
      (typen-1, dispn-1 + r × darg × ex + psize × darg × ex × (count - 1)), 22
(type0, disp0 + (r × darg + 1) × ex + psize × darg × ex × (count - 1)), ..., 23
      (typen-1, dispn-1 + (r × darg + 1) × ex 24
      + psize × darg × ex × (count - 1)), 25
... 26
... 27
(type0, disp0 + (r × darg + darglast - 1) × ex 28
      + psize × darg × ex × (count - 1)), ..., 29
      (typen-1, dispn-1 + (r × darg + darglast - 1) × ex 30
      + psize × darg × ex × (count - 1)), 31
      (ub_marker, gsize * ex)} 32
... 33
... 34

```

where *count* is defined by this code fragment:

```

nblocks = (gsize + (darg - 1)) / darg; 36
count = nblocks / psize; 37
left_over = nblocks - count * psize; 38
if (r < left_over) 39
    count = count + 1; 40

```

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) 43
    darg_last = darg; 44
else { 45
    darg_last = num_in_last_cyclic - darg * r; 46
    if (darg_last > darg) 47
        darg_last = darg; 48

```

```

1   if (darg_last <= 0)
2       darg_last = darg;
3   }

```

Example 5.7. Consider generating the filetypes corresponding to the HPF distribution:

```

6   <oldtype> FILEARRAY(100, 200, 300)
7   !HPF$ PROCESSORS PROCESSES(2, 3)
8   !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:

```

12  ndims = 3
13  array_of_gsizes(1) = 100
14  array_of_distribs(1) = MPI_DISTRIBUTE_CYCLIC
15  array_of_dargs(1) = 10
16  array_of_gsizes(2) = 200
17  array_of_distribs(2) = MPI_DISTRIBUTE_NONE
18  array_of_dargs(2) = 0
19  array_of_gsizes(3) = 300
20  array_of_distribs(3) = MPI_DISTRIBUTE_BLOCK
21  array_of_dargs(3) = MPI_DISTRIBUTE_DFLT_DARG
22  array_of_psize(1) = 2
23  array_of_psize(2) = 1
24  array_of_psize(3) = 3
25  call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
26  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
27  call MPI_TYPE_CREATE_DARRAY(size, rank, ndims, array_of_gsizes, &
28  array_of_distribs, array_of_dargs, array_of_psize, &
29  MPI_ORDER_FORTRAN, oldtype, newtype, ierr)

```

5.1.5 Address and Size Functions

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

The address of a location in memory can be found by invoking the function `MPI_GET_ADDRESS`. The **relative displacement** between two absolute addresses can be calculated with the function `MPI_AINT_DIFF`. A new absolute address as sum of an absolute base address and a relative displacement can be calculated with the function `MPI_AINT_ADD`. To ensure portability, arithmetic on absolute addresses should not be performed with the intrinsic operators “-” and “+”. See also Sections 2.5.6 and 5.1.12 on pages 19 and 149.

Rationale. Address sized integer values, i.e., `MPI_Aint` or `INTEGER(KIND=MPI_ADDRESS_KIND)` values, are signed integers, while absolute addresses

are unsigned quantities. Direct arithmetic on addresses stored in address sized signed variables can cause overflows, resulting in undefined behavior. (*End of rationale.*)

MPI_GET_ADDRESS(location, address)

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

C binding

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

Fortran 2008 binding

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

Fortran binding

```
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`, which was not defined before MPI-3.0. (*End of rationale.*)

Example 5.8. 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*SIZEOF(REAL); 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 19.1.10–19.1.20. (*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)

Fortran 2008 binding

INTEGER(KIND=MPI_ADDRESS_KIND) MPI_Aint_add(base, disp)

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

Fortran 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 5.1.12. The addition is performed in a manner that results in the correct MPI_Aint representation of the output address, as if the process that originally produced base had called:

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

MPI_AINT_DIFF(addr1, addr2)

IN addr1 minuend address (integer)

IN addr2 subtrahend address (integer)

C binding

MPI_Aint MPI_Aint_diff(MPI_Aint addr1, MPI_Aint addr2)

Fortran 2008 binding

INTEGER(KIND=MPI_ADDRESS_KIND) MPI_Aint_diff(addr1, addr2)

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

Fortran binding

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

INTEGER(KIND=MPI_ADDRESS_KIND) ADDR1, ADDR2

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

The following auxiliary functions provide useful information on derived datatypes.

`MPI_TYPE_SIZE(datatype, size)`

IN	<code>datatype</code>	datatype to get information on (handle)
OUT	<code>size</code>	datatype size (integer)

C binding

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

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

Fortran 2008 binding

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

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

Fortran binding

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

`MPI_TYPE_SIZE_X(datatype, size)`

IN	<code>datatype</code>	datatype to get information on (handle)
OUT	<code>size</code>	datatype size (integer)

C binding

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

Fortran 2008 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
```

Fortran 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`.

5.1.6 Lower-Bound and Upper-Bound Markers

It is often convenient to define explicitly the lower bound and upper bound of a type map, and override the definition given on page 139. 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 5.1.14. Also, the user may want to override the alignment rules that are used to compute upper bounds and extents. E.g., a C compiler may allow the user to override default alignment rules for some of the structures within a program. The user has to specify explicitly the bounds of the datatypes that match these structures.

To achieve this, we add two additional 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 5.9. A call to `MPI_TYPE_CREATE_RESIZED(MPI_INT, -3, 9, type1)` creates a new datatype that has an extent of 9 (from -3 to 5, 5 included), and contains an integer at displacement 0. This is the datatype defined by the typemap $\{(\text{lb_marker}, -3), (\text{int}, 0), (\text{ub_marker}, 6)\}$. 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 $\{(\text{lb_marker}, -3), (\text{int}, 0), (\text{int}, 9), (\text{ub_marker}, 15)\}$. (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

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

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

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

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

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

Then

$$\text{extent}(\text{Typemap}) = \text{ub}(\text{Typemap}) - \text{lb}(\text{Typemap})$$

If type_i requires alignment to a byte address that is a multiple of k_i , then ϵ is the least nonnegative increment needed to round $\text{extent}(\text{Typemap})$ 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 noncontiguous because of such alignment-gaps.

As an example, instead of

```

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

```

one should define

```

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

```

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

In principle, a compiler may define an additional alignment rule for structures, e.g., to use at least 4 or 8 byte alignment, although the content may have a $max_i k_i$ 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 19.1.15. (*End of advice to users.*)

5.1.7 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)
```

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

Fortran 2008 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
```

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

Fortran 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)
```

Fortran 2008 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
```

Fortran 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 5.1).

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)
```

```
int MPI_Type_create_resized_c(MPI_Datatype oldtype, MPI_Count lb,
                              MPI_Count extent, MPI_Datatype *newtype)
```

Fortran 2008 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
```

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

Fortran binding

```

1 MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)
2     INTEGER OLDTYPE, NEWTYPE, IERROR
3     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.

5.1.8 True Extent of Datatypes

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

```

24 MPI_TYPE_GET_TRUE_EXTENT(datatype, true_lb, true_extent)

```

25	IN	<code>datatype</code>	datatype to get information on (handle)
26	OUT	<code>true_lb</code>	true lower bound of datatype (integer)
27	OUT	<code>true_extent</code>	true extent of datatype (integer)

C binding

```

31 int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb,
32     MPI_Aint *true_extent)
33
34 int MPI_Type_get_true_extent_c(MPI_Datatype datatype, MPI_Count *true_lb,
35     MPI_Count *true_extent)

```

Fortran 2008 binding

```

37 MPI_Type_get_true_extent(datatype, true_lb, true_extent, ierror)
38     TYPE(MPI_Datatype), INTENT(IN) :: datatype
39     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: true_lb, true_extent
40     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42 MPI_Type_get_true_extent(datatype, true_lb, true_extent, ierror) !(_c)
43     TYPE(MPI_Datatype), INTENT(IN) :: datatype
44     INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: true_lb, true_extent
45     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

47 MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
48

```

```

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 extent of datatype (integer)

C binding

```

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

```

Fortran 2008 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

```

Fortran 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 that 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

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

Then

$$true_lb(Typemap) = \min_j \{disp_j : type_j \neq lb_marker, ub_marker\},$$

$$true_ub(Typemap) = \max_j \{disp_j + sizeof(type_j) : type_j \neq lb_marker, ub_marker\},$$

and

$$true_extent(Typemap) = true_ub(Typemap) - true_lb(Typemap).$$

(Readers should compare this with the definitions in Section 5.1.6 and Section 5.1.7, 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`.

5.1.9 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)
```

Fortran 2008 binding

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

Fortran 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 5.10. 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) 1

 INOUT datatype datatype that is freed (handle) 2

C binding 4

int MPI_Type_free(MPI_Datatype *datatype) 5

Fortran 2008 binding 7

MPI_Type_free(datatype, ierror) 8

 TYPE(MPI_Datatype), INTENT(INOUT) :: datatype 9

 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 10

Fortran binding 11

MPI_TYPE_FREE(DATATYPE, IERROR) 12

 INTEGER DATATYPE, IERROR 13

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. 14

Advice to implementors. The implementation may keep a reference count of active communications that use the datatype, in order to decide when to free it. Also, one may implement constructors of derived datatypes so that they keep pointers to their datatype arguments, rather than copying them. In this case, one needs to keep track of active datatype definition references in order to know when a datatype object can be freed. (*End of advice to implementors.*) 15

5.1.10 Duplicating a Datatype 16

MPI_TYPE_DUP(oldtype, newtype) 17

 IN oldtype datatype (handle) 18

 OUT newtype copy of oldtype (handle) 19

C binding 20

int MPI_Type_dup(MPI_Datatype oldtype, MPI_Datatype *newtype) 21

Fortran 2008 binding 22

MPI_Type_dup(oldtype, newtype, ierror) 23

 TYPE(MPI_Datatype), INTENT(IN) :: oldtype 24

 TYPE(MPI_Datatype), INTENT(OUT) :: newtype 25

 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 26

Fortran binding 27

MPI_TYPE_DUP(OLDTYPE, NEWTYPE, IERROR) 28

 INTEGER OLDTYPE, NEWTYPE, IERROR 29

MPI_TYPE_DUP is a type constructor that 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 datatype; 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 7.7.4. The new datatype has identical upper bound and lower bound and yields the same net result when fully decoded with the functions in Section 5.1.13. The newtype has the same committed state as the old oldtype.

5.1.11 Use of General Datatypes in Communication

Handles to derived datatypes can be passed to a communication call wherever a datatype argument is required. A call of the form MPI_SEND(buf, count, datatype, ...), where count > 1, is interpreted as if the call was passed a new datatype that 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,

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

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

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

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

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

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

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

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

```

...
CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, type2, ...)
CALL MPI_TYPE_CONTIGUOUS(4, MPI_REAL, type4, ...)
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 any communication in association with a buffer updated by the 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,

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

The received message need not fill all the receive buffer, nor does it need to fill a number of locations that is a multiple of n . Any number, k , of basic elements can be received, where $0 \leq k \leq \text{count} \cdot n$. The number of basic elements received can be retrieved from `status` using the query functions `MPI_GET_ELEMENTS` or `MPI_GET_ELEMENTS_X`.

`MPI_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 (integer)

C binding

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

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

Fortran 2008 binding

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

```

1     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
2
3 MPI_Get_elements(status, datatype, count, ierror) !(_c)
4     TYPE(MPI_Status), INTENT(IN) :: status
5     TYPE(MPI_Datatype), INTENT(IN) :: datatype
6     INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: count
7     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

9 MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
10    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR

```

11

```

13 MPI_GET_ELEMENTS_X(status, datatype, count)

```

```

14
15 IN      status          return status of receive operation (status)
16 IN      datatype       datatype used by receive operation (handle)
17 OUT     count          number of received basic elements (integer)
18

```

19

C binding

```

20 int MPI_Get_elements_x(const MPI_Status *status, MPI_Datatype datatype,
21                       MPI_Count *count)
22

```

Fortran 2008 binding

```

24 MPI_Get_elements_x(status, datatype, count, ierror)
25     TYPE(MPI_Status), INTENT(IN) :: status
26     TYPE(MPI_Datatype), INTENT(IN) :: datatype
27     INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: count
28     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29

```

Fortran binding

```

30 MPI_GET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)
31    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR
32    INTEGER(KIND=MPI_COUNT_KIND) COUNT
33

```

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

38 The previously defined function `MPI_GET_COUNT` (Section 3.2.5), has a different be-
39 havior. It returns the number of “top-level entries” received, i.e., the number of “copies” of
40 type `datatype`. In the previous example, `MPI_GET_COUNT` may return any integer value
41 k , where $0 \leq k \leq \text{count}$. If `MPI_GET_COUNT` returns k , then the number of basic elements
42 received (and the value returned by `MPI_GET_ELEMENTS` or `MPI_GET_ELEMENTS_X`) is
43 $n \cdot k$. If the number of basic elements received is not a multiple of n , that is, if the receive
44 operation has not received an integral number of `datatype` “copies,” then `MPI_GET_COUNT`
45 sets the value of `count` to `MPI_UNDEFINED`.

46

47

48

Example 5.12. 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.*)

5.1.12 Correct Use of Addresses

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

1 Variables belong to the same **sequential storage** if they belong to the same array, to
2 the same COMMON block in Fortran, or to the same structure in C. Valid addresses are defined
3 recursively as follows:
4

- 5 1. The function `MPI_GET_ADDRESS` returns a valid address, when passed as argument
6 a variable of the calling program.
- 7 2. The `buf` argument of a communication function evaluates to a valid address, when
8 passed as argument a variable of the calling program.
- 9 3. If `v` is a valid address, and `i` is an integer, then `v+i` is a valid address, provided `v` and
10 `v+i` are in the same sequential storage.
11
12

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

17 The rules above impose no constraints on the use of derived datatypes, as long as
18 they are used to define a communication buffer that is wholly contained within the same
19 sequential storage. However, the construction of a communication buffer that contains
20 variables that are not within the same sequential storage must obey certain restrictions.
21 Basically, a communication buffer with variables that are not within the same sequential
22 storage can be used only by specifying in the communication call `buf = MPI_BOTTOM`, `count`
23 `= 1`, and using a `datatype` argument where all displacements are valid (absolute) addresses.
24

25 *Advice to users.* It is not expected that MPI implementations will be able to de-
26 tect erroneous, “out of bound” displacements—unless those overflow the user address
27 space—since the MPI call may not know the extent of the arrays and records in the
28 host program. (*End of advice to users.*)
29

30 *Advice to implementors.* There is no need to distinguish (absolute) addresses and
31 (relative) displacements on a machine with contiguous address space: `MPI_BOTTOM`
32 is zero, and both addresses and displacements are integers. On machines where the
33 distinction is required, addresses are recognized as expressions that involve
34 `MPI_BOTTOM`. (*End of advice to implementors.*)
35

36 5.1.13 Decoding a Datatype

37 MPI datatype objects allow users to specify an arbitrary layout of data in memory. There
38 are several cases where accessing the layout information in opaque datatype objects would
39 be useful. The opaque datatype object has found a number of uses outside MPI. Further-
40 more, a number of tools wish to display internal information about a datatype. To achieve
41 this, datatype decoding functions are provided. The two functions in this section are used
42 together to decode datatypes to recreate the calling sequence used in their initial defini-
43 tion. These can be used to allow a user to determine the type map and type signature of a
44 datatype.
45
46
47
48

MPI_TYPE_GET_ENVELOPE(datatype, num_integers, num_addresses, num_large_counts,			1
num_datatypes, combiner)			2
IN	datatype	datatype to decode (handle)	3
OUT	num_integers	number of input integers used in call constructing	4
		combiner (non-negative integer)	5
OUT	num_addresses	number of input addresses used in call constructing	6
		combiner (non-negative integer)	7
OUT	num_large_counts	number of input large counts used in call	8
		constructing combiner (non-negative integer, only	9
		present for large count variants)	10
OUT	num_datatypes	number of input datatypes used in call constructing	11
		combiner (non-negative integer)	12
OUT	combiner	combiner (state)	13
			14
			15
			16
			17
C binding			18
int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers,			19
int *num_addresses, int *num_datatypes, int *combiner)			20
int MPI_Type_get_envelope_c(MPI_Datatype datatype, MPI_Count *num_integers,			21
MPI_Count *num_addresses, MPI_Count *num_large_counts,			22
MPI_Count *num_datatypes, int *combiner)			23
			24
Fortran 2008 binding			25
MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes,			26
combiner, ierror)			27
TYPE(MPI_Datatype), INTENT(IN) :: datatype			28
INTEGER, INTENT(OUT) :: num_integers, num_addresses, num_datatypes,			29
combiner			30
INTEGER, OPTIONAL, INTENT(OUT) :: ierror			31
MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_large_counts,			32
num_datatypes, combiner, ierror) !(_c)			33
TYPE(MPI_Datatype), INTENT(IN) :: datatype			34
INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: num_integers, num_addresses,			35
num_large_counts, num_datatypes			36
INTEGER, INTENT(OUT) :: combiner			37
INTEGER, OPTIONAL, INTENT(OUT) :: ierror			38
			39
Fortran binding			40
MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES,			41
COMBINER, IERROR)			42
INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER,			43
IERROR			44

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

Table 5.1: combiner values returned from MPI_TYPE_GET_ENVELOPE

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

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

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

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

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

If combiner is MPI_COMBINER_NAMED then datatype is a named predefined datatype.

If the MPI_TYPE_GET_ENVELOPE variant without num_large_counts is invoked with a datatype that requires an output value of num_large_counts > 0, then an error of class MPI_ERR_TYPE is raised.

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

The actual arguments used in the creation call for a datatype can be obtained using MPI_TYPE_GET_CONTENTS.

MPI_TYPE_GET_ENVELOPE and MPI_TYPE_GET_CONTENTS also support large

count types in separate additional MPI procedures in C (suffixed with the “_c”) and interface polymorphism in Fortran when using USE mpi_f08.

```
MPI_TYPE_GET_CONTENTS(datatype, max_integers, max_addresses, max_large_counts,
    max_datatypes, array_of_integers, array_of_addresses, array_of_large_counts,
    array_of_datatypes)
```

IN	datatype	datatype to decode (handle)	8
IN	max_integers	number of elements in array_of_integers (non-negative integer)	9 10 11
IN	max_addresses	number of elements in array_of_addresses (non-negative integer)	12 13
IN	max_large_counts	number of elements in array_of_large_counts (non-negative integer, only present for large count variants)	14 15 16
IN	max_datatypes	number of elements in array_of_datatypes (non-negative integer)	17 18 19
OUT	array_of_integers	contains integer arguments used in constructing datatype (array of integers)	20 21
OUT	array_of_addresses	contains address arguments used in constructing datatype (array of integers)	22 23
OUT	array_of_large_counts	contains large count arguments used in constructing datatype (array of integers, only present for large count variants)	24 25 26
OUT	array_of_datatypes	contains datatype arguments used in constructing datatype (array of handles)	27 28 29

C binding

```
int MPI_Type_get_contents(MPI_Datatype datatype, int max_integers,
    int max_addresses, int max_datatypes, int array_of_integers[],
    MPI_Aint array_of_addresses[], MPI_Datatype array_of_datatypes[])
```

```
int MPI_Type_get_contents_c(MPI_Datatype datatype, MPI_Count max_integers,
    MPI_Count max_addresses, MPI_Count max_large_counts,
    MPI_Count max_datatypes, int array_of_integers[],
    MPI_Aint array_of_addresses[], MPI_Count array_of_large_counts[],
    MPI_Datatype array_of_datatypes[])
```

Fortran 2008 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)
```

```

1     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::
2         array_of_addresses(max_addresses)
3     TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes)
4     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
6     MPI_Type_get_contents(datatype, max_integers, max_addresses, max_large_counts,
7         max_datatypes, array_of_integers, array_of_addresses,
8         array_of_large_counts, array_of_datatypes, ierror) !(_c)
9     TYPE(MPI_Datatype), INTENT(IN) :: datatype
10    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: max_integers, max_addresses,
11        max_large_counts, max_datatypes
12    INTEGER, INTENT(OUT) :: array_of_integers(max_integers)
13    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::
14        array_of_addresses(max_addresses)
15    INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) ::
16        array_of_large_counts(max_large_counts)
17    TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes)
18    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

19    MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,
20        ARRAY_OF_INTEGERS, ARRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES,
21        IERROR)
22
23    INTEGER DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,
24        ARRAY_OF_INTEGERS(*), ARRAY_OF_DATATYPES(*), IERROR
25    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, max_large_counts, and max_datatypes must be at least as large as the value returned in num_integers, num_addresses, num_large_counts, and num_datatypes, respectively, in the call MPI_TYPE_GET_ENVELOPE for the same datatype argument.

Rationale. The arguments max_integers, max_addresses, max_large_counts, and max_datatypes allow for error checking in the call. (*End of rationale.*)

If the MPI_TYPE_GET_CONTENTS variant without max_large_counts is invoked with a datatype that requires > 0 values in array_of_large_counts, then an error of class MPI_ERR_TYPE is raised.

Rationale. The large count variant of this MPI procedure was added in MPI-4. It contains new max_large_counts and array_of_large_counts parameters. The other variant—the variant that existed before MPI-4—was not changed in order to preserve backwards compatibility. (*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:

```

1  PARAMETER (LARGE = 1000)
2  INTEGER DTYPE, NI, NA, ND, COMBINER, I(LARGE), D(LARGE), IERROR
3  INTEGER(KIND=MPI_ADDRESS_KIND) A(LARGE)
4  ! CONSTRUCT DATATYPE DTYPE (NOT SHOWN)
5  CALL MPI_TYPE_GET_ENVELOPE(DTYPE, NI, NA, ND, COMBINER, IERROR)
6  IF ((NI .GT. LARGE) .OR. (NA .GT. LARGE) .OR. (ND .GT. LARGE)) THEN
7      WRITE (*, *) "NI, NA, OR ND = ", NI, NA, ND, &
8          " RETURNED BY MPI_TYPE_GET_ENVELOPE IS LARGER THAN LARGE = ", LARGE
9      CALL MPI_ABORT(MPI_COMM_WORLD, 99, IERROR)
10 ENDIF
11 CALL MPI_TYPE_GET_CONTENTS(DTYPE, NI, NA, ND, I, A, D, IERROR)

```

or in C the analogous calls of:

```

12 #define LARGE 1000
13 int ni, na, nd, combiner, i[LARGE];
14 MPI_Aint a[LARGE];
15 MPI_Datatype dtype, d[LARGE];
16 /* construct datatype dtype (not shown) */
17 MPI_Type_get_envelope(dtype, &ni, &na, &nd, &combiner);
18 if ((ni > LARGE) || (na > LARGE) || (nd > LARGE)) {
19     fprintf(stderr, "ni, na, or nd = %d %d %d returned by ", ni, na, nd);
20     fprintf(stderr, "MPI_Type_get_envelope is larger than LARGE = %d\n",
21         LARGE);
22     MPI_Abort(MPI_COMM_WORLD, 99);
23 }
24 MPI_Type_get_contents(dtype, ni, na, nd, i, a, d);

```

The following describes the values of the arguments for each combiner. The lower case name of arguments is used. Also, the descriptions below refer to MPI datatypes created with procedures without large count arguments.

MPI_COMBINER_NAMED the datatype represent a predefined type and therefore it is erroneous to call `MPI_TYPE_GET_CONTENTS`.

MPI_COMBINER_DUP `ni = 0, na = 0, nd = 1`, and

Constructor argument	C	Fortran location
oldtype	d[0]	D(1)

MPI_COMBINER_CONTIGUOUS `ni = 1, na = 0, nd = 1`, and

Constructor argument	C	Fortran location
count	i[0]	I(1)
oldtype	d[0]	D(1)

MPI_COMBINER_VECTOR `ni = 3, na = 0, nd = 1`, and

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)

MPI_COMBINER_HVECTOR `ni = 2, na = 1, nd = 1`, and

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)

MPI_COMBINER_INDEXED ni = 2*count+1, na = 0, nd = 1, and

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)

MPI_COMBINER_HINDEXED ni = count+1, na = count, nd = 1, and

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)

MPI_COMBINER_INDEXED_BLOCK ni = count+2, na = 0, nd = 1, and

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)

MPI_COMBINER_HINDEXED_BLOCK ni = 2, na = count, nd = 1, and

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)

MPI_COMBINER_STRUCT ni = count+1, na = count, nd = count, and

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))

MPI_COMBINER_SUBARRAY ni = 3*ndims+2, na = 0, nd = 1, and

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)

MPI_COMBINER_DARRAY ni = 4*ndims+4, na = 0, nd = 1, and

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)

MPI_COMBINER_F90_REAL ni = 2, na = 0, nd = 0, and

Constructor argument	C	Fortran location
p	i[0]	I(1)
r	i[1]	I(2)

MPI_COMBINER_F90_COMPLEX ni = 2, na = 0, nd = 0, and

Constructor argument	C	Fortran location
p	i[0]	I(1)
r	i[1]	I(2)

MPI_COMBINER_F90_INTEGER ni = 1, na = 0, nd = 0, and

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

MPI_COMBINER_RESIZED ni = 0, na = 2, nd = 1, and

Constructor argument	C	Fortran location
lb	a[0]	A(1)
extent	a[1]	A(2)
oldtype	d[0]	D(1)

5.1.14 Examples

The following examples illustrate the use of derived datatypes.

Example 5.13. Send and receive a section of a 3D array.

```
REAL a(100,100,100), e(9,9,9)
INTEGER oneslice, twoslice, threeslice, myrank, ierr
```

```

1  INTEGER(KIND=MPI_ADDRESS_KIND) lb, sizeofreal
2  INTEGER status(MPI_STATUS_SIZE)
3
4  ! extract the section a(1:17:2, 3:11, 2:10)
5  ! and store it in e(:, :, :).
6
7  CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
8
9  CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lb, sizeofreal, ierr)
10
11 ! create datatype for a 1D section
12 CALL MPI_TYPE_VECTOR(9, 1, 2, MPI_REAL, oneslice, ierr)
13
14 ! create datatype for a 2D section
15 CALL MPI_TYPE_CREATE_HVECTOR(9, 1, 100*sizeofreal, oneslice, &
16                               twoslice, ierr)
17
18 ! create datatype for the entire section
19 CALL MPI_TYPE_CREATE_HVECTOR(9, 1, 100*100*sizeofreal, twoslice, &
20                               threeslice, ierr)
21
22 CALL MPI_TYPE_COMMIT(threeslice, ierr)
23 CALL MPI_SENDRECV(a(1,3,2), 1, threeslice, myrank, 0, e, 9*9*9, &
24                   MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

```

Example 5.14. Copy the (strictly) lower triangular part of a matrix.

```

26 REAL a(100,100), b(100,100)
27 INTEGER disp(100), blocklen(100), ltype, myrank, ierr
28 INTEGER status(MPI_STATUS_SIZE)
29
30 ! copy lower triangular part of array a
31 ! onto lower triangular part of array b
32
33 CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
34
35 ! compute start and size of each column
36 DO i=1,100
37     disp(i) = 100*(i-1) + i
38     blocklen(i) = 100-i
39 END DO
40
41 ! create datatype for lower triangular part
42 CALL MPI_TYPE_INDEXED(100, blocklen, disp, MPI_REAL, ltype, ierr)
43
44 CALL MPI_TYPE_COMMIT(ltype, ierr)
45 CALL MPI_SENDRECV(a, 1, ltype, myrank, 0, b, 1, &
46                   ltype, myrank, 0, MPI_COMM_WORLD, status, ierr)
47
48

```

Example 5.15. Transpose a matrix.

```

1  REAL a(100,100), b(100,100)
2  INTEGER row, xpose, myrank, ierr
3  INTEGER(KIND=MPI_ADDRESS_KIND) lb, sizeofreal
4  INTEGER status(MPI_STATUS_SIZE)
5
6  ! transpose matrix a onto b
7
8  CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
9
10 CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lb, sizeofreal, ierr)
11
12 ! create datatype for one row
13 CALL MPI_TYPE_VECTOR(100, 1, 100, MPI_REAL, row, ierr)
14
15 ! create datatype for matrix in row-major order
16 CALL MPI_TYPE_CREATE_HVECTOR(100, 1, sizeofreal, row, xpose, ierr)
17
18 CALL MPI_TYPE_COMMIT(xpose, ierr)
19
20 ! send matrix in row-major order and receive in column major order
21 CALL MPI_SENDRECV(a, 1, xpose, myrank, 0, b, 100*100, &
22                  MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
23

```

Example 5.16. Another approach to the transpose problem:

```

24  REAL a(100,100), b(100,100)
25  INTEGER row, row1
26  INTEGER(KIND=MPI_ADDRESS_KIND) lb, sizeofreal
27  INTEGER myrank, ierr
28  INTEGER status(MPI_STATUS_SIZE)
29
30  CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
31
32 ! transpose matrix a onto b
33
34 CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lb, sizeofreal, ierr)
35
36 ! create datatype for one row
37 CALL MPI_TYPE_VECTOR(100, 1, 100, MPI_REAL, row, ierr)
38
39 ! create datatype for one row, with the extent of one real number
40 lb = 0
41 CALL MPI_TYPE_CREATE_RESIZED(row, lb, sizeofreal, row1, ierr)
42
43 CALL MPI_TYPE_COMMIT(row1, ierr)
44
45 ! send 100 rows and receive in column major order
46 CALL MPI_SENDRECV(a, 100, row1, myrank, 0, b, 100*100, &
47                  MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
48

```


Example 5.17. Use of MPI datatypes to 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);

/* Since the compiler may pad the structure, it is best to explicitly
   set the extent of the MPI datatype for a structure element using
   MPI_Type_create_resized */

/* compute extent of the structure */
MPI_Get_address(particle+1, &sizeofentry);
sizeofentry = MPI_Aint_diff(sizeofentry, base);

/* build datatype describing structure */
MPI_Type_create_resized(Particlestruct, 0, sizeofentry, &Particletype);

/* 4.1: send the entire array */

MPI_Type_commit(&Particletype);
MPI_Send(particle, 1000, Particletype, dest, tag, comm);

/* 4.2: send only the entries of type zero particles,

```

```

1      preceded by the number of such entries */
2
3  MPI_Datatype Zparticles; /* datatype describing all particles
4                          with type zero (needs to be recomputed
5                          if types change) */
6
7  MPI_Datatype Ztype;
8
9  int      zdisp[1000];
10 int      zblock[1000], j, k;
11 int      zzblock[2] = {1,1};
12 MPI_Aint  zzdisp[2];
13 MPI_Datatype zztype[2];
14
15 /* compute displacements of type zero particles */
16 j = 0;
17 for (i=0; i < 1000; i++)
18     if (particle[i].type == 0)
19     {
20         zdisp[j] = i;
21         zblock[j] = 1;
22         j++;
23     }
24
25 /* create datatype for type zero particles */
26 MPI_Type_indexed(j, zblock, zdisp, Particletype, &Zparticles);
27
28 /* prepend particle count */
29 MPI_Get_address(&j, zzdisp);
30 MPI_Get_address(particle, zzdisp+1);
31 zztype[0] = MPI_INT;
32 zztype[1] = Zparticles;
33 MPI_Type_create_struct(2, zzblock, zzdisp, zztype, &Ztype);
34
35 MPI_Type_commit(&Ztype);
36 MPI_Send(MPI_BOTTOM, 1, Ztype, dest, tag, comm);
37
38 /* A probably more efficient way of defining Zparticles */
39
40 /* consecutive particles with index zero are handled as one block */
41 j=0;
42 for (i=0; i < 1000; i++)
43     if (particle[i].type == 0)
44     {
45         for (k=i+1; (k < 1000)&&(particle[k].type == 0); k++);
46         zdisp[j] = i;
47         zblock[j] = k-i;
48         j++;
49         i = k;
50     }
51 MPI_Type_indexed(j, zblock, zdisp, Particletype, &Zparticles);

```

```

1  /* 4.3: send the first two coordinates of all entries */
2
3  MPI_Datatype Allpairs;      /* datatype for all pairs of coordinates */
4
5  MPI_Type_get_extent(Particletype, &lb, &sizeofentry);
6
7  /* sizeofentry can also be computed by subtracting the address
8     of particle[0] from the address of particle[1] */
9
10 MPI_Type_create_hvector(1000, 2, sizeofentry, MPI_DOUBLE, &Allpairs);
11 MPI_Type_commit(&Allpairs);
12 MPI_Send(particle[0].d, 1, Allpairs, dest, tag, comm);
13
14 /* an alternative solution to 4.3 */
15
16 MPI_Datatype Twodouble;
17
18 MPI_Type_contiguous(2, MPI_DOUBLE, &Twodouble);
19
20 MPI_Datatype Onepair;      /* datatype for one pair of coordinates, with
21                             the extent of one particle entry */
22
23 MPI_Type_create_resized(Twodouble, 0, sizeofentry, &Onepair );
24 MPI_Type_commit(&Onepair);
25 MPI_Send(particle[0].d, 1000, Onepair, dest, tag, comm);

```

Example 5.18. The same manipulations as in the previous example, but use absolute addresses in datatypes.

```

26
27
28 struct Partstruct
29 {
30     int    type;
31     double d[6];
32     char   b[7];
33 };
34
35 struct Partstruct particle[1000];
36
37 /* build datatype describing first array entry */
38
39 MPI_Datatype Particletype;
40 MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
41 int          block[3] = {1, 6, 7};
42 MPI_Aint     disp[3];
43
44 MPI_Get_address(particle, disp);
45 MPI_Get_address(particle[0].d, disp+1);
46 MPI_Get_address(particle[0].b, disp+2);
47 MPI_Type_create_struct(3, block, disp, type, &Particletype);
48
49 /* Particletype describes first array entry -- using absolute
50    addresses */

```

```

1
2  /* 5.1: send the entire array */
3
4  MPI_Type_commit(&Particletype);
5  MPI_Send(MPI_BOTTOM, 1000, Particletype, dest, tag, comm);
6
7
8  /* 5.2: send the entries of type zero,
9         preceded by the number of such entries */
10
11 MPI_Datatype Zparticles, Ztype;
12
13 int          zdisp[1000];
14 int          zblock[1000], i, j, k;
15 int          zzbblock[2] = {1,1};
16 MPI_Datatype zztype[2];
17 MPI_Aint     zzdisp[2];
18
19 j=0;
20 for (i=0; i < 1000; i++)
21     if (particle[i].type == 0)
22     {
23         for (k=i+1; (k < 1000)&&(particle[k].type == 0); k++);
24         zdisp[j] = i;
25         zblock[j] = k-i;
26         j++;
27         i = k;
28     }
29 MPI_Type_indexed(j, zblock, zdisp, Particletype, &Zparticles);
30 /* Zparticles describe particles with type zero, using
31    their absolute addresses*/
32
33 /* prepend particle count */
34 MPI_Get_address(&j, zzdisp);
35 zzdisp[1] = (MPI_Aint)0;
36 zztype[0] = MPI_INT;
37 zztype[1] = Zparticles;
38 MPI_Type_create_struct(2, zzbblock, zzdisp, zztype, &Ztype);
39
40 MPI_Type_commit(&Ztype);
41 MPI_Send(MPI_BOTTOM, 1, Ztype, dest, tag, comm);

```

Example 5.19. This example shows how datatypes can be used to handle unions.

```

41 union {
42     int    ival;
43     float fval;
44 } u[1000];
45
46 int    i, utype;
47
48 /* All entries of u have identical type; variable

```

```

    utype keeps track of their current type */
1
2
MPI_Datatype  mpi_utype[2];
MPI_Aint      ubase, extent;
3
4
/* compute an MPI datatype for each possible union type;
   assume values are left-aligned in union storage. */
5
6
MPI_Get_address(u, &ubase);
MPI_Get_address(u+1, &extent);
extent = MPI_Aint_diff(extent, ubase);
7
8
9
MPI_Type_create_resized(MPI_INT, 0, extent, &mpi_utype[0]);
10
11
MPI_Type_create_resized(MPI_FLOAT, 0, extent, &mpi_utype[1]);
12
13
for(i=0; i<2; i++) MPI_Type_commit(&mpi_utype[i]);
14
15
/* actual communication */
MPI_Send(u, 1000, mpi_utype[utype], dest, tag, comm);
16
17
18
19

```

Example 5.20. This example shows how a datatype can be decoded. The routine `printdatatype` prints out the elements of the datatype. Note the use of `MPI_Type_free` for datatypes that are not predefined.

```

/*
   Example of decoding a datatype.
25
   Returns 0 if the datatype is predefined, 1 otherwise
   */
26
27
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
28
29
int printdatatype(MPI_Datatype datatype)
30
31
{
    int *array_of_ints;
    MPI_Aint *array_of_adds;
    MPI_Datatype *array_of_dtypes;
    int num_ints, num_adds, num_dtypes, combiner;
    int i;
32
33
34
35
36
37
38
    MPI_Type_get_envelope(datatype,
        &num_ints, &num_adds, &num_dtypes, &combiner);
39
40
    switch (combiner) {
    case MPI_COMBINER_NAMED:
41
        printf("Datatype is named:");
42
        /* 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 preferred to use
           names that the user may have changed.
43
44
45
           */
        if (datatype == MPI_INT) printf("MPI_INT\n");
46
47
48

```

```

1     else if (datatype == MPI_DOUBLE) printf("MPI_DOUBLE\n");
2     ... else test for other types ...
3     return 0;
4     break;
5     case MPI_COMBINER_STRUCT:
6         printf("Datatype is struct containing");
7         array_of_ints = (int *)malloc(num_ints * sizeof(int));
8         array_of_adds =
9             (MPI_Aint *) malloc(num_adds * sizeof(MPI_Aint));
10        array_of_dtypes = (MPI_Datatype *)
11            malloc(num_dtypes * sizeof(MPI_Datatype));
12        MPI_Type_get_contents(datatype, num_ints, num_adds, num_dtypes,
13            array_of_ints, array_of_adds, array_of_dtypes);
14        printf(" %d datatypes:\n", array_of_ints[0]);
15        for (i=0; i<array_of_ints[0]; i++) {
16            printf("blocklength %d, displacement %ld, type:\n",
17                array_of_ints[i+1], (long)array_of_adds[i]);
18            if (printdatatype(array_of_dtypes[i])) {
19                /* Note that we free the type ONLY if it
20                 is not predefined */
21                MPI_Type_free(&array_of_dtypes[i]);
22            }
23        }
24        free(array_of_ints);
25        free(array_of_adds);
26        free(array_of_dtypes);
27        break;
28        ... other combiner values ...
29    default:
30        printf("Unrecognized combiner type\n");
31    }
32    return 1;
33 }

```

5.2 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 5.1, 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)	1
IN inbuf	2
	3
IN incount	4
IN datatype	5
OUT outbuf	6
	7
IN outsize	8
INOUT position	9
	10
IN comm	11

C binding

```

int MPI_Pack(const void *inbuf, int incount, MPI_Datatype datatype,
             void *outbuf, int outsize, int *position, MPI_Comm comm)
int MPI_Pack_c(const void *inbuf, MPI_Count incount, MPI_Datatype datatype,
               void *outbuf, MPI_Count outsize, MPI_Count *position,
               MPI_Comm comm)

```

Fortran 2008 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
MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror)
  !(_c)
  TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
  INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: incount, outsize
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(*), DIMENSION(..) :: outbuf
  INTEGER(KIND=MPI_COUNT_KIND), INTENT(INOUT) :: position
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran 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`).

1 The input value of `position` is the first location in the output buffer to be used for
 2 packing. `position` is incremented by the size of the packed message, and the output value
 3 of `position` is the first location in the output buffer following the locations occupied by the
 4 packed message. The `comm` argument is the communicator that will be subsequently used
 5 for sending the packed message.
 6

7
 8 **MPI_UNPACK**(`inbuf`, `insize`, `position`, `outbuf`, `outcount`, `datatype`, `comm`)

9 IN `inbuf` input buffer start (choice)
 10 IN `insize` size of input buffer, in bytes (non-negative integer)
 11 INOUT `position` current position in bytes (integer)
 12 OUT `outbuf` output buffer start (choice)
 13 IN `outcount` number of items to be unpacked (integer)
 14 IN `datatype` datatype of each output data item (handle)
 15 IN `comm` communicator for packed message (handle)
 16
 17
 18

19 C binding

20 `int MPI_Unpack(const void *inbuf, int insize, int *position, void *outbuf,`
 21 `int outcount, MPI_Datatype datatype, MPI_Comm comm)`

22
 23 `int MPI_Unpack_c(const void *inbuf, MPI_Count insize, MPI_Count *position,`
 24 `void *outbuf, MPI_Count outcount, MPI_Datatype datatype,`
 25 `MPI_Comm comm)`

26 Fortran 2008 binding

27 `MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm, ierror)`

28 `TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf`

29 `INTEGER, INTENT(IN) :: insize, outcount`

30 `INTEGER, INTENT(INOUT) :: position`

31 `TYPE(*), DIMENSION(..) :: outbuf`

32 `TYPE(MPI_Datatype), INTENT(IN) :: datatype`

33 `TYPE(MPI_Comm), INTENT(IN) :: comm`

34 `INTEGER, OPTIONAL, INTENT(OUT) :: ierror`
 35

36 `MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm, ierror)`

37 `!(_c)`

38 `TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf`

39 `INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: insize, outcount`

40 `INTEGER(KIND=MPI_COUNT_KIND), INTENT(INOUT) :: position`

41 `TYPE(*), DIMENSION(..) :: outbuf`

42 `TYPE(MPI_Datatype), INTENT(IN) :: datatype`

43 `TYPE(MPI_Comm), INTENT(IN) :: comm`

44 `INTEGER, OPTIONAL, INTENT(OUT) :: ierror`
 45

46 Fortran binding

47 `MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM, IERROR)`

48 `<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

1 of a packing unit as a separate packing unit. Each packing unit, that was created by a related
 2 sequence of pack calls, or by a regular send, must be unpacked as a unit, by a sequence of
 3 related unpack calls.

4
 5 *Rationale.* The restriction on “atomic” packing and unpacking of packing units
 6 allows the implementation to add at the head of packing units additional information,
 7 such as a description of the sender architecture (to be used for type conversion, in a
 8 heterogeneous environment) (*End of rationale.*)

9
 10 The following call allows the user to find out how much space is needed to pack a
 11 message and, thus, manage space allocation for buffers.

12
 13 MPI_PACK_SIZE(incount, datatype, comm, size)

14	IN	incount	count argument to packing call (non-negative integer)
15	IN	datatype	datatype argument to packing call (handle)
16	IN	comm	communicator argument to packing call (handle)
17	OUT	size	upper bound on size of packed message, in bytes (non-negative integer)
18			
19			
20			
21			

22 C binding

23 int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm, int *size)

24
 25 int MPI_Pack_size_c(MPI_Count incount, MPI_Datatype datatype, MPI_Comm comm,
 26 MPI_Count *size)

27 Fortran 2008 binding

28 MPI_Pack_size(incount, datatype, comm, size, ierror)

29 INTEGER, INTENT(IN) :: incount
 30 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 31 TYPE(MPI_Comm), INTENT(IN) :: comm
 32 INTEGER, INTENT(OUT) :: size
 33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

34
 35 MPI_Pack_size(incount, datatype, comm, size, ierror) !(_c)

36 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: incount
 37 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 38 TYPE(MPI_Comm), INTENT(IN) :: comm
 39 INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
 40 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

41 Fortran binding

42 MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)

43 INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR

44
 45 A call to MPI_PACK_SIZE(incount, datatype, comm, size) returns in size an upper bound
 46 on the increment in position that is effected by a call to MPI_PACK(inbuf, incount, datatype,
 47 outbuf, outcount, position, comm). If the packed size of the datatype cannot be expressed
 48 by the size parameter, then MPI_PACK_SIZE sets the value of size to MPI_UNDEFINED.

Rationale. The call returns an upper bound, rather than an exact bound, since the exact amount of space needed to pack the message may depend on the context (e.g., first message packed in a packing unit may take more space). (*End of rationale.*)

Example 5.21. An example using MPI_PACK.

```

int      position, i, j, a[2];
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 5.22. An elaborate example.

```

int      position, i = 200;
float    a[200];
char     buff[1000]; /* larger than or equal to the size returned
                      from MPI_PACK_SIZE for 1,newtype */
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0)
{
    /* SENDER CODE */
    int len[2];
    MPI_Aint disp[2];
    MPI_Datatype type[2], newtype;

    /* build datatype for i followed by a[0]...a[i-1] */
    len[0] = 1;
    len[1] = i;
    MPI_Get_address(&i, disp);
    MPI_Get_address(a, disp+1);
    type[0] = MPI_INT;
    type[1] = MPI_FLOAT;
    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,

```

```

1      MPI_COMM_WORLD);
2
3      /* *****
4      One can replace the last three lines with
5      MPI_Send(MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
6      ***** */
7  }
8  else if (myrank == 1)
9  {
10     /* RECEIVER CODE */
11     MPI_Status status;
12
13     /* Receive */
14     MPI_Recv(buff, 1000, MPI_PACKED, 0, 0, MPI_COMM_WORLD, &status);
15
16     /* Unpack i */
17     position = 0;
18     MPI_Unpack(buff, 1000, &position, &i, 1, MPI_INT, MPI_COMM_WORLD);
19
20     /* Unpack a[0]...a[i-1] */
21     MPI_Unpack(buff, 1000, &position, a, i, MPI_FLOAT, MPI_COMM_WORLD);
22 }

```

Example 5.23. Each process sends a count, followed by count characters to the root; the root concatenates all characters into one string.

```

25 int count, gsize, counts[64], totalcount, k1, k2, k,
26     displs[64], position, concat_pos;
27 char chr[100], *lbuf, *rbuf, *cbuf;
28
29 MPI_Comm_size(comm, &gsize);
30 MPI_Comm_rank(comm, &myrank);
31
32     /* allocate local pack buffer */
33 MPI_Pack_size(1, MPI_INT, comm, &k1);
34 MPI_Pack_size(count, MPI_CHAR, comm, &k2);
35 k = k1+k2;
36 lbuf = (char *)malloc(k);
37
38     /* pack count, followed by count characters */
39 position = 0;
40 MPI_Pack(&count, 1, MPI_INT, lbuf, k, &position, comm);
41 MPI_Pack(chr, count, MPI_CHAR, lbuf, k, &position, comm);
42
43 if (myrank != root) {
44     /* gather at root sizes of all packed messages */
45     MPI_Gather(&position, 1, MPI_INT, NULL, 0,
46              MPI_DATATYPE_NULL, root, comm);
47
48     /* gather at root packed messages */
49     MPI_Gatherv(lbuf, position, MPI_PACKED, NULL,
50               NULL, NULL, MPI_DATATYPE_NULL, root, comm);

```

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
} 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[gsiz-1] + counts[gsiz-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';
}

```

5.3 Canonical MPI_PACK and MPI_UNPACK

These functions read/write data to/from the buffer in the "external32" data format specified in Section 14.5.2, 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 `daterep` 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.*)

```

1 MPI_PACK_EXTERNAL(datarep, inbuf, incount, datatype, outbuf, outsize, position)
2   IN      datarep      data representation (string)
3   IN      inbuf        input buffer start (choice)
4   IN      incount      number of input data items (integer)
5   IN      datatype     datatype of each input data item (handle)
6   OUT     outbuf       output buffer start (choice)
7   IN      outsize      output buffer size, in bytes (integer)
8   INOUT   position     current position in buffer, in bytes (integer)

```

C binding

```

13 int MPI_Pack_external(const char datarep[], const void *inbuf, int incount,
14                     MPI_Datatype datatype, void *outbuf, MPI_Aint outsize,
15                     MPI_Aint *position)
16
17 int MPI_Pack_external_c(const char datarep[], const void *inbuf,
18                        MPI_Count incount, MPI_Datatype datatype, void *outbuf,
19                        MPI_Count outsize, MPI_Count *position)
20

```

Fortran 2008 binding

```

21 MPI_PACK_EXTERNAL(datarep, inbuf, incount, datatype, outbuf, outsize, position,
22                  ierror)
23   CHARACTER(LEN=*), INTENT(IN) :: datarep
24   TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
25   INTEGER, INTENT(IN) :: incount
26   TYPE(MPI_Datatype), INTENT(IN) :: datatype
27   TYPE(*), DIMENSION(..) :: outbuf
28   INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: outsize
29   INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position
30   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32 MPI_PACK_EXTERNAL(datarep, inbuf, incount, datatype, outbuf, outsize, position,
33                  ierror) !(_c)
34   CHARACTER(LEN=*), INTENT(IN) :: datarep
35   TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
36   INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: incount, outsize
37   TYPE(MPI_Datatype), INTENT(IN) :: datatype
38   TYPE(*), DIMENSION(..) :: outbuf
39   INTEGER(KIND=MPI_COUNT_KIND), INTENT(INOUT) :: position
40   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41

```

Fortran binding

```

42 MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION,
43                  IERROR)
44   CHARACTER*(*) DATAREP
45   <type> INBUF(*), OUTBUF(*)
46   INTEGER INCOUNT, DATATYPE, IERROR
47   INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, POSITION
48

```

MPI_UNPACK_EXTERNAL(datarep, inbuf, insize, position, outbuf, outcount, datatype)	1
IN datarep	2
	3
IN inbuf	4
IN insize	5
INOUT position	6
OUT outbuf	7
IN outcount	8
IN datatype	9
	10
	11
	12
C binding	13
int MPI_Unpack_external(const char datarep[], const void *inbuf,	14
MPI_Aint insize, MPI_Aint *position, void *outbuf, int outcount,	15
MPI_Datatype datatype)	16
int MPI_Unpack_external_c(const char datarep[], const void *inbuf,	17
MPI_Count insize, MPI_Count *position, void *outbuf,	18
MPI_Count outcount, MPI_Datatype datatype)	19
	20
Fortran 2008 binding	21
MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outcount,	22
datatype, ierror)	23
CHARACTER(LEN=*), INTENT(IN) :: datarep	24
TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf	25
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: insize	26
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position	27
TYPE(*), DIMENSION(..) :: outbuf	28
INTEGER, INTENT(IN) :: outcount	29
TYPE(MPI_Datatype), INTENT(IN) :: datatype	30
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	31
	32
MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outcount,	33
datatype, ierror) !(_c)	34
CHARACTER(LEN=*), INTENT(IN) :: datarep	35
TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf	36
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: insize, outcount	37
INTEGER(KIND=MPI_COUNT_KIND), INTENT(INOUT) :: position	38
TYPE(*), DIMENSION(..) :: outbuf	39
TYPE(MPI_Datatype), INTENT(IN) :: datatype	40
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	41
	42
Fortran binding	43
MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT,	44
DATATYPE, IERROR)	45
CHARACTER*(*) DATAREP	46
<type> INBUF(*), OUTBUF(*)	47
INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION	48
INTEGER OUTCOUNT, DATATYPE, IERROR	

```

1 MPI_PACK_EXTERNAL_SIZE(datarep, incount, datatype, size)
2   IN      datarep          data representation (string)
3
4   IN      incount         number of input data items (integer)
5
6   IN      datatype        datatype of each input data item (handle)
7
8   OUT     size            output buffer size, in bytes (integer)

```

C binding

```

9 int MPI_Pack_external_size(const char datarep[], int incount,
10                          MPI_Datatype datatype, MPI_Aint *size)
11
12 int MPI_Pack_external_size_c(const char datarep[], MPI_Count incount,
13                             MPI_Datatype datatype, MPI_Count *size)
14

```

Fortran 2008 binding

```

15 MPI_Pack_external_size(datarep, incount, datatype, size, ierror)
16   CHARACTER(LEN=*), INTENT(IN) :: datarep
17   INTEGER, INTENT(IN) :: incount
18   TYPE(MPI_Datatype), INTENT(IN) :: datatype
19   INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size
20   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22 MPI_Pack_external_size(datarep, incount, datatype, size, ierror) !(_c)
23   CHARACTER(LEN=*), INTENT(IN) :: datarep
24   INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: incount
25   TYPE(MPI_Datatype), INTENT(IN) :: datatype
26   INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
27   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28

```

Fortran binding

```

29 MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR)
30   CHARACTER*(*) DATAREP
31   INTEGER INCOUNT, DATATYPE, IERROR
32   INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

```


Chapter 6

Collective Communication

6.1 Introduction and Overview

Collective communication is defined as communication that involves a group or groups of processes. The functions of this type provided by MPI are the following:

- `MPI_BARRIER`, `MPI_IBARRIER`, `MPI_BARRIER_INIT`: Barrier synchronization across all members of a group (Section 6.3, Section 6.12.1, and Section 6.13.1).
- `MPI_BCAST`, `MPI_IBCAST`, `MPI_BCAST_INIT`: Broadcast from one member to all members of a group (Section 6.4, Section 6.12.2, and Section 6.13.2). This is shown as “broadcast” in Figure 6.1.
- `MPI_GATHER`, `MPI_IGATHER`, `MPI_GATHER_INIT`, `MPI_GATHERV`, `MPI_IGATHERV`, `MPI_GATHERV_INIT`, : Gather data from all members of a group to one member (Section 6.5, Section 6.12.3, and Section 6.13.3). This is shown as “gather” in Figure 6.1.
- `MPI_SCATTER`, `MPI_ISCATTER`, `MPI_SCATTER_INIT`, `MPI_SCATTERV`, `MPI_ISCATTERV`, `MPI_SCATTERV_INIT`: Scatter data from one member to all members of a group (Section 6.6, Section 6.12.4, and Section 6.13.4). This is shown as “scatter” in Figure 6.1.
- `MPI_ALLGATHER`, `MPI_IALLGATHER`, `MPI_ALLGATHER_INIT`, `MPI_ALLGATHERV`, `MPI_IALLGATHERV`, `MPI_ALLGATHERV_INIT`: A variation on Gather where all members of a group receive the result (Section 6.7, Section 6.12.5, and Section 6.13.5). This is shown as “allgather” in Figure 6.1.
- `MPI_ALLTOALL`, `MPI_IALLTOALL`, `MPI_ALLTOALL_INIT`, `MPI_ALLTOALLV`, `MPI_IALLTOALLV`, `MPI_ALLTOALLV_INIT`, `MPI_ALLTOALLW`, `MPI_IALLTOALLW`, `MPI_ALLTOALLW_INIT`: Scatter/Gather data from all members to all members of a group (also called complete exchange) (Section 6.8, Section 6.12.6, and Section 6.13.6). This is shown as “complete exchange” in Figure 6.1.
- `MPI_ALLREDUCE`, `MPI_IALLREDUCE`, `MPI_ALLREDUCE_INIT`, `MPI_REDUCE`, `MPI_IREDUCE`, `MPI_REDUCE_INIT`: Global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to all members of a group (Section 6.9.6, Section 6.12.8, and Section 6.13.8) and a variation where the result is returned to only one member (Section 6.9, Section 6.12.7, and Section 6.13.7).
- `MPI_REDUCE_SCATTER_BLOCK`, `MPI_IREDUCE_SCATTER_BLOCK`, `MPI_REDUCE_SCATTER_BLOCK_INIT`, `MPI_REDUCE_SCATTER`,

1 MPI_IREDUCE_SCATTER, MPI_REDUCE_SCATTER_INIT: A combined reduction and
2 scatter operation (Section 6.10, Section 6.12.9, Section 6.12.10, Section 6.13.9, and
3 Section 6.13.10).

- 4 • MPI_SCAN, MPI_ISCAN, MPI_SCAN_INIT, MPI_EXSCAN, MPI_IEXSCAN,
5 MPI_EXSCAN_INIT: Scan across all members of a group (also called prefix) (Sec-
6 tion 6.11, Section 6.11.2, Section 6.12.11, Section 6.12.12, Section 6.13.11, and Sec-
7 tion 6.13.12).

9 One of the key arguments in a call to a collective routine is a communicator that
10 defines the group or groups of participating processes and provides a context for the oper-
11 ation. This is discussed further in Section 6.2. The syntax and semantics of the collective
12 operations are defined to be consistent with the syntax and semantics of the point-to-point
13 operations. Thus, general datatypes are allowed and must match between sending and re-
14 ceiving processes as specified in Chapter 5. Several collective routines such as broadcast
15 and gather have a single originating or receiving process. Such a process is called the **root**.
16 Some arguments in the collective functions are specified as “significant only at root,” and
17 are ignored for all participants except the root. The reader is referred to Chapter 5 for
18 information concerning communication buffers, general datatypes and type matching rules,
19 and to Chapter 7 for information on how to define groups and create communicators.

20 The type-matching conditions for the collective operations are more strict than the cor-
21 responding conditions between sender and receiver in point-to-point. Namely, for collective
22 operations, the amount of data sent must exactly match the amount of data specified by
23 the receiver. Different type maps (the layout in memory, see Section 5.1) between sender
24 and receiver are still allowed.

25 Collective operations can (but are not required to) complete as soon as the caller’s
26 participation in the collective communication is finished. A blocking operation is complete
27 as soon as the call returns. A nonblocking (immediate) call requires a separate completion
28 call (cf. Section 3.7). The completion of a collective operation indicates that the caller is free
29 to modify locations in the communication buffer. It does not indicate that other processes
30 in the group have completed or even started the operation (unless otherwise implied by the
31 description of the operation). Thus, a collective communication operation may, or may not,
32 have the effect of synchronizing all participating MPI processes.

33 Collective communication calls may use the same communicators as point-to-point
34 communication; MPI guarantees that messages generated on behalf of collective communi-
35 cation calls will not be confused with messages generated by point-to-point communication.
36 The collective operations do not have a message tag argument. A more detailed discussion
37 of correct use of collective routines is found in Section 6.14.

38
39 *Rationale.* The equal-data restriction (on type matching) was made so as to avoid
40 the complexity of providing a facility analogous to the status argument of MPI_RECV
41 for discovering the amount of data sent. Some of the collective routines would require
42 an array of status values.

43 The statements about synchronization are made so as to allow a variety of implemen-
44 tations of the collective functions.

45 (*End of rationale.*)

46
47 *Advice to users.* It is dangerous to rely on synchronization side-effects of the col-
48 lective operations for program correctness. For example, even though a particular

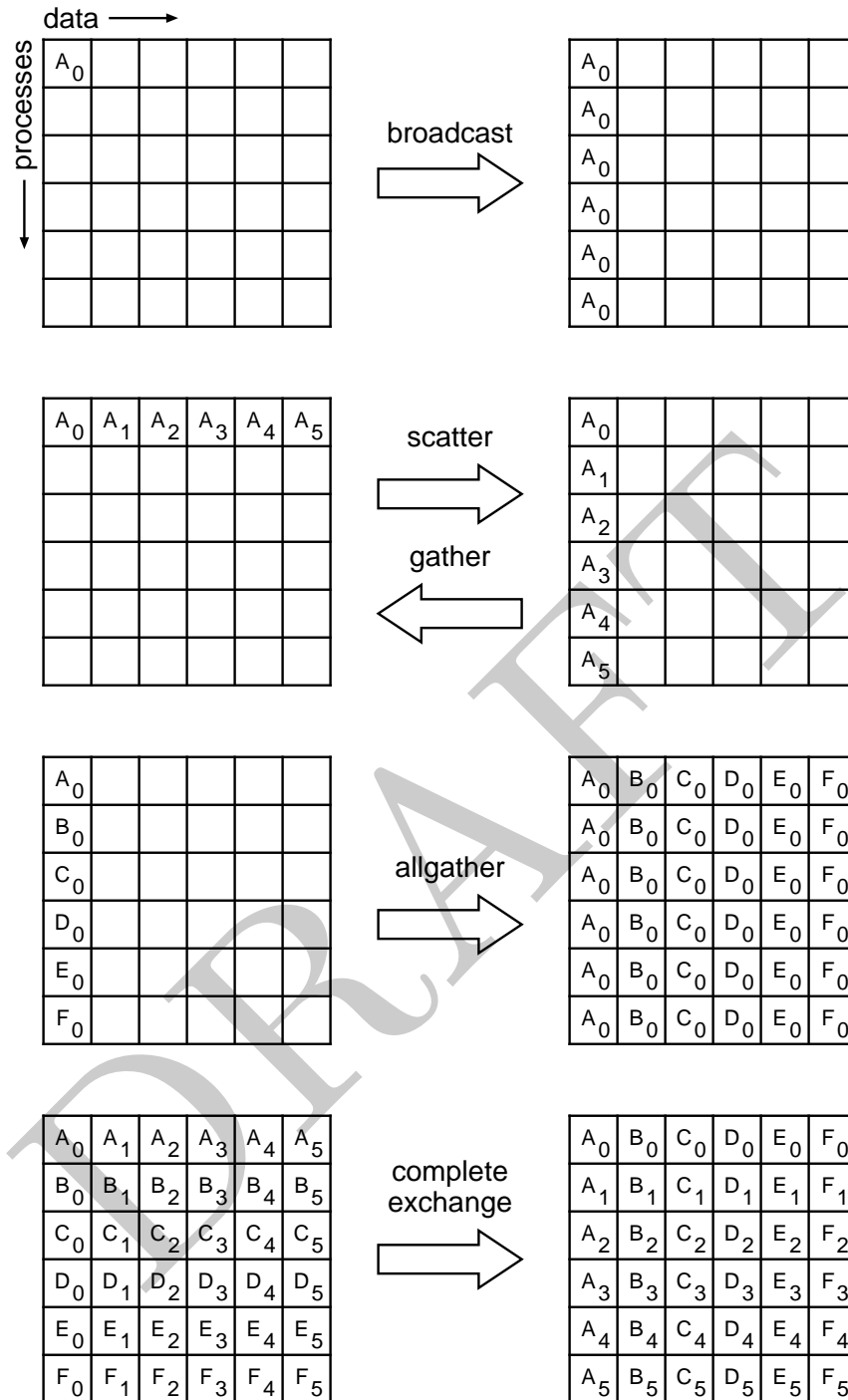


Figure 6.1: Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially just the first process contains the data A_0 , but after the broadcast all processes contain it.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

1 implementation may provide a broadcast routine with a side-effect of synchroniza-
2 tion, the standard does not require this, and a program that relies on this will not be
3 portable.

4 On the other hand, a correct, portable program must allow for the fact that a collective
5 call *may* be synchronizing. Though one cannot rely on any synchronization side-effect,
6 one must program so as to allow it. These issues are discussed further in Section 6.14.
7 (*End of advice to users.*)
8

9 *Advice to implementors.* While vendors may write optimized collective routines
10 matched to their architectures, a complete library of the collective communication
11 routines can be written entirely using the MPI point-to-point communication func-
12 tions and a few auxiliary functions. If implementing on top of point-to-point, a hidden,
13 special communicator might be created for the collective operation so as to avoid inter-
14 ference with any on-going point-to-point communication at the time of the collective
15 call. This is discussed further in Section 6.14. (*End of advice to implementors.*)
16

17 Many of the descriptions of the collective routines provide illustrations in terms of
18 blocking MPI point-to-point routines. These are intended solely to indicate what data is
19 sent or received by what process. Many of these examples are *not* correct MPI programs;
20 for purposes of simplicity, they often assume infinite buffering.
21

22 6.2 Communicator Argument

23 The key concept of the collective functions is to have a group or groups of participating
24 processes. The routines do not have group identifiers as explicit arguments. Instead, there
25 is a communicator argument. Groups and communicators are discussed in full detail in
26 Chapter 7. For the purposes of this chapter, it is sufficient to know that there are two
27 types of communicators: **intra-communicators** and **inter-communicators**. An intra-
28 communicator can be thought of as an identifier for a single group of processes linked with
29 a context. An inter-communicator identifies two distinct groups of processes linked with a
30 context.
31

32 6.2.1 Specifics for Intra-Communicator Collective Operations

33 All processes in the group identified by the intra-communicator must call the collective
34 routine.
35

36 In many cases, collective communication can occur “in place” for intra-communicators,
37 with the output buffer being identical to the input buffer. This is specified by providing
38 a special argument value, `MPI_IN_PLACE`, instead of the send buffer or the receive buffer
39 argument, depending on the operation performed.
40

41 *Rationale.* The “in place” operations are provided to reduce unnecessary memory
42 motion by both the MPI implementation and by the user. Note that while the simple
43 check of testing whether the send and receive buffers have the same address will
44 work for some cases (e.g., `MPI_ALLREDUCE`), they are inadequate in others (e.g.,
45 `MPI_GATHER`, with root not equal to zero). Further, Fortran explicitly prohibits
46 aliasing of arguments; the approach of using a special value to denote “in place”
47 operation eliminates that difficulty. (*End of rationale.*)
48

Advice to users. By allowing the “in place” option, the receive buffer in many of the collective calls becomes a send-and-receive buffer. For this reason, a Fortran binding that includes INTENT must mark these as INOUT, not OUT.

Note that MPI_IN_PLACE is a special kind of value; it has the same restrictions on its use that MPI_BOTTOM has (not usable in Fortran for initialization or assignment). See Section 2.5.4. (*End of advice to users.*)

6.2.2 Applying Collective Operations to Inter-Communicators

To understand how collective operations apply to inter-communicators, we can view most MPI intra-communicator collective operations as fitting one of the following categories (see, for instance, [63]):

All-To-All: All processes contribute to the result. All processes receive the result.

- MPI_ALLGATHER, MPI_IALLGATHER, MPI_ALLGATHER_INIT, MPI_ALLGATHERV, MPI_IALLGATHERV, MPI_ALLGATHERV_INIT
- MPI_ALLTOALL, MPI_IALLTOALL, MPI_ALLTOALL_INIT, MPI_ALLTOALLV, MPI_IALLTOALLV, MPI_ALLTOALLV_INIT, MPI_ALLTOALLW, MPI_IALLTOALLW, MPI_ALLTOALLW_INIT
- MPI_ALLREDUCE, MPI_IALLREDUCE, MPI_ALLREDUCE_INIT, MPI_REDUCE_SCATTER_BLOCK, MPI_IREDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER_BLOCK_INIT, MPI_REDUCE_SCATTER, MPI_IREDUCE_SCATTER, MPI_REDUCE_SCATTER_INIT
- MPI_BARRIER, MPI_IBARRIER, MPI_BARRIER_INIT

All-To-One: All processes contribute to the result. One process receives the result.

- MPI_GATHER, MPI_IGATHER, MPI_GATHER_INIT, MPI_GATHERV, MPI_IGATHERV, MPI_GATHERV_INIT
- MPI_REDUCE, MPI_IREDUCE, MPI_REDUCE_INIT,

One-To-All: One process contributes to the result. All processes receive the result.

- MPI_BCAST, MPI_IBCAST, MPI_BCAST_INIT
- MPI_SCATTER, MPI_ISCATTER, MPI_SCATTER_INIT, MPI_SCATTERV, MPI_ISCATTERV, MPI_SCATTERV_INIT

Other: Collective operations that do not fit into one of the above categories.

- MPI_SCAN, MPI_ISCAN, MPI_SCAN_INIT MPI_EXSCAN, MPI_IEXSCAN, MPI_EXSCAN_INIT

The data movement patterns of MPI_SCAN, MPI_ISCAN, MPI_EXSCAN, and MPI_IEXSCAN do not fit this taxonomy.

The application of collective communication to inter-communicators is best described in terms of two groups. For example, an all-to-all MPI_ALLGATHER operation can be described as collecting data from all members of one group with the result appearing in all members of the other group (see Figure 6.2). As another example, a one-to-all 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 6.3). For intra-communicators, these two groups are the same. For inter-communicators, 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 inter-communicators:

- MPI_BARRIER, MPI_IBARRIER, MPI_BARRIER_INIT,
- MPI_BCAST, MPI_IBCAST, MPI_BCAST_INIT,
- MPI_GATHER, MPI_IGATHER, MPI_GATHER_INIT, MPI_GATHERV,
MPI_IGATHERV, MPI_GATHERV_INIT,
- MPI_SCATTER, MPI_ISCATTER, MPI_SCATTER_INIT, MPI_SCATTERV,
MPI_ISCATTERV, MPI_SCATTERV_INIT,
- MPI_ALLGATHER, MPI_IALLGATHER, MPI_ALLGATHER_INIT, MPI_ALLGATHERV,
MPI_IALLGATHERV, MPI_ALLGATHERV_INIT,
- MPI_ALLTOALL, MPI_IALLTOALL, MPI_ALLTOALL_INIT, MPI_ALLTOALLV,
MPI_IALLTOALLV, MPI_ALLTOALLV_INIT, MPI_ALLTOALLW, MPI_IALLTOALLW,
MPI_ALLTOALLW_INIT,
- MPI_ALLREDUCE, MPI_IALLREDUCE, MPI_ALLREDUCE_INIT, MPI_REDUCE,
MPI_IREDUCE, MPI_REDUCE_INIT,
- MPI_REDUCE_SCATTER_BLOCK, MPI_IREDUCE_SCATTER_BLOCK,
MPI_REDUCE_SCATTER_BLOCK_INIT, MPI_REDUCE_SCATTER,
MPI_IREDUCE_SCATTER, MPI_REDUCE_SCATTER_INIT.

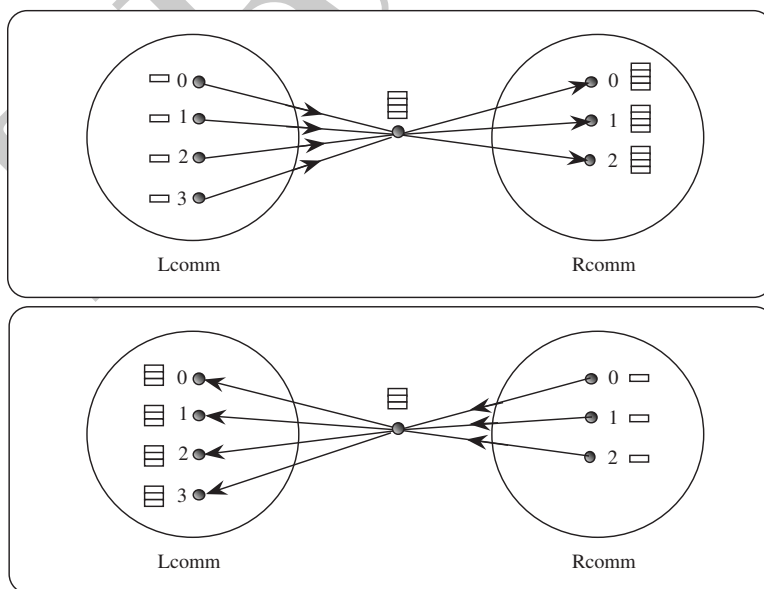


Figure 6.2: Inter-communicator allgather. The focus of data to one process is represented, not mandated by the semantics. The two phases do allgathers in both directions.

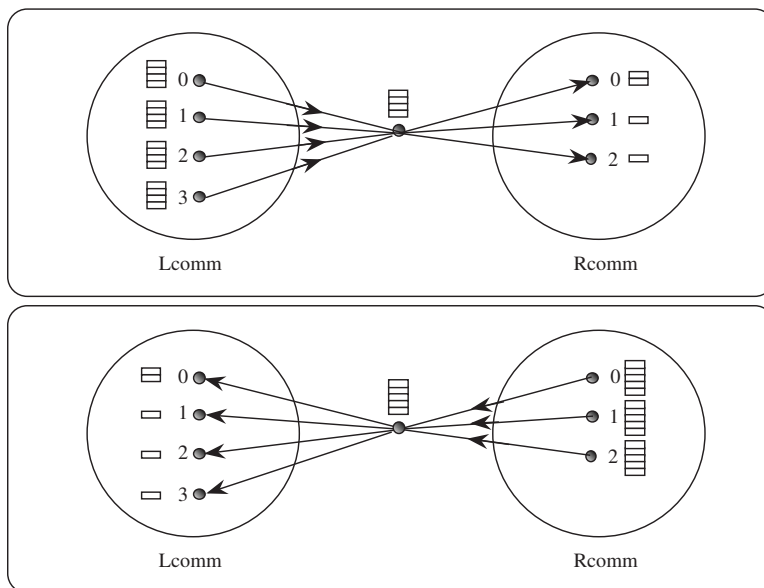


Figure 6.3: Inter-communicator reduce-scatter. The focus of data to one process is represented, not mandated by the semantics. The two phases do reduce-scatters in both directions.

6.2.3 Specifics for Inter-Communicator Collective Operations

All processes in both groups identified by the inter-communicator must call the collective routine.

Note that the “in place” option for intra-communicators does not apply to inter-communicators since in the inter-communicator case there is no communication from a process to itself.

For inter-communicator 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 category will often occur as part of an exchange, where it makes sense to communicate in both directions at once. (*End of rationale.*)

6.3 Barrier Synchronization

`MPI_BARRIER(comm)`

1 IN comm communicator (handle)

2

3

C binding

4

```
int MPI_Barrier(MPI_Comm comm)
```

5

6

Fortran 2008 binding

7

```
MPI_Barrier(comm, ierror)
```

8

```
    TYPE(MPI_Comm), INTENT(IN) :: comm
```

9

```
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

10

11

Fortran binding

12

```
MPI_BARRIER(COMM, IERROR)
```

13

```
    INTEGER COMM, IERROR
```

14

If `comm` is an intra-communicator, `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.

15

16

17

If `comm` is an inter-communicator, `MPI_BARRIER` involves two groups. The call returns at processes in one group (group A) of the inter-communicator 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.

18

19

20

21

22

6.4 Broadcast

23

24

25

26

```
MPI_BCAST(buffer, count, datatype, root, comm)
```

27

```
    INOUT   buffer                           starting address of buffer (choice)
```

28

```
    IN       count                           number of entries in buffer (non-negative integer)
```

29

30

```
    IN       datatype                       datatype of buffer (handle)
```

31

32

```
    IN       root                            rank of broadcast root (integer)
```

33

```
    IN       comm                           communicator (handle)
```

34

35

C binding

36

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root,
```

37

```
          MPI_Comm comm)
```

38

39

```
int MPI_Bcast_c(void *buffer, MPI_Count count, MPI_Datatype datatype, int root,
```

40

```
          MPI_Comm comm)
```

41

42

Fortran 2008 binding

43

```
MPI_Bcast(buffer, count, datatype, root, comm, ierror)
```

44

```
    TYPE(*), DIMENSION(..) :: buffer
```

45

```
    INTEGER, INTENT(IN) :: count, root
```

46

```
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
```

47

```
    TYPE(MPI_Comm), INTENT(IN) :: comm
```

48

```
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```



```

MPI_Bcast(buffer, count, datatype, root, comm, ierror) !(_c)
  TYPE(*), DIMENSION(..) :: buffer
  INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  INTEGER, INTENT(IN) :: root
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
  <type> BUFFER(*)
  INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR

```

If `comm` is an intra-communicator, `MPI_BCAST` broadcasts a message from the process with rank `root` to all processes of the group, itself included. It is called by all members of the group using the same arguments for `comm` and `root`. On return, the content of `root`'s buffer is copied to all other processes.

General, derived datatypes are allowed for `datatype`. The type signature of `count`, `datatype` on any process must be equal to the type signature of `count`, `datatype` at the root. This implies that the amount of data sent must be equal to the amount received, pairwise between each process and the root. `MPI_BCAST` and all other data-movement collective routines make this restriction. Distinct type maps between sender and receiver are still allowed.

The “in place” option is not meaningful here.

If `comm` is an inter-communicator, then the call involves all processes in the inter-communicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument `root`, which is the rank of the root in group A. The root passes the value `MPI_ROOT` in `root`. All other processes in group A pass the value `MPI_PROC_NULL` in `root`. Data is broadcast from the root to all processes in group B. The buffer arguments of the processes in group B must be consistent with the buffer argument of the root.

6.4.1 Example using `MPI_BCAST`

The examples in this section use intra-communicators.

Example 6.1. Broadcast 100 ints from process 0 to every process in the group.

```

MPI_Comm comm;
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.

6.5 Gather

```

1 MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
2
3
4
5 MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
6     IN     sendbuf     starting address of send buffer (choice)
7     IN     sendcount   number of elements in send buffer (non-negative
8                          integer)
9
10    IN     sendtype    datatype of send buffer elements (handle)
11    OUT    recvbuf     address of receive buffer (choice, significant only at
12                          root)
13    IN     recvcount   number of elements for any single receive
14                          (non-negative integer, significant only at root)
15
16    IN     recvtype    datatype of recv buffer elements (handle, significant
17                          only at root)
18
19    IN     root        rank of receiving process (integer)
20
21    IN     comm        communicator (handle)

```

C binding

```

22 int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
23               void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
24               MPI_Comm comm)
25

```

```

26 int MPI_Gather_c(const void *sendbuf, MPI_Count sendcount,
27                 MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
28                 MPI_Datatype recvtype, int root, MPI_Comm comm)
29

```

Fortran 2008 binding

```

30 MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root,
31           comm, ierror)
32

```

```

33     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
34     INTEGER, INTENT(IN) :: sendcount, recvcount, root
35     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
36     TYPE(*), DIMENSION(..) :: recvbuf
37     TYPE(MPI_Comm), INTENT(IN) :: comm
38     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

39 MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root,
40           comm, ierror) !(_c)
41

```

```

42     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
43     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
44     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
45     TYPE(*), DIMENSION(..) :: recvbuf
46     INTEGER, INTENT(IN) :: root
47     TYPE(MPI_Comm), INTENT(IN) :: comm
48     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, REVCOUNT, RECVTYPE, ROOT,
           COMM, IERROR)
```

```
<type> SENDBUF(*), RECVBUF(*)
```

```
INTEGER SENDCOUNT, SENDTYPE, REVCOUNT, RECVTYPE, ROOT, COMM, IERROR
```

If `comm` is an intra-communicator, 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 nonroot 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 intra-communicators 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 inter-communicator, then the call involves all processes in the inter-communicator, 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, recvcnts, displs, recvtype, root,
           comm)
```

1	IN	sendbuf	starting address of send buffer (choice)
2	IN	sendcount	number of elements in send buffer (non-negative integer)
3			
4			
5	IN	sendtype	datatype of send buffer elements (handle)
6	OUT	recvbuf	address of receive buffer (choice, significant only at root)
7			
8	IN	recvcounts	nonnegative integer array (of length group size) containing the number of elements that are received from each process (significant only at root)
9			
10			
11			
12	IN	displs	integer array (of length group size). Entry <i>i</i> specifies the displacement relative to <code>recvbuf</code> at which to place the incoming data from process <i>i</i> (significant only at root)
13			
14			
15			
16	IN	recvtype	datatype of recv buffer elements (handle, significant only at root)
17			
18	IN	root	rank of receiving process (integer)
19			
20	IN	comm	communicator (handle)

C binding

```

23 int MPI_Gatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
24               void *recvbuf, const int recvcounts[], const int displs[],
25               MPI_Datatype recvtype, int root, MPI_Comm comm)
26
27 int MPI_Gatherv_c(const void *sendbuf, MPI_Count sendcount,
28                 MPI_Datatype sendtype, void *recvbuf,
29                 const MPI_Count recvcounts[], const MPI_Aint displs[],
30                 MPI_Datatype recvtype, int root, MPI_Comm comm)

```

Fortran 2008 binding

```

32 MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
33            recvtype, root, comm, ierror)
34     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
35     INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*), root
36     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
37     TYPE(*), DIMENSION(..) :: recvbuf
38     TYPE(MPI_Comm), INTENT(IN) :: comm
39     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41 MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
42            recvtype, root, comm, ierror) !(_c)
43     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
44     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcounts(*)
45     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
46     TYPE(*), DIMENSION(..) :: recvbuf
47     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
48     INTEGER, INTENT(IN) :: root

```

```

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

```

Fortran 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 intra-communicator, 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 nonroot processes.

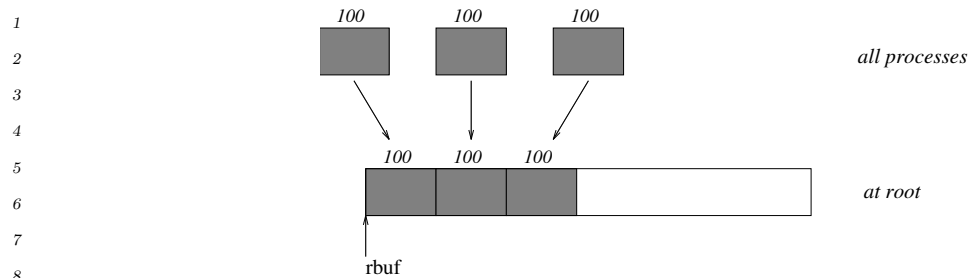
The type signature implied by `sendcount`, `sendtype` on process `i` must be equal to the type signature implied by `recvcounts[i]`, `recvtype` at the root. This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed, as illustrated in Example 6.6.

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 intra-communicators 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 inter-communicator, then the call involves all processes in the inter-communicator, 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.



10 Figure 6.4: The root process gathers 100 ints from each process in the group.

12 6.5.1 Examples using MPI_GATHER, MPI_GATHERV

13 The examples in this section use intra-communicators.

14 **Example 6.2.** Gather 100 ints from every process in group to root. See Figure 6.4.

```

15 MPI_Comm comm;
16 int gsize, sendarray[100];
17 int root, *rbuf;
18 ...
19 MPI_Comm_size(comm, &gsize);
20 rbuf = (int *)malloc(gsize*100*sizeof(int));
21 MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
22
23

```

24 **Example 6.3.** Previous example modified—only the root allocates memory for the receive buffer.

```

25 MPI_Comm comm;
26 int gsize, sendarray[100];
27 int root, myrank, *rbuf;
28 ...
29 MPI_Comm_rank(comm, &myrank);
30 if (myrank == root) {
31     MPI_Comm_size(comm, &gsize);
32     rbuf = (int *)malloc(gsize*100*sizeof(int));
33 }
34 MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
35
36

```

37 **Example 6.4.** Do the same as the previous example, but use a derived datatype. Note that the type cannot be the entire set of $gsize*100$ ints since type matching is defined pairwise between the root and each process in the gather.

```

38 MPI_Comm comm;
39 int gsize, sendarray[100];
40 int root, *rbuf;
41 MPI_Datatype rtype;
42 ...
43 MPI_Comm_size(comm, &gsize);
44 MPI_Type_contiguous(100, MPI_INT, &rtype);
45
46

```

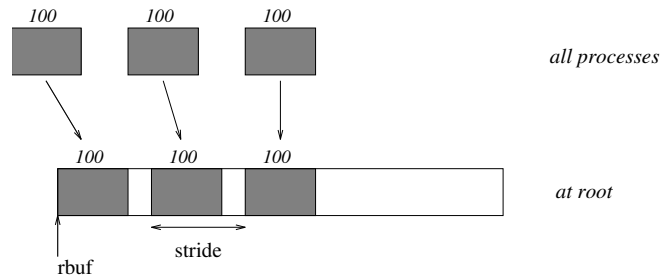


Figure 6.5: The root process gathers 100 ints from each process in the group, each set is placed *stride* ints apart.

```
MPI_Type_commit(&rtype);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather(sendarray, 100, MPI_INT, rbuf, 1, rtype, root, comm);
```

Example 6.5. Now have each process send 100 ints to root, but place each set (of 100) *stride* ints apart at receiving end. Use `MPI_GATHERV` and the `displs` argument to achieve this effect. Assume $stride \geq 100$. See Figure 6.5.

```
MPI_Comm comm;
int gsize, sendarray[100];
int root, *rbuf, stride;
int *displs, i, *rcounts;
...

MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    rcounts[i] = 100;
}
MPI_Gatherv(sendarray, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT,
            root, comm);
```

Note that the program is erroneous if $stride < 100$.

Example 6.6. Same as Example 6.5 on the receiving side, but send the 100 ints from the 0th column of a 100×150 int array, in C. See Figure 6.6.

```
MPI_Comm comm;
int gsize, sendarray[100][150];
int root, *rbuf, stride;
MPI_Datatype stype;
int *displs, i, *rcounts;
...

```

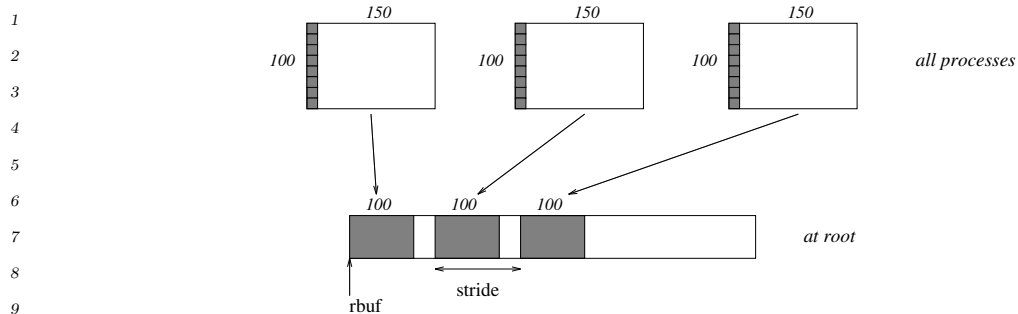


Figure 6.6: The root process gathers column 0 of a 100×150 C array, and each set is placed stride ints apart.

```

15 MPI_Comm_size(comm, &gsize);
16 rbuf = (int *)malloc(gsize*stride*sizeof(int));
17 displs = (int *)malloc(gsize*sizeof(int));
18 rcounts = (int *)malloc(gsize*sizeof(int));
19 for (i=0; i<gsize; ++i) {
20     displs[i] = i*stride;
21     rcounts[i] = 100;
22 }
23 /* Create datatype for 1 column of array
24 */
25 MPI_Type_vector(100, 1, 150, MPI_INT, &stype);
26 MPI_Type_commit(&stype);
27 MPI_Gatherv(sendarray, 1, stype, rbuf, rcounts, displs, MPI_INT,
28             root, comm);

```

Example 6.7. 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 6.7.

```

33 MPI_Comm comm;
34 int gsize, sendarray[100][150], *sptr;
35 int root, *rbuf, stride, myrank;
36 MPI_Datatype stype;
37 int *displs, i, *rcounts;
38 ...
39
40 MPI_Comm_size(comm, &gsize);
41 MPI_Comm_rank(comm, &myrank);
42 rbuf = (int *)malloc(gsize*stride*sizeof(int));
43 displs = (int *)malloc(gsize*sizeof(int));
44 rcounts = (int *)malloc(gsize*sizeof(int));
45 for (i=0; i<gsize; ++i) {
46     displs[i] = i*stride;
47     rcounts[i] = 100-i;    /* note change from previous example */
48 }

```

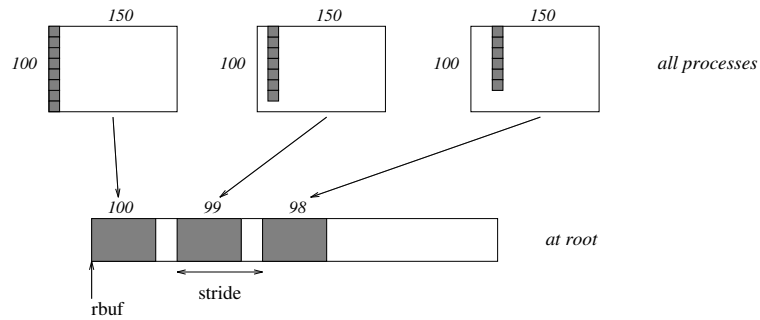



Figure 6.7: The root process gathers $100-i$ ints from column i of a 100×150 C array, and each set is placed stride ints apart.

```

/* Create datatype for the column we are sending
 */
MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &stype);
MPI_Type_commit(&stype);
/* sptr is the address of start of "myrank" column
 */
sptr = &sendarray[0][myrank];
MPI_Gatherv(sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
            root, comm);

```

Note that a different amount of data is received from each process.

Example 6.8. Same as Example 6.7, but done in a different way at the sending end. We create a datatype that causes the correct striding at the sending end so that we read a column of a C array. A similar thing was done in Example 5.16, Section 5.1.14.

```

MPI_Comm comm;
int gsize, sendarray[100][150], *sptr;
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];

```

```

1 MPI_Gatherv(sptr, 100-myrank, stype, rbuf, rcounts, displs, MPI_INT,
2           root, comm);
3
4

```

Example 6.9. Same as Example 6.7 at sending side, but at receiving side we make the stride between received blocks vary from block to block. See Figure 6.8.

```

5 MPI_Comm comm;
6 int gsize, sendarray[100][150], *sptr;
7 int root, *rbuf, *stride, myrank, bufsize;
8 MPI_Datatype stype;
9 int *displs, i, *rcounts, offset;
10
11 ...
12
13 MPI_Comm_size(comm, &gsize);
14 MPI_Comm_rank(comm, &myrank);
15
16 stride = (int *)malloc(gsize*sizeof(int));
17 ...
18 /* stride[i] for i = 0 to gsize-1 is set somehow
19 */
20
21 /* set up displs and rcounts vectors first
22 */
23 displs = (int *)malloc(gsize*sizeof(int));
24 rcounts = (int *)malloc(gsize*sizeof(int));
25 offset = 0;
26 for (i=0; i<gsize; ++i) {
27     displs[i] = offset;
28     offset += stride[i];
29     rcounts[i] = 100-i;
30 }
31 /* the required buffer size for rbuf is now easily obtained
32 */
33 bufsize = displs[gsize-1]+rcounts[gsize-1];
34 rbuf = (int *)malloc(bufsize*sizeof(int));
35 /* Create datatype for the column we are sending
36 */
37 MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &stype);
38 MPI_Type_commit(&stype);
39 sptr = &sendarray[0][myrank];
40 MPI_Gatherv(sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
41           root, comm);
42
43
44
45
46
47 MPI_Comm comm;
48

```

Example 6.10. Process i sends num ints from the i -th column of a 100×150 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.

```

47 MPI_Comm comm;
48

```

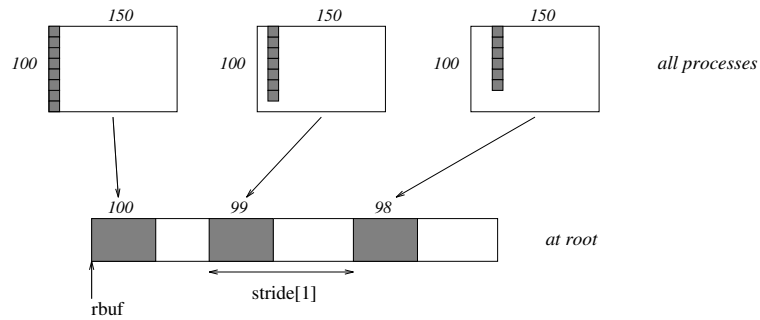


Figure 6.8: The root process gathers $100-i$ ints from column i of a 100×150 C array, and each set is placed $\text{stride}[i]$ ints apart (a varying stride).

```

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);

```

6.6 Scatter

```

1 MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
2
3
4
5 MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
6     IN      sendbuf      address of send buffer (choice, significant only at
7                          root)
8     IN      sendcount    number of elements sent to each process
9                          (non-negative integer, significant only at root)
10
11    IN      sendtype      datatype of send buffer elements (handle, significant
12                          only at root)
13
14    OUT     recvbuf       address of receive buffer (choice)
15
16    IN      recvcount     number of elements in receive buffer (non-negative
17                          integer)
18
19    IN      recvtype      datatype of receive buffer elements (handle)
20
21    IN      root          rank of sending process (integer)
22
23    IN      comm          communicator (handle)

```

C binding

```

24 int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
25                void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
26                MPI_Comm comm)
27
28 int MPI_Scatter_c(const void *sendbuf, MPI_Count sendcount,
29                  MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
30                  MPI_Datatype recvtype, int root, MPI_Comm comm)

```

Fortran 2008 binding

```

31 MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root,
32            comm, ierror)
33     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
34     INTEGER, INTENT(IN) :: sendcount, recvcount, root
35     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
36     TYPE(*), DIMENSION(..) :: recvbuf
37     TYPE(MPI_Comm), INTENT(IN) :: comm
38     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40 MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root,
41            comm, ierror) !(_c)
42     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
43     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
44     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
45     TYPE(*), DIMENSION(..) :: recvbuf
46     INTEGER, INTENT(IN) :: root
47     TYPE(MPI_Comm), INTENT(IN) :: comm
48     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, REVCOUNT, RECVTYPE, ROOT,
            COMM, IERROR)
```

```
<type> SENDBUF(*), RECVBUF(*)
```

```
INTEGER SENDCOUNT, SENDTYPE, REVCOUNT, RECVTYPE, ROOT, COMM, IERROR
```

MPI_SCATTER is the inverse operation to MPI_GATHER.

If `comm` is an intra-communicator, 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 nonroot 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 intra-communicators is specified by passing `MPI_IN_PLACE` as the value of `recvbuf` at the root. In such a case, `recvcount` and `recvtype` are ignored, and root “sends” no data to itself. The scattered vector is still assumed to contain n segments, where n is the group size; the `root`-th segment, which root should “send to itself,” is not moved.

If `comm` is an inter-communicator, then the call involves all processes in the inter-communicator, 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.

```

1 MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root,
2             comm)
3
4 IN          sendbuf          address of send buffer (choice, significant only at
5                               root)
6
7 IN          sendcounts       nonnegative integer array (of length group size)
8                               specifying the number of elements to send to each
9                               rank (significant only at root)
10
11 IN         displs           integer array (of length group size). Entry i specifies
12                               the displacement (relative to sendbuf) from which to
13                               take the outgoing data to process i (significant only
14                               at root)
15
16 IN         sendtype         datatype of send buffer elements. (handle, significant
17                               only at root)
18
19 OUT        recvbuf          address of receive buffer (choice)
20
21 IN         recvcount         number of elements in receive buffer (non-negative
22                               integer)
23
24 IN         recvtype         datatype of receive buffer elements (handle)
25
26 IN         root             rank of sending process (integer)
27
28 IN         comm             communicator (handle)

```

C binding

```

29 int MPI_Scatterv(const void *sendbuf, const int sendcounts[],
30                const int displs[], MPI_Datatype sendtype, void *recvbuf,
31                int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
32
33 int MPI_Scatterv_c(const void *sendbuf, const MPI_Count sendcounts[],
34                  const MPI_Aint displs[], MPI_Datatype sendtype, void *recvbuf,
35                  MPI_Count recvcount, MPI_Datatype recvtype, int root,
36                  MPI_Comm comm)

```

Fortran 2008 binding

```

37 MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
38             recvtype, root, comm, ierror)
39
40 TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
41 INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root
42 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
43 TYPE(*), DIMENSION(..) :: recvbuf
44 TYPE(MPI_Comm), INTENT(IN) :: comm
45 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47 MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
48             recvtype, root, comm, ierror) !(_c)
49
50 TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
51 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*), recvcount
52 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
53 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype

```

```

TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: root
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran 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 intra-communicator, 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 nonroot 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 intra-communicators is specified by passing `MPI_IN_PLACE` as the value of `recvbuf` at the root. In such a case, `recvcount` and `recvtype` are ignored, and root “sends” no data to itself. The scattered vector is still assumed to contain `n` segments, where `n` is the group size; the root-th segment, which root should “send to itself,” is not moved.

If `comm` is an inter-communicator, then the call involves all processes in the inter-communicator, 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.

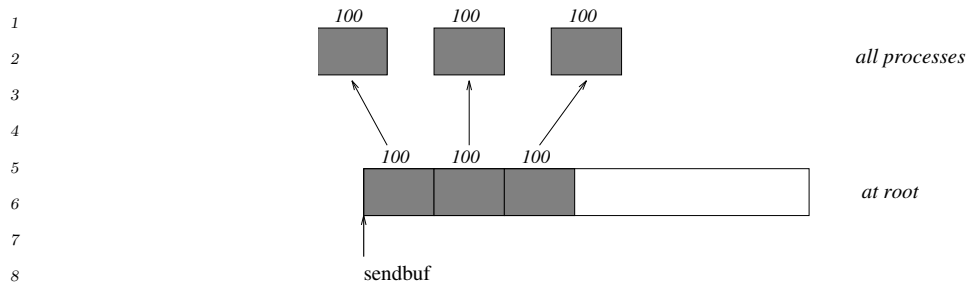


Figure 6.9: The root process scatters sets of 100 ints to each process in the group.

6.6.1 Examples using MPI_SCATTER, MPI_SCATTERV

The examples in this section use intra-communicators.

Example 6.11. The reverse of Example 6.2. Scatter sets of 100 ints from the root to each process in the group. See Figure 6.9.

```

18 MPI_Comm comm;
19 int gsize,*sendbuf;
20 int root, rbuf[100];
21 ...
22 MPI_Comm_size(comm, &gsize);
23 sendbuf = (int *)malloc(gsize*100*sizeof(int));
24 ...
25 MPI_Scatter(sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);

```

Example 6.12. The reverse of Example 6.5. The root process scatters sets of 100 ints to the other processes, but the sets of 100 are *stride ints* apart in the sending buffer. Requires use of MPI_SCATTERV. Assume $stride \geq 100$. See Figure 6.10.

```

30 MPI_Comm comm;
31 int gsize,*sendbuf;
32 int root, rbuf[100], i, *displs, *scounts;
33 ...
34 ...
35 MPI_Comm_size(comm, &gsize);
36 sendbuf = (int *)malloc(gsize*stride*sizeof(int));
37 ...
38 displs = (int *)malloc(gsize*sizeof(int));
39 scounts = (int *)malloc(gsize*sizeof(int));
40 for (i=0; i<gsize; ++i) {
41     displs[i] = i*stride;
42     scounts[i] = 100;
43 }
44 MPI_Scatterv(sendbuf, scounts, displs, MPI_INT, rbuf, 100, MPI_INT,
45             root, comm);

```

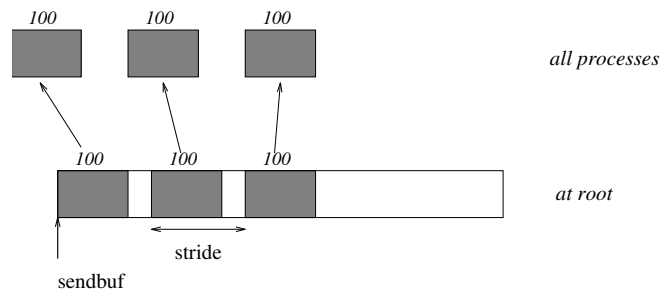



Figure 6.10: The root process scatters sets of 100 ints, moving by `stride` ints from `sendbuf` to `send` in the scatter.

Example 6.13. The reverse of Example 6.9. We have a varying stride between blocks at sending (root) side, at the receiving side we receive into the i -th column of a 100×150 C array. See Figure 6.11.

```

MPI_Comm comm;
int gsize, recvarray[100][150], *rptr;
int root, *sendbuf, myrank, *stride;
MPI_Datatype rtype;
int i, *displs, *scounts, offset;
...
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);

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));
scount = (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);

```

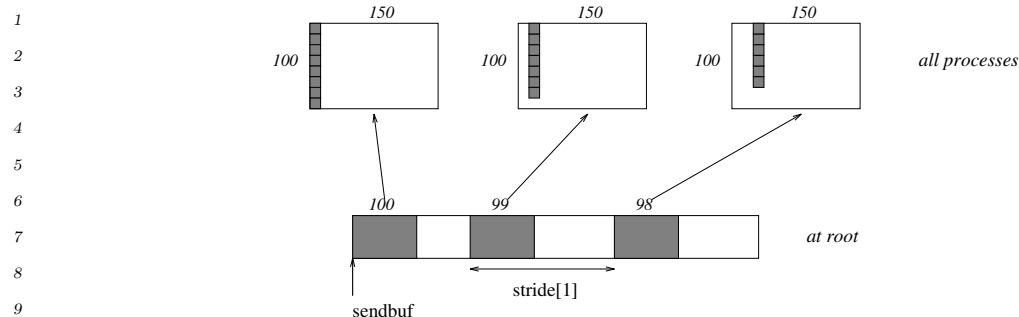


Figure 6.11: The root scatters blocks of $100-i$ ints into column i of a 100×150 C array. At the sending side, the blocks are `stride[i]` ints apart.

6.7 Gather-to-all

`MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)`

IN	<code>sendbuf</code>	starting address of send buffer (choice)
IN	<code>sendcount</code>	number of elements in send buffer (non-negative integer)
IN	<code>sendtype</code>	datatype of send buffer elements (handle)
OUT	<code>recvbuf</code>	address of receive buffer (choice)
IN	<code>recvcount</code>	number of elements received from any process (non-negative integer)
IN	<code>recvtype</code>	datatype of receive buffer elements (handle)
IN	<code>comm</code>	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)
```

```
int MPI_Allgather_c(const void *sendbuf, MPI_Count sendcount,
                   MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
                   MPI_Datatype recvtype, MPI_Comm comm)
```

Fortran 2008 binding

```
MPI_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_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm,
              ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
INTEGER(KIND=MPI_COUNT_KIND), 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

```

Fortran binding

```

MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, REVCOUNT, RECVTYPE, COMM,
              IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, REVCOUNT, 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 intra-communicator, 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 intra-communicators 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 inter-communicator, 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.*)

```

1 MPI_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcnts, displs, recvtype,
2               comm)
3
4   IN      sendbuf      starting address of send buffer (choice)
5   IN      sendcount    number of elements in send buffer (non-negative
6                       integer)
7   IN      sendtype     datatype of send buffer elements (handle)
8   OUT     recvbuf      address of receive buffer (choice)
9
10  IN      recvcnts      nonnegative integer array (of length group size)
11                       containing the number of elements that are received
12                       from each process
13  IN      displs       integer array (of length group size). Entry i specifies
14                       the displacement (relative to recvbuf) at which to
15                       place the incoming data from process i
16
17  IN      recvtype     datatype of receive buffer elements (handle)
18  IN      comm         communicator (handle)
19

```

C binding

```

20
21 int MPI_Allgatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
22                  void *recvbuf, const int recvcnts[], const int displs[],
23                  MPI_Datatype recvtype, MPI_Comm comm)
24
25 int MPI_Allgatherv_c(const void *sendbuf, MPI_Count sendcount,
26                    MPI_Datatype sendtype, void *recvbuf,
27                    const MPI_Count recvcnts[], const MPI_Aint displs[],
28                    MPI_Datatype recvtype, MPI_Comm comm)
29

```

Fortran 2008 binding

```

30 MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcnts, displs,
31               recvtype, comm, ierror)
32   TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
33   INTEGER, INTENT(IN) :: sendcount, recvcnts(*), displs(*)
34   TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
35   TYPE(*), DIMENSION(..) :: recvbuf
36   TYPE(MPI_Comm), INTENT(IN) :: comm
37   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39 MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcnts, displs,
40               recvtype, comm, ierror) !(_c)
41   TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
42   INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcnts(*)
43   TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
44   TYPE(*), DIMENSION(..) :: recvbuf
45   INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
46   TYPE(MPI_Comm), INTENT(IN) :: comm
47   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48

```

Fortran binding

```

MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVCOUNTS, DISPLS,
               RECVTYPE, COMM, IERROR)
<type> SENDBUF(*), RECVCOUNTS(*)
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 intra-communicator, 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 intra-communicators 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 inter-communicator, 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.

6.7.1 Example using MPI_ALLGATHER

The example in this section uses intra-communicators.

Example 6.14. The all-gather version of Example 6.2. 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.

6.8 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	datatype of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements received from any process (non-negative integer)
IN	recvtype	datatype of receive buffer elements (handle)
IN	comm	communicator (handle)

C binding

```
int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                void *recvbuf, int recvcount, MPI_Datatype recvtype,
                MPI_Comm comm)
```

```
int MPI_Alltoall_c(const void *sendbuf, MPI_Count sendcount,
                  MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
                  MPI_Datatype recvtype, MPI_Comm comm)
```

Fortran 2008 binding

```
MPI_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_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm,
             ierror) !(_c)
```

```
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
INTEGER(KIND=MPI_COUNT_KIND), 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
```

Fortran 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 intra-communicator, 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 intra-communicators 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 inter-communicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The j -th send buffer of process i in group A should be consistent with the i -th receive buffer of process j in group B, and vice versa.

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, recvcnts, rdispls,
               recvtype, comm)
```

```
IN      sendbuf      starting address of send buffer (choice)
```

1	IN	sendcounts	nonnegative integer array (of length group size)
2			specifying the number of elements to send to each
3			rank
4	IN	sdispls	integer array (of length group size). Entry j specifies
5			the displacement (relative to <code>sendbuf</code>) from which to
6			take the outgoing data destined for process j
7			
8	IN	sendtype	datatype of send buffer elements (handle)
9	OUT	recvbuf	address of receive buffer (choice)
10	IN	recvcounts	nonnegative integer array (of length group size)
11			specifying the number of elements that can be
12			received from each rank
13			
14	IN	rdispls	integer array (of length group size). Entry i specifies
15			the displacement (relative to <code>recvbuf</code>) at which to
16			place the incoming data from process i
17	IN	recvtype	datatype of receive buffer elements (handle)
18	IN	comm	communicator (handle)
19			

C binding

```

21 int MPI_Alltoallv(const void *sendbuf, const int sendcounts[],
22                 const int sdispls[], MPI_Datatype sendtype, void *recvbuf,
23                 const int recvcounts[], const int rdispls[],
24                 MPI_Datatype recvtype, MPI_Comm comm)
25
26 int MPI_Alltoallv_c(const void *sendbuf, const MPI_Count sendcounts[],
27                    const MPI_Aint sdispls[], MPI_Datatype sendtype, void *recvbuf,
28                    const MPI_Count recvcounts[], const MPI_Aint rdispls[],
29                    MPI_Datatype recvtype, MPI_Comm comm)

```

Fortran 2008 binding

```

31 MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
32              rdispls, recvtype, comm, ierror)
33
34   TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
35   INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*), rdispls(*)
36   TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
37   TYPE(*), DIMENSION(..) :: recvbuf
38   TYPE(MPI_Comm), INTENT(IN) :: comm
39   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41 MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
42              rdispls, recvtype, comm, ierror) !(_c)
43   TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
44   INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*), recvcounts(*)
45   INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
46   TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
47   TYPE(*), DIMENSION(..) :: recvbuf
48   TYPE(MPI_Comm), INTENT(IN) :: comm
49   INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```


Fortran 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 intra-communicator, 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 intra-communicators 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 inter-communicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The j -th send buffer of process i in group A should be consistent with the i -th receive buffer of process j in group B, and vice versa.

Rationale. The definitions of MPI_ALLTOALL and MPI_ALLTOALLV give as much flexibility as one would achieve by specifying n independent, point-to-point communications, with two exceptions: all messages use the same datatype, and messages are scattered from (or gathered to) sequential storage. (*End of rationale.*)

Advice to implementors. Although the discussion of collective communication in terms of point-to-point operation implies that each message is transferred directly from sender to receiver, implementations may use a tree communication pattern. Messages can be forwarded by intermediate nodes where they are split (for scatter) or concatenated (for gather), if this is more efficient. (*End of advice to implementors.*)

```

MPI_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcoun
ts, rdispls,
               recvtypes, comm)
IN      sendbuf      starting address of send buffer (choice)
IN      sendcounts   nonnegative 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      recvcoun
ts      nonnegative 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 recvcoun
ts[], const int rdispls[],
                 const MPI_Datatype recvtypes[], MPI_Comm comm)

```

```

int MPI_Alltoallw_c(const void *sendbuf, const MPI_Count sendcounts[],
                   const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],
                   void *recvbuf, const MPI_Count recvcoun
ts[],
                   const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],
                   MPI_Comm comm)

```

Fortran 2008 binding

```

MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcoun-
    rdispls, recvtypes, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcoun-
    TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
    TYPE(*), DIMENSION(..) :: recvbuf
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcoun-
    rdispls, recvtypes, comm, ierror) !(_c)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*), recvcoun-
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
    TYPE(*), DIMENSION(..) :: recvbuf
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECV-
    RDISPLS, RECVTYPES, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), REVCOUN-
    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 intra-communicator, 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 *recvcoun-*[*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],recvcoun- [i],recvtypes[i] ,i,...).
```

All arguments on all processes are significant. The argument *comm* must describe the same communicator on all processes.

1 Like for MPI_ALLTOALLV, the “in place” option for intra-communicators is specified by
 2 passing MPI_IN_PLACE to the argument `sendbuf` at *all* processes. In such a case, `sendcounts`,
 3 `sdispls` and `sendtypes` are ignored. The data to be sent is taken from the `recvbuf` and replaced
 4 by the received data. Data sent and received must have the same type map as specified
 5 by the `recvcounts` and `recvtypes` arrays, and is taken from the locations of the receive buffer
 6 specified by `rdispls`.

7 If `comm` is an inter-communicator, then the outcome is as if each process in group A
 8 sends a message to each process in group B, and vice versa. The *j*-th send buffer of process
 9 *i* in group A should be consistent with the *i*-th receive buffer of process *j* in group B, and
 10 vice versa.

11
 12 *Rationale.* The MPI_ALLTOALLW function generalizes several MPI functions by
 13 carefully selecting the input arguments. For example, by making all but one process
 14 have `sendcounts[i] = 0`, this achieves an MPI_SCATTERW function. (*End of rationale.*)
 15

16 6.9 Global Reduction Operations

17
 18 The functions in this section perform a global reduce operation (for example sum, maximum,
 19 and logical and) across all members of a group. The reduction operation can be either one of
 20 a predefined list of operations, or a user-defined operation. The global reduction functions
 21 come in several flavors: a reduce that returns the result of the reduction to one member of a
 22 group, an all-reduce that returns this result to all members of a group, and two scan (parallel
 23 prefix) operations. In addition, a reduce-scatter operation combines the functionality of a
 24 reduce and of a scatter operation.
 25

26 6.9.1 Reduce

27
 28
 29 MPI_REDUCE(`sendbuf`, `recvbuf`, `count`, `datatype`, `op`, `root`, `comm`)

30	IN	<code>sendbuf</code>	address of send buffer (choice)
31	OUT	<code>recvbuf</code>	address of receive buffer (choice, significant only at 32 root)
33	IN	<code>count</code>	number of elements in send buffer (non-negative 34 integer)
35	IN	<code>datatype</code>	datatype of elements of send buffer (handle)
36	IN	<code>op</code>	reduce operation (handle)
37	IN	<code>root</code>	rank of root process (integer)
38	IN	<code>comm</code>	communicator (handle)
39			
40			
41			
42			

43 C binding

```
44 int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
45               MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
46
47 int MPI_Reduce_c(const void *sendbuf, void *recvbuf, MPI_Count count,
48                 MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

Fortran 2008 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

MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror) !(_c)
  TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
  TYPE(*), DIMENSION(..) :: recvbuf
  INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Op), INTENT(IN) :: op
  INTEGER, INTENT(IN) :: root
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran 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 intra-communicator, `MPI_REDUCE` combines the elements provided in the input buffer of each process in the group, using the operation `op`, and returns the 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 6.9.2, lists the set of predefined operations provided by MPI. That section also enumerates the datatypes to which each operation can be applied.

In addition, users may define their own operations that can be overloaded to operate on several datatypes, either basic or derived. This is further explained in Section 6.9.5.

The operation `op` is always assumed to be associative. All predefined operations are also assumed to be commutative. Users may define operations that are assumed to be associative, but not commutative. The “canonical” evaluation order of a reduction is determined by the ranks of the processes in the group. However, the implementation can take advantage of associativity, or associativity and commutativity in order to change the order of evaluation. This may change the result of the reduction for operations that are not strictly associative and commutative, such as floating point addition.

Advice to implementors. It is strongly recommended that `MPI_REDUCE` be implemented so that the same result be obtained whenever the function is applied on the same arguments, appearing in the same order. Note that this may prevent optimizations that take advantage of the physical location of ranks. (*End of advice to implementors.*)

Advice to users. Some applications may not be able to ignore the nonassociative nature of floating-point operations or may use user-defined operations (see Section 6.9.5) that require a special reduction order and cannot be treated as associative. Such applications should enforce the order of evaluation explicitly. For example, in the case of operations that require a strict left-to-right (or right-to-left) evaluation order, this could be done by gathering all operands at a single process (e.g., with `MPI_GATHER`), applying the reduction operation in the desired order (e.g., with `MPI_REDUCE_LOCAL`), and if needed, broadcast or scatter the result to the other processes (e.g., with `MPI_BCAST`). (*End of advice to users.*)

The `datatype` argument of `MPI_REDUCE` must be compatible with `op`. Predefined operators work only with the MPI types listed in Section 6.9.2 and Section 6.9.4. Furthermore, the `datatype` and `op` given for predefined operators must be the same on all processes.

Note that it is possible for users to supply different user-defined operations to `MPI_REDUCE` in each process. MPI does not define which operations are used on which operands in this case. User-defined operators may operate on general, derived datatypes. In this case, each argument that the reduce operation is applied to is one element described by such a datatype, which may contain several basic values. This is further explained in Section 6.9.5.

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 intra-communicators is specified by passing the value `MPI_IN_PLACE` to the argument `sendbuf` at the root. In such a case, the input data is taken at the root from the receive buffer, where it will be replaced by the output data.

If `comm` is an inter-communicator, then the call involves all processes in the inter-communicator, 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.

6.9.2 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 6.12), and `MPI_REDUCE_LOCAL`. These operations are invoked by placing the following in `op`.

Name	Meaning	
MPI_MAX	maximum	1
MPI_MIN	minimum	2
MPI_SUM	sum	3
MPI_PROD	product	4
MPI_LAND	logical and	5
MPI_BAND	bit-wise and	6
MPI_LOR	logical or	7
MPI_BOR	bit-wise or	8
MPI_LXOR	logical exclusive or (xor)	9
MPI_BXOR	bit-wise exclusive or (xor)	10
MPI_MAXLOC	max value and location	11
MPI_MINLOC	min value and location	12

The two operations MPI_MINLOC and MPI_MAXLOC are discussed separately in Section 6.9.4. For the other predefined operations, we enumerate below the allowed combinations of `op` and `datatype` arguments. First, define groups of MPI basic datatypes in the following way.

C integer:	MPI_INT, MPI_LONG, MPI_SHORT, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_LONG_LONG_INT, MPI_LONG_LONG (as synonym), MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, MPI_UNSIGNED_CHAR, MPI_INT8_T, MPI_INT16_T, MPI_INT32_T, MPI_INT64_T, MPI_UINT8_T, MPI_UINT16_T, MPI_UINT32_T, and MPI_UINT64_T	13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Fortran integer:	MPI_INTEGER and handles returned from MPI_TYPE_CREATE_F90_INTEGER and, if available, MPI_INTEGER1, MPI_INTEGER2, MPI_INTEGER4, MPI_INTEGER8, and MPI_INTEGER16	32 33 34 35 36 37
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	38 39 40 41 42 43
Logical:	MPI_LOGICAL, MPI_C_BOOL, and MPI_CXX_BOOL	44 45
Complex:	MPI_COMPLEX, MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX (as synonym), MPI_C_DOUBLE_COMPLEX,	46 47 48

```

1      MPI_C_LONG_DOUBLE_COMPLEX,
2      MPI_CXX_FLOAT_COMPLEX,
3      MPI_CXX_DOUBLE_COMPLEX,
4      MPI_CXX_LONG_DOUBLE_COMPLEX,
5      and handles returned from
6      MPI_TYPE_CREATE_F90_COMPLEX
7      and, if available, MPI_DOUBLE_COMPLEX,
8      MPI_COMPLEX4, MPI_COMPLEX8,
9      MPI_COMPLEX16, and MPI_COMPLEX32
10     Byte: MPI_BYTE
11     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 807 in Section 19.3.6.

The following examples use intra-communicators.

Example 6.15. 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.

```

30 SUBROUTINE PAR_BLAS1(m, a, b, c, comm)
31 USE MPI
32 REAL a(m), b(m)      ! local slice of array
33 REAL c                ! result (at node zero)
34 REAL sum
35 INTEGER m, comm, i, ierr
36
37 ! local sum
38 sum = 0.0
39 DO i = 1, m
40     sum = sum + a(i)*b(i)
41 END DO
42
43 ! global sum
44 CALL MPI_REDUCE(sum, c, 1, MPI_REAL, MPI_SUM, 0, comm, ierr)
45 RETURN
46 END

```


Example 6.16. 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)
USE MPI
REAL a(m), b(m,n)    ! local slice of array
REAL c(n)            ! result
REAL sum(n)
INTEGER m, 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

```

6.9.3 Signed Characters and Reductions

The types `MPI_SIGNED_CHAR` and `MPI_UNSIGNED_CHAR` can be used in reduction operations. `MPI_CHAR`, `MPI_WCHAR`, and `MPI_CHARACTER` (which represent printable characters) cannot be used in reduction operations. In a heterogeneous environment, `MPI_CHAR`, `MPI_WCHAR`, and `MPI_CHARACTER` will be translated so as to preserve the printable character, whereas `MPI_SIGNED_CHAR` and `MPI_UNSIGNED_CHAR` will be translated so as to preserve the integer value.

Advice to users. The types `MPI_CHAR`, `MPI_WCHAR`, and `MPI_CHARACTER` are intended for characters, and so will be translated to preserve the printable representation, rather than the integer value, if sent between machines with different character codes. The types `MPI_SIGNED_CHAR` and `MPI_UNSIGNED_CHAR` should be used in C if the integer value should be preserved. (*End of advice to users.*)

6.9.4 MINLOC and MAXLOC

The operator `MPI_MINLOC` is used to compute a global minimum and also an index attached to the minimum value. `MPI_MAXLOC` similarly computes a global maximum and index. One application of these is to compute a global minimum (maximum) and the rank of the process containing this value.

The operation that defines `MPI_MAXLOC` is:

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$

1 where

$$2 \quad w = \max(u, v)$$

4 and

$$5 \quad k = \begin{cases} i & \text{if } u > v \\ \min(i, j) & \text{if } u = v \\ j & \text{if } u < v \end{cases}$$

9 MPI_MINLOC is defined similarly:

$$11 \quad \begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$

14 where

$$16 \quad w = \min(u, v)$$

18 and

$$19 \quad k = \begin{cases} i & \text{if } u < v \\ \min(i, j) & \text{if } u = v \\ j & \text{if } u > v \end{cases}$$

23 Both operations are associative and commutative. Note that if MPI_MAXLOC is applied
 24 to reduce a sequence of pairs $(u_0, 0), (u_1, 1), \dots, (u_{n-1}, n-1)$, then the value returned is
 25 (u, r) , where $u = \max_i u_i$ and r is the index of the first global maximum in the sequence.
 26 Thus, if each process supplies a value and its rank within the group, then a reduce operation
 27 with `op = MPI_MAXLOC` will return the maximum value and the rank of the first process with
 28 that value. Similarly, MPI_MINLOC can be used to return a minimum and its index. More
 29 generally, MPI_MINLOC computes a *lexicographic minimum*, where elements are ordered
 30 according to the first component of each pair, and ties are resolved according to the second
 31 component.

32 The reduce operation is defined to operate on arguments that consist of a pair: value
 33 and index. For both Fortran and C, types are provided to describe the pair. The potentially
 34 mixed-type nature of such arguments is a problem in Fortran. The problem is circumvented,
 35 for Fortran, by having the MPI-provided type consist of a pair of the same type as value,
 36 and coercing the index to this type also. In C, the MPI-provided pair type has distinct
 37 types and the index is an `int`.

38 In order to use MPI_MINLOC and MPI_MAXLOC in a reduce operation, one must provide
 39 a `datatype` argument that represents a pair (value and index). MPI provides nine such
 40 predefined datatypes. The operations MPI_MAXLOC and MPI_MINLOC can be used with
 41 each of the following datatypes.

42 Fortran:

43 Name	Description
44 MPI_2REAL	pair of REALs
45 MPI_2DOUBLE_PRECISION	pair of DOUBLE PRECISION variables
46 MPI_2INTEGER	pair of INTEGERS

48 C:

Name	Description	
MPI_FLOAT_INT	float and int	1
MPI_DOUBLE_INT	double and int	2
MPI_LONG_INT	long and int	3
MPI_2INT	pair of int	4
MPI_SHORT_INT	short and int	5
MPI_LONG_DOUBLE_INT	long double and int	6

The datatype MPI_2REAL is *as if* defined by the following (see Section 5.1).

```
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);
MPI_Type_commit(&MPI_SHORT_INT);
```

Similar statements apply for MPI_FLOAT_INT, MPI_LONG_INT and MPI_DOUBLE_INT.

The following examples use intra-communicators.

Example 6.17. Each process has an array of 30 doubles, in C. For each of the 30 locations, compute the value and rank of the process containing the largest value.

```
...
/* each process has an array of 30 double: ain[30]
*/
double ain[30], aout[30];
int ind[30];
struct {
    double val;
    int rank;
} in[30], out[30];
int i, myrank, root;

MPI_Comm_rank(comm, &myrank);
for (i=0; i<30; ++i) {
    in[i].val = ain[i];
    in[i].rank = myrank;
}
MPI_Reduce(in, out, 30, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm);
/* At this point, the answer resides on process root
*/
if (myrank == root) {
    /* read ranks out
```

```

1      */
2      for (i=0; i<30; ++i) {
3          aout[i] = out[i].val;
4          ind[i] = out[i].rank;
5      }
6  }

```

Example 6.18. Same example, in Fortran.

```

10      ...
11      ! each process has an array of 30 double: ain(30)
12
13      DOUBLE PRECISION ain(30), aout(30)
14      INTEGER ind(30)
15      DOUBLE PRECISION in(2,30), out(2,30)
16      INTEGER i, myrank, root, ierr
17
18      CALL MPI_COMM_RANK(comm, myrank, ierr)
19      DO i=1,30
20          in(1,i) = ain(i)
21          in(2,i) = myrank      ! myrank is coerced to a double
22      END DO
23
24      CALL MPI_REDUCE(in, out, 30, MPI_2DOUBLE_PRECISION, MPI_MAXLOC, root,&
25                      comm, ierr)
26      ! At this point, the answer resides on process root
27
28      IF (myrank .EQ. root) THEN
29          ! read ranks out
30          DO i=1,30
31              aout(i) = out(1,i)
32              ind(i) = out(2,i) ! rank is coerced back to an integer
33          END DO
34      END IF

```

Example 6.19. Each process has a nonempty array of values. Find the minimum global value, the rank of the process that holds it and its index on this process.

```

36      #define LEN 1000
37
38      float val[LEN];          /* local array of values */
39      int count;              /* local number of values */
40      int myrank, minrank, minindex;
41      float minval;
42
43      struct {
44          float value;
45          int index;
46      } in, out;
47
48      /* local minloc */

```

```

1  in.value = val[0];
2  in.index = 0;
3  for (i=1; i < count; i++)
4      if (in.value > val[i]) {
5          in.value = val[i];
6          in.index = i;
7      }
8
9      /* global minloc */
10 MPI_Comm_rank(comm, &myrank);
11 in.index = myrank*LEN + in.index;
12 MPI_Reduce(&in, &out, 1, MPI_FLOAT_INT, MPI_MINLOC, root, comm);
13 /* At this point, the answer resides on process root
14 */
15 if (myrank == root) {
16     /* read answer out
17     */
18     minval = out.value;
19     minrank = out.index / LEN;
20     minindex = out.index % LEN;
21 }

```

Rationale. The definition of MPI_MINLOC and MPI_MAXLOC given here has the advantage that it does not require any special-case handling of these two operations: they are handled like any other reduce operation. By assigning a value other than myrank to the in.index field, a programmer can provide a different definition of MPI_MAXLOC and MPI_MINLOC, if so desired. The disadvantage is that values and indices have to be first interleaved, and that indices and values have to be coerced to the same type, in Fortran. (*End of rationale.*)

6.9.5 User-Defined Reduction Operations

MPI_OP_CREATE(user_fn, commute, op)

IN	user_fn	user defined function (function)
IN	commute	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)
```

```
int MPI_op_create_c(MPI_User_function_c *user_fn, int commute, MPI_Op *op)
```

Fortran 2008 binding

```

MPI_op_create(user_fn, commute, op, ierror)
  PROCEDURE(MPI_User_function) :: user_fn
  LOGICAL, INTENT(IN) :: commute
  TYPE(MPI_Op), INTENT(OUT) :: op
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

1 MPI_Op_create_c(user_fn, commute, op, ierror) !(_c)
2     PROCEDURE(MPI_User_function_c) :: user_fn
3     LOGICAL, INTENT(IN) :: commute
4     TYPE(MPI_Op), INTENT(OUT) :: op
5     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

7 MPI_OP_CREATE(USER_FN, COMMUTE, OP, IERROR)
8     EXTERNAL USER_FN
9     LOGICAL COMMUTE
10    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 6.12), 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, taking advantage of the associativity of the operation. If `commute = true` then the order of evaluation can be changed, taking advantage of commutativity and associativity.

In Fortran when using `USE mpi_f08`, the large count variant shall be called explicitly as `MPI_Op_create_c` (i.e., with suffix “_c”) because interface polymorphism cannot be used to differentiate between the two different user callback prototypes despite their different type signatures.

The argument `user_fn` is the user-defined function, which must have the following four arguments: `invec`, `inoutvec`, `len`, and `datatype`.

`MPI_USER_FUNCTION` also supports large count types in separate additional MPI callback function prototype declarations in C (suffixed with the “_c”) and in Fortran when using `USE mpi_f08`.

The ISO C prototypes for the functions are the following.

```

32 typedef void MPI_User_function(void *invec, void *inoutvec, int *len,
33     MPI_Datatype *datatype);
34
35 typedef void MPI_User_function_c(void *invec, void *inoutvec, MPI_Count *len,
36     MPI_Datatype *datatype);

```

The Fortran declarations of the user-defined function `user_fn` appear below.

```

38 ABSTRACT INTERFACE
39     SUBROUTINE MPI_User_function(invec, inoutvec, len, datatype)
40     USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
41     TYPE(C_PTR), VALUE :: invec, inoutvec
42     INTEGER :: len
43     TYPE(MPI_Datatype) :: datatype
44
45 ABSTRACT INTERFACE
46     SUBROUTINE MPI_User_function_c(invec, inoutvec, len, datatype) !(_c)
47     USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
48     TYPE(C_PTR), VALUE :: invec, inoutvec

```

```

INTEGER(KIND=MPI_COUNT_KIND) :: 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 datatype 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 $w[i] = u[i] \circ v[i]$, for $i=0, \dots, len-1$, where \circ is the reduce operation that the function computes.

Informally, we can think of `invec` and `inoutvec` as arrays of `len` elements that `user_fn` is combining. The result of the reduction over-writes values in `inoutvec`, hence the name. Each invocation of the function results in the pointwise evaluation of the reduce operator on `len` elements: i.e., the function returns in `inoutvec[i]` the value `invec[i] \circ inoutvec[i]`, for $i=0, \dots, count-1$, where \circ is the combining operation computed by the function.

Rationale. The `len` argument allows `MPI_REDUCE` to avoid calling the function for each element in the input buffer. Rather, the system can choose to apply the function to chunks of input. In C, it is passed in as a reference for reasons of compatibility with Fortran.

By internally comparing the value of the `datatype` argument to known, global handles, it is possible to overload the use of a single user-defined function for several, different datatypes. (*End of rationale.*)

When calling any reduction or prefix scan MPI procedure with a user-defined MPI operator, the type of the `count` parameter in the call to the reduction or prefix scan MPI procedure does not need to be identical to the type of the `len` parameter in the user function associated with the user-defined MPI operator. If the `count` parameter has a type of `int` in C or `INTEGER` in Fortran and the `len` parameter has a type of `MPI_COUNT`, then MPI will perform the appropriate widening type conversion of the `len` parameter. If the `count` parameter has a type of `MPI_COUNT` and the `len` parameter has a type of `int` in C or `INTEGER` in Fortran, then MPI will perform the appropriate narrowing type conversion of the `len` parameter. If this narrowing conversion would result in truncation of the `len` value, then MPI will call the user function multiple times with a sequence of values for `len` that sum to the value of `count`.

Advice to implementors. If the number of data items cannot be represented in `len`, the implementation may need to invoke `user_fn` multiple times. (*End of advice to implementors.*)

General datatypes may be passed to the user function. However, use of datatypes that are not contiguous is likely to lead to inefficiencies.

No MPI communication function may be called inside the user function. `MPI_ABORT` may be called inside the function in case of an error.

Advice to users. Suppose one defines a library of user-defined reduce functions 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 and only valid for intra-communicators.

```

20 MPI_Comm_size(comm, &groupsize);
21 MPI_Comm_rank(comm, &rank);
22 if (rank > 0) {
23     MPI_Recv(tempbuf, count, datatype, rank-1,...);
24     User_reduce(tempbuf, sendbuf, count, datatype);
25 }
26 if (rank < groupsize-1) {
27     MPI_Send(sendbuf, count, datatype, rank+1, ...);
28 }
29 /* answer now resides in process groupsize-1 ... now send to root
30 */
31 if (rank == root) {
32     MPI_Irecv(recvbuf, count, datatype, groupsize-1,..., &req);
33 }
34 if (rank == groupsize-1) {
35     MPI_Send(sendbuf, count, datatype, root, ...);
36 }
37 if (rank == root) {
38     MPI_Wait(&req, &status);
39 }

```

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 noncommutative 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)
```

Fortran 2008 binding

```
MPI_Op_free(op, ierror)
  TYPE(MPI_Op), INTENT(INOUT) :: op
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran 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 intra-communicator.

Example 6.20. 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...
*/
...
```

```

1
2      /* each process has an array of 100 Complexes
3      */
4      Complex a[100], answer[100];
5      MPI_Op myOp;
6      MPI_Datatype ctype;
7
8      /* explain to MPI how type Complex is defined
9      */
10     MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
11     MPI_Type_commit(&ctype);
12     /* create the complex-product user-op
13     */
14     MPI_Op_create(myProd, 1, &myOp);
15
16     MPI_Reduce(a, answer, 100, ctype, myOp, root, comm);
17
18     /* At this point, the answer, which consists of 100 Complexes,
19     * resides on process root
20     */

```

Example 6.21. How to use the `mpi_f08` interface of the Fortran `MPI_User_function`.

```

21
22 subroutine my_user_function(invec, inoutvec, len, dtype)  bind(c)
23 use, intrinsic :: iso_c_binding, only : c_ptr, c_f_pointer
24 use mpi_f08
25 type(c_ptr), value :: invec, inoutvec
26 integer :: len
27 type(MPI_Datatype) :: dtype
28 real, pointer :: invec_r(:), inoutvec_r(:)
29 if (dtype == MPI_REAL) then
30     call c_f_pointer(invec, invec_r, (/ len /))
31     call c_f_pointer(inoutvec, inoutvec_r, (/ len /))
32     inoutvec_r = invec_r + inoutvec_r
33 end if
34 end subroutine

```

6.9.6 All-Reduce

MPI includes a variant of the reduce operations where the result is returned to all processes in a group. MPI requires that all processes from the same group participating in these operations receive identical results.

`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	datatype of elements of send buffer (handle)	1
IN	op	operation (handle)	2
IN	comm	communicator (handle)	3
			4
			5

C binding

```
int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,
                 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

```

```
int MPI_Allreduce_c(const void *sendbuf, void *recvbuf, MPI_Count count,
                   MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

```

Fortran 2008 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

```

```
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror) !(_c)
  TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
  TYPE(*), DIMENSION(..) :: recvbuf
  INTEGER(KIND=MPI_COUNT_KIND), 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

```

Fortran binding

```
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
  <type> SENDBUF(*), RECVBUF(*)
  INTEGER COUNT, DATATYPE, OP, COMM, IERROR

```

If `comm` is an intra-communicator, `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 intra-communicators is specified by passing the value `MPI_IN_PLACE` to the argument `sendbuf` at all processes. In this case, the input data is taken at each process from the receive buffer, where it will be replaced by the output data.

If `comm` is an inter-communicator, then the result of the reduction of the data provided by processes in group A is stored at each process in group B, and vice versa. Both groups should provide `count` and `datatype` arguments that specify the same type signature.

The following example uses an intra-communicator.

Example 6.22. A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 6.16).

```

1  SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
2  USE MPI
3  REAL a(m), b(m,n)    ! local slice of array
4  REAL c(n)            ! result
5  REAL sum(n)
6  INTEGER m, n, comm, i, j, ierr
7
8  ! local sum
9  DO j=1,n
10     sum(j) = 0.0
11     DO i=1,m
12         sum(j) = sum(j) + a(i)*b(i,j)
13     END DO
14 END DO
15
16 ! global sum
17 CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, comm, ierr)
18
19 ! return result at all nodes
20 RETURN
21 END

```

6.9.7 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	datatype 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)
```

```
int MPI_Reduce_local_c(const void *inbuf, void *inoutbuf, MPI_Count count,
                       MPI_Datatype datatype, MPI_Op op)
```

Fortran 2008 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

MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror) !(_c)
  TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
  TYPE(*), DIMENSION(..) :: inoutbuf
  INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Op), INTENT(IN) :: op
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran 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 6.9.5. Both `inbuf` and `inoutbuf` (input as well as result) have the same number of elements given by `count` and the same datatype given by `datatype`. The `MPI_IN_PLACE` option is not allowed.

Reduction operations can be queried for their commutativity.

```

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)

```

Fortran 2008 binding

```

MPI_Op_commutative(op, commute, ierror)
  TYPE(MPI_Op), INTENT(IN) :: op
  LOGICAL, INTENT(OUT) :: commute
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR)
  INTEGER OP, IERROR
  LOGICAL COMMUTE

```

6.10 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.

6.10.1 MPI_REDUCE_SCATTER_BLOCK

`MPI_REDUCE_SCATTER_BLOCK(sendbuf, recvbuf, recvcnt, datatype, op, comm)`

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcnt	element count per block (non-negative integer)
IN	datatype	datatype 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 recvcnt,
    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

```
int MPI_Reduce_scatter_block_c(const void *sendbuf, void *recvbuf,
    MPI_Count recvcnt, MPI_Datatype datatype, MPI_Op op,
    MPI_Comm comm)
```

Fortran 2008 binding

```
MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcnt, datatype, op, comm,
    ierror)
```

```
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
```

```
TYPE(*), DIMENSION(..) :: recvbuf
```

```
INTEGER, INTENT(IN) :: recvcnt
```

```
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, recvcnt, datatype, op, comm,
    ierror) !(_c)
```

```
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
```

```
TYPE(*), DIMENSION(..) :: recvbuf
```

```
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcnt
```

```
TYPE(MPI_Datatype), INTENT(IN) :: datatype
```

```
TYPE(MPI_Op), INTENT(IN) :: op
```

```
TYPE(MPI_Comm), INTENT(IN) :: comm
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran 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 intra-communicator, `MPI_REDUCE_SCATTER_BLOCK` first performs a global, element-wise reduction on vectors of `count = n*recvcount` elements in the send buffers defined by `sendbuf`, `count` and `datatype`, using the operation `op`, where `n` is the number of processes in the group of `comm`. The routine is called by all group members using the same arguments for `recvcount`, `datatype`, `op` and `comm`. The resulting vector is treated as `n` consecutive blocks of `recvcount` elements that are scattered to the processes of the group. The `i`-th block is sent to process `i` and stored in the receive buffer defined by `recvbuf`, `recvcount`, and `datatype`.

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 intra-communicators is specified by passing `MPI_IN_PLACE` in the `sendbuf` argument on *all* processes. In this case, the input data is taken from the receive buffer.

If `comm` is an inter-communicator, then the result of the reduction of the data provided by processes in one group (group A) is scattered among processes in the other group (group B) and vice versa. Within each group, all processes provide the same value for the `recvcount` argument, and provide input vectors of `count = n*recvcount` elements stored in the send buffers, where `n` is the size of the group. The number of elements `count` must be the same 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.*)

6.10.2 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	<code>sendbuf</code>	starting address of send buffer (choice)
OUT	<code>recvbuf</code>	starting address of receive buffer (choice)
IN	<code>recvcounts</code>	nonnegative integer array (of length group size) specifying the number of elements of the result distributed to each process.

```

1      IN      datatype      datatype of elements of send and receive buffers
2                                (handle)
3
4      IN      op            operation (handle)
5
6      IN      comm         communicator (handle)

```

C binding

```

7
8      int MPI_Reduce_scatter(const void *sendbuf, void *recvbuf,
9                            const int recvcnts[], MPI_Datatype datatype, MPI_Op op,
10                           MPI_Comm comm)
11
12     int MPI_Reduce_scatter_c(const void *sendbuf, void *recvbuf,
13                             const MPI_Count recvcnts[], MPI_Datatype datatype, MPI_Op op,
14                             MPI_Comm comm)

```

Fortran 2008 binding

```

15
16     MPI_Reduce_scatter(sendbuf, recvbuf, recvcnts, datatype, op, comm, ierror)
17         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
18         TYPE(*), DIMENSION(..) :: recvbuf
19         INTEGER, INTENT(IN) :: recvcnts(*)
20         TYPE(MPI_Datatype), INTENT(IN) :: datatype
21         TYPE(MPI_Op), INTENT(IN) :: op
22         TYPE(MPI_Comm), INTENT(IN) :: comm
23         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
25     MPI_Reduce_scatter(sendbuf, recvbuf, recvcnts, datatype, op, comm, ierror)
26         !(_c)
27         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
28         TYPE(*), DIMENSION(..) :: recvbuf
29         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcnts(*)
30         TYPE(MPI_Datatype), INTENT(IN) :: datatype
31         TYPE(MPI_Op), INTENT(IN) :: op
32         TYPE(MPI_Comm), INTENT(IN) :: comm
33         INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

34
35     MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, IERROR)
36         <type> SENDBUF(*), RECVBUF(*)
37         INTEGER REVCOUNTS(*), DATATYPE, OP, COMM, IERROR

```

If `comm` is an intra-communicator, `MPI_REDUCE_SCATTER` first performs a global, element-wise reduction on vectors of $\text{count} = \sum_{i=0}^{n-1} \text{recvcnts}[i]$ 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 `recvcnts`, `datatype`, `op` and `comm`. The resulting vector is treated as `n` consecutive blocks where the number of elements of the `i`-th block is `recvcnts[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`, `recvcnts[i]` and `datatype`.

Advice to implementors. The `MPI_REDUCE_SCATTER` routine is functionally equivalent to: an `MPI_REDUCE` collective operation with `count` equal to the sum of

recvcounts[i] followed by MPI_SCATTERV with sendcounts equal to recvcounts. However, a direct implementation may run faster. (*End of advice to implementors.*)

The “in place” option for intra-communicators is specified by passing MPI_IN_PLACE in the sendbuf argument. In this case, the input data is taken from the receive buffer. It is not required to specify the “in place” option on all processes, since the processes for which recvcounts[i] = 0 may not have allocated a receive buffer.

If comm is an inter-communicator, then the result of the reduction of the data provided by processes in one group (group A) is scattered among processes in the other group (group B), and vice versa. Within each group, all processes provide the same recvcounts argument, and provide input vectors of count = $\sum_{i=0}^{n-1} \text{recvcounts}[i]$ elements stored in the send buffers, where n is the size of the group. The resulting vector from the other group is scattered in blocks of recvcounts[i] elements among the processes in the group. The number of elements count must be the same for the two groups.

Rationale. The last restriction is needed so that the length of the send buffer can be determined by the sum of the local recvcounts entries. Otherwise, a communication is needed to figure out how many elements are reduced. (*End of rationale.*)

6.11 Scan

6.11.1 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	datatype 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)
```

```
int MPI_Scan_c(const void *sendbuf, void *recvbuf, MPI_Count count,
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Fortran 2008 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
```

```

1     TYPE(MPI_Op), INTENT(IN) :: op
2     TYPE(MPI_Comm), INTENT(IN) :: comm
3     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
5 MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror) !(_c)
6     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
7     TYPE(*), DIMENSION(..) :: recvbuf
8     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
9     TYPE(MPI_Datatype), INTENT(IN) :: datatype
10    TYPE(MPI_Op), INTENT(IN) :: op
11    TYPE(MPI_Comm), INTENT(IN) :: comm
12    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

13 MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
14     <type> SENDBUF(*), RECVBUF(*)
15     INTEGER COUNT, DATATYPE, OP, COMM, IERROR
16
17

```

18 If `comm` is an intra-communicator, `MPI_SCAN` is used to perform a prefix reduction
19 on data distributed across the group. The operation returns, in the receive buffer of the
20 process with rank `i`, the reduction of the values in the send buffers of processes with ranks
21 `0,...,i` (inclusive). The routine is called by all group members using the same arguments
22 for `count`, `datatype`, `op` and `comm`, except that for user-defined operations, the same rules
23 apply as for `MPI_REDUCE`. The type of operations supported, their semantics, and the
24 constraints on send and receive buffers are as for `MPI_REDUCE`.

25 The “in place” option for intra-communicators is specified by passing `MPI_IN_PLACE`
26 in the `sendbuf` argument. In this case, the input data is taken from the receive buffer, and
27 replaced by the output data.

28 This operation is invalid for inter-communicators.

6.11.2 Exclusive Scan

```

33 MPI_EXSCAN(sendbuf, recvbuf, count, datatype, op, comm)
34     IN     sendbuf           starting address of send buffer (choice)
35     OUT    recvbuf          starting address of receive buffer (choice)
36     IN     count            number of elements in input buffer (non-negative
37                               integer)
38
39     IN     datatype         datatype of elements of input buffer (handle)
40
41     IN     op               operation (handle)
42
43     IN     comm             intra-communicator (handle)

```

C binding

```

45 int MPI_Exscan(const void *sendbuf, void *recvbuf, int count,
46               MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
47
48

```

```
int MPI_Exscan_c(const void *sendbuf, void *recvbuf, MPI_Count count,
                MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Fortran 2008 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

MPI_Exscan(sendbuf, recvbuf, count, datatype, op, comm, ierror) !(_c)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER(KIND=MPI_COUNT_KIND), 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
```

Fortran binding

```
MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

If `comm` is an intra-communicator, `MPI_EXSCAN` is used to perform a prefix reduction on data distributed across the group. The value in `recvbuf` on the process with rank 0 is undefined, and `recvbuf` is not significant on process 0. The value in `recvbuf` on the process with rank 1 is defined as the value in `sendbuf` on the process with rank 0. For processes with rank $i > 1$, the operation returns, in the receive buffer of the process with rank i , the reduction of the values in the send buffers of processes with ranks $0, \dots, i-1$ (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 intra-communicators is specified by passing `MPI_IN_PLACE` in the `sendbuf` argument. In this case, the input data is taken from the receive buffer, and replaced by the output data. The receive buffer on rank 0 is not changed by this operation.

This operation is invalid for inter-communicators.

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 noninvertable operations such as `MPI_MAX`, the exclusive scan cannot be computed with the inclusive scan. (*End of rationale.*)

6.11.3 Example using MPI_SCAN

The example in this section uses an intra-communicator.

Example 6.23. 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:

<i>values</i>	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
<i>logicals</i>	0	0	1	1	1	0	0	1
<i>result</i>	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 noncommutative operator. C code that implements it is given below.

```

19 typedef struct {
20     double val;
21     int log;
22 } SegScanPair;
23
24 /* the user-defined function
25 */
26 void segScan(SegScanPair *in, SegScanPair *inout, int *len,
27             MPI_Datatype *dptr)
28 {
29     int i;
30     SegScanPair c;
31
32     for (i=0; i< *len; ++i) {
33         if (in->log == inout->log)
34             c.val = in->val + inout->val;
35         else
36             c.val = inout->val;
37         c.log = inout->log;
38         *inout = c;
39         in++; inout++;
40     }
41 }

```

Note that the `inout` argument to the user-defined function corresponds to the right-hand operand of the operator. When using this operator, we must be careful to specify that it is noncommutative, as in the following.

```

43 int i, base;
44 SegScanPair a, answer;
45 MPI_Op myOp;
46 MPI_Datatype type[2] = {MPI_DOUBLE, MPI_INT};
47 MPI_Aint disp[2];
48 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);

```

6.12 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 [34, 38]. 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 [36].

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

1 safely accessed and modified. Completion does not indicate that other processes have
2 completed or even started the operation (unless otherwise implied by the description of
3 the operation). Completion of a particular nonblocking collective operation also does not
4 indicate completion of any other posted nonblocking collective (or send-receive) operations,
5 whether they are posted before or after the completed operation.
6

7 *Advice to users.* Users should be aware that implementations are allowed, but
8 not required (with exception of `MPI_IBARRIER`), to synchronize processes during the
9 completion of a nonblocking collective operation. (*End of advice to users.*)

10
11 Upon returning from a completion call in which a nonblocking collective operation
12 completes, the values of the `MPI_SOURCE` and `MPI_TAG` fields in the associated status object,
13 if any, are undefined. The value of `MPI_ERROR` may be defined, if appropriate, according
14 to the specification in Section 3.2.5. It is valid to mix different request types (i.e., any
15 combination of collective requests, I/O requests, generalized requests, or point-to-point
16 requests) in functions that enable multiple completions (e.g., `MPI_WAITALL`). It is erroneous
17 to call `MPI_REQUEST_FREE` or `MPI_CANCEL` for a request associated with a nonblocking
18 collective operation. Nonblocking collective requests created using the APIs described in
19 this section are not persistent. However, persistent collective requests can be created using
20 persistent collective operations described in Sections 6.13 and 8.8.

21 *Rationale.* Freeing an active nonblocking collective request could cause similar
22 problems as discussed for point-to-point requests (see Section 3.7.3). Cancelling a
23 request is not supported because the semantics of this operation are not well-defined.
24 (*End of rationale.*)
25

26 Multiple nonblocking collective operations can be outstanding on a single communica-
27 tor. If the nonblocking call causes some system resource to be exhausted, then it will fail
28 and raise an error. Quality implementations of MPI should ensure that this happens only
29 in pathological cases. That is, an MPI implementation should be able to support a large
30 number of pending nonblocking operations.

31 Unlike point-to-point operations, nonblocking collective operations do not match with
32 blocking collective operations, and collective operations do not have a tag argument. All
33 processes must call collective operations (blocking and nonblocking) in the same order
34 per communicator. In particular, once a process calls a collective operation, all other
35 processes in the communicator must eventually call the same collective operation, and no
36 other collective operation with the same communicator in between. This is consistent with
37 the ordering rules for blocking collective operations in threaded environments.
38

39 *Rationale.* Matching blocking and nonblocking collective operations is not allowed
40 because the implementation might use different communication algorithms for the two
41 cases. Blocking collective operations may be optimized for minimal time to comple-
42 tion, while nonblocking collective operations may balance time to completion with
43 CPU overhead and asynchronous progress.

44 The use of tags for collective operations can prevent certain hardware optimizations.
45 (*End of rationale.*)
46

47 *Advice to users.* If program semantics require matching blocking and nonblocking
48 collective operations, then a nonblocking collective operation can be initiated and

immediately completed with a blocking wait to emulate blocking behavior. (*End of advice to users.*)

In terms of data movement, each nonblocking collective operation has the same effect as its blocking counterpart for intra-communicators and inter-communicators after completion. Likewise, upon completion, nonblocking collective reduction operations have the same effect as their blocking counterparts, and the same restrictions and recommendations on reduction orders apply.

The use of the “in place” option is allowed exactly as described for the corresponding blocking collective operations. When using the “in place” option, message buffers function as both send and receive buffers. Such buffers should not be modified or accessed until the operation completes.

The *progress* rules for nonblocking collective operations are similar to the progress rules for nonblocking point-to-point operations, refer to Section 3.7.4.

Advice to implementors. Nonblocking collective operations can be implemented with local execution schedules [37] using nonblocking point-to-point communication and a reserved tag-space. (*End of advice to implementors.*)

6.12.1 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)
```

Fortran 2008 binding

```
MPI_Ibarrier(comm, request, ierror)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Request), INTENT(OUT) :: request
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran 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 inter-communicator 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

1 can overlap the barrier latency and therefore shorten possible waiting times. The se-
 2 mantic properties are also useful when mixing collective operations and point-to-point
 3 messages. (*End of advice to users.*)
 4

5 6.12.2 Nonblocking Broadcast

8 MPI_IBCAST(buffer, count, datatype, root, comm, request)

10	INOUT	buffer	starting address of buffer (choice)
11	IN	count	number of entries in buffer (non-negative integer)
12	IN	datatype	datatype of buffer (handle)
13	IN	root	rank of broadcast root (integer)
14	IN	comm	communicator (handle)
15	IN	comm	communicator (handle)
16	OUT	request	communication request (handle)
17			

18 C binding

19 int MPI_Ibcast(void *buffer, int count, MPI_Datatype datatype, int root,
 20 MPI_Comm comm, MPI_Request *request)

21
 22 int MPI_Ibcast_c(void *buffer, MPI_Count count, MPI_Datatype datatype,
 23 int root, MPI_Comm comm, MPI_Request *request)

24 Fortran 2008 binding

25 MPI_Ibcast(buffer, count, datatype, root, comm, request, ierror)

26 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer

27 INTEGER, INTENT(IN) :: count, root

28 TYPE(MPI_Datatype), INTENT(IN) :: datatype

29 TYPE(MPI_Comm), INTENT(IN) :: comm

30 TYPE(MPI_Request), INTENT(OUT) :: request

31 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

32
 33 MPI_Ibcast(buffer, count, datatype, root, comm, request, ierror) !(_c)

34 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer

35 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count

36 TYPE(MPI_Datatype), INTENT(IN) :: datatype

37 INTEGER, INTENT(IN) :: root

38 TYPE(MPI_Comm), INTENT(IN) :: comm

39 TYPE(MPI_Request), INTENT(OUT) :: request

40 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

41 Fortran binding

42 MPI_IBCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR)

43 <type> BUFFER(*)

44 INTEGER COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR

45 This call starts a nonblocking variant of MPI_BCAST (see Section 6.4).
 46
 47
 48

Example using MPI_IBCAST

The example in this section uses an intra-communicator.

Example 6.24. Start a broadcast of 100 ints 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);

```

6.12.3 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	datatype 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	datatype of recv buffer elements (handle, significant only at root)
IN	root	rank of receiving process (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

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)

```

```

int MPI_Igather_c(const void *sendbuf, MPI_Count sendcount,
                  MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
                  MPI_Datatype recvtype, int root, MPI_Comm comm,
                  MPI_Request *request)

```

Fortran 2008 binding

```

1 MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root,
2           comm, request, ierror)
3

```

```

4     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
5     INTEGER, INTENT(IN) :: sendcount, recvcount, root
6     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
7     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
8     TYPE(MPI_Comm), INTENT(IN) :: comm
9     TYPE(MPI_Request), INTENT(OUT) :: request
10    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11

```

```

12 MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root,
13           comm, request, ierror) !(_c)
14

```

```

15     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
16     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
17     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
18     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
19     INTEGER, INTENT(IN) :: root
20     TYPE(MPI_Comm), INTENT(IN) :: comm
21     TYPE(MPI_Request), INTENT(OUT) :: request
22     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

23 MPI_IGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT,
24           COMM, REQUEST, IERROR)
25

```

```

26 <type> SENDBUF(*), RECVBUF(*)
27

```

```

28 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,
29 IERROR
30

```

This call starts a nonblocking variant of MPI_GATHER (see Section 6.5).

```

31 MPI_IGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcnts, displs, recvtype, root,
32           comm, request)
33

```

34	IN	sendbuf	starting address of send buffer (choice)
35	IN	sendcount	number of elements in send buffer (non-negative integer)
36			
37	IN	sendtype	datatype of send buffer elements (handle)
38	OUT	recvbuf	address of receive buffer (choice, significant only at root)
39			
40			
41	IN	recvcnts	nonnegative integer array (of length group size) containing the number of elements that are received from each process (significant only at root)
42			
43			
44	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)
45			
46			
47			
48			

IN	recvtype	datatype of recv buffer elements (handle, significant only at root)	1 2
IN	root	rank of receiving process (integer)	3 4
IN	comm	communicator (handle)	5
OUT	request	communication request (handle)	6 7

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)
```

```
int MPI_Igatherv_c(const void *sendbuf, MPI_Count sendcount,
                  MPI_Datatype sendtype, void *recvbuf,
                  const MPI_Count recvcounts[], const MPI_Aint displs[],
                  MPI_Datatype recvtype, int root, MPI_Comm comm,
                  MPI_Request *request)
```

Fortran 2008 binding

```
MPI_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
             recvtype, root, comm, request, ierror)
```

```
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
```

```
INTEGER, INTENT(IN) :: sendcount, root
```

```
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
```

```
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
```

```
INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
```

```
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) !(_c)
```

```
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
```

```
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
```

```
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
```

```
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
```

```
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
```

```
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
```

```
INTEGER, INTENT(IN) :: root
```

```
TYPE(MPI_Comm), INTENT(IN) :: comm
```

```
TYPE(MPI_Request), INTENT(OUT) :: request
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_IGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
             RECVTYPE, ROOT, COMM, REQUEST, IERROR)
```

```
<type> SENDBUF(*), RECVBUF(*)
```

```

1     INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVMODE, ROOT,
2         COMM, REQUEST, IERROR

```

This call starts a nonblocking variant of MPI_GATHERV (see Section 6.5).

6.12.4 Nonblocking Scatter

```

9     MPI_ISCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcnt, recvtpe, root, comm,
10                request)

```

11	IN	sendbuf	address of send buffer (choice, significant only at root)
12			
13	IN	sendcount	number of elements sent to each process (non-negative integer, significant only at root)
14			
15	IN	sendtype	datatype of send buffer elements (handle, significant only at root)
16			
17	OUT	recvbuf	address of receive buffer (choice)
18			
19	IN	recvcnt	number of elements in receive buffer (non-negative integer)
20			
21	IN	recvtpe	datatype of receive buffer elements (handle)
22			
23	IN	root	rank of sending process (integer)
24			
25	IN	comm	communicator (handle)
26	OUT	request	communication request (handle)

C binding

```

29     int MPI_Iscatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
30                    void *recvbuf, int recvcnt, MPI_Datatype recvtpe, int root,
31                    MPI_Comm comm, MPI_Request *request)

```

```

32     int MPI_Iscatter_c(const void *sendbuf, MPI_Count sendcount,
33                      MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcnt,
34                      MPI_Datatype recvtpe, int root, MPI_Comm comm,
35                      MPI_Request *request)

```

Fortran 2008 binding

```

38     MPI_ISCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcnt, recvtpe, root,
39                comm, request, ierror)

```

```

40     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
41     INTEGER, INTENT(IN) :: sendcount, recvcnt, root
42     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtpe
43     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
44     TYPE(MPI_Comm), INTENT(IN) :: comm
45     TYPE(MPI_Request), INTENT(OUT) :: request
46     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root,
             comm, request, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER, INTENT(IN) :: root
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

MPI_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 6.6).

```

MPI_ISCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root,
              comm, request)

```

IN	sendbuf	address of send buffer (choice, significant only at root)	23 24 25
IN	sendcounts	nonnegative integer array (of length group size) specifying the number of elements to send to each rank (significant only at root)	26 27 28
IN	displs	integer array (of length group size). Entry <i>i</i> specifies the displacement (relative to <code>sendbuf</code>) from which to take the outgoing data to process <i>i</i> (significant only at root)	29 30 31 32
IN	sendtype	datatype of send buffer elements (handle, significant only at root)	33 34
OUT	recvbuf	address of receive buffer (choice)	35 36
IN	recvcount	number of elements in receive buffer (non-negative integer)	37 38
IN	recvtype	datatype of receive buffer elements (handle)	39
IN	root	rank of sending process (integer)	40 41
IN	comm	communicator (handle)	42
OUT	request	communication request (handle)	43 44

C binding

```

int MPI_Iscatterv(const void *sendbuf, const int sendcounts[],
                 const int displs[], MPI_Datatype sendtype, void *recvbuf,

```

```

1         int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm,
2         MPI_Request *request)
3
4     int MPI_Iscatterv_c(const void *sendbuf, const MPI_Count sendcounts[],
5         const MPI_Aint displs[], MPI_Datatype sendtype, void *recvbuf,
6         MPI_Count recvcount, MPI_Datatype recvtype, int root,
7         MPI_Comm comm, MPI_Request *request)

```

Fortran 2008 binding

```

9     MPI_Iscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
10        recvtype, root, comm, request, ierror)
11        TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
12        INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
13        TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
14        TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
15        INTEGER, INTENT(IN) :: recvcount, root
16        TYPE(MPI_Comm), INTENT(IN) :: comm
17        TYPE(MPI_Request), INTENT(OUT) :: request
18        INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20    MPI_Iscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
21        recvtype, root, comm, request, ierror) !(_c)
22        TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
23        INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*)
24        INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
25        TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
26        TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
27        INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
28        INTEGER, INTENT(IN) :: root
29        TYPE(MPI_Comm), INTENT(IN) :: comm
30        TYPE(MPI_Request), INTENT(OUT) :: request
31        INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

33    MPI_ISCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
34        RECVTYPE, ROOT, COMM, REQUEST, IERROR)
35        <type> SENDBUF(*), RECVBUF(*)
36        INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
37        COMM, REQUEST, IERROR

```

38
39 This call starts a nonblocking variant of MPI_SCATTERV (see Section 6.6).

40 41 6.12.5 Nonblocking Gather-to-all

```

42
43
44    MPI_IALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm,
45        request)
46
47    IN        sendbuf                starting address of send buffer (choice)

```

48

IN	sendcount	number of elements in send buffer (non-negative integer)	1 2
IN	sendtype	datatype of send buffer elements (handle)	3 4
OUT	recvbuf	address of receive buffer (choice)	5
IN	recvcount	number of elements received from any process (non-negative integer)	6 7
IN	recvtype	datatype of receive buffer elements (handle)	8 9
IN	comm	communicator (handle)	10
OUT	request	communication request (handle)	11 12

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)
```

```
int MPI_Iallgather_c(const void *sendbuf, MPI_Count sendcount,
                    MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
                    MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
```

Fortran 2008 binding

```
MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
               comm, request, ierror)
```

```
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
```

```
INTEGER, INTENT(IN) :: sendcount, recvcount
```

```
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
```

```
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
```

```
TYPE(MPI_Comm), INTENT(IN) :: comm
```

```
TYPE(MPI_Request), INTENT(OUT) :: request
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
               comm, request, ierror) !(_c)
```

```
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
```

```
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
```

```
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
```

```
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
```

```
TYPE(MPI_Comm), INTENT(IN) :: comm
```

```
TYPE(MPI_Request), INTENT(OUT) :: request
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran 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 6.7).

```

1 MPI_IALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcnts, displs, recvtype,
2                 comm, request)
3
4     IN        sendbuf          starting address of send buffer (choice)
5
6     IN        sendcount       number of elements in send buffer (non-negative
7                               integer)
8
9     IN        sendtype        datatype of send buffer elements (handle)
10
11    OUT       recvbuf          address of receive buffer (choice)
12
13    IN        recvcnts         nonnegative integer array (of length group size)
14                               containing the number of elements that are received
15                               from each process
16
17    IN        displs           integer array (of length group size). Entry i specifies
18                               the displacement (relative to recvbuf) at which to
19                               place the incoming data from process i
20
21    IN        recvtype         datatype of receive buffer elements (handle)
22
23    IN        comm             communicator (handle)
24
25    OUT       request          communication request (handle)

```

C binding

```

26 int MPI_Iallgatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
27                   void *recvbuf, const int recvcnts[], const int displs[],
28                   MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
29
30 int MPI_Iallgatherv_c(const void *sendbuf, MPI_Count sendcount,
31                     MPI_Datatype sendtype, void *recvbuf,
32                     const MPI_Count recvcnts[], const MPI_Aint displs[],
33                     MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)

```

Fortran 2008 binding

```

34 MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcnts, displs,
35                recvtype, comm, request, ierror)
36
37     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
38     INTEGER, INTENT(IN) :: sendcount
39     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
40     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
41     INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcnts(*), displs(*)
42     TYPE(MPI_Comm), INTENT(IN) :: comm
43     TYPE(MPI_Request), INTENT(OUT) :: request
44     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46 MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcnts, displs,
47                recvtype, comm, request, ierror) !(_c)
48
49     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
50     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
51     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
52     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
53     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcnts(*)

```



```

INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

MPI_IALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, REVCOUNTS, DISPLS,
                RECVTYPE, COMM, REQUEST, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, REVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
REQUEST, IERROR

```

This call starts a nonblocking variant of MPI_ALLGATHERV (see Section 6.7).

6.12.6 Nonblocking All-to-All Scatter/Gather

MPI_IALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcnt, 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	datatype of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcnt	number of elements received from any process (non-negative integer)
IN	recvtype	datatype of receive buffer elements (handle)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

C binding

```

int MPI_Ialltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 void *recvbuf, int recvcnt, MPI_Datatype recvtype,
                 MPI_Comm comm, MPI_Request *request)

```

```

int MPI_Ialltoall_c(const void *sendbuf, MPI_Count sendcount,
                   MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcnt,
                   MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)

```

Fortran 2008 binding

```

MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcnt, recvtype, comm,
              request, ierror)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
INTEGER, INTENT(IN) :: sendcount, recvcnt
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf

```

```

1     TYPE(MPI_Comm), INTENT(IN) :: comm
2     TYPE(MPI_Request), INTENT(OUT) :: request
3     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
5 MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtpe, comm,
6               request, ierror) !(_c)
7     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
8     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
9     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtpe
10    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
11    TYPE(MPI_Comm), INTENT(IN) :: comm
12    TYPE(MPI_Request), INTENT(OUT) :: request
13    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

14 MPI_IALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM,
15              REQUEST, IERROR)
16    <type> SENDBUF(*), RECVBUF(*)
17    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
18
19    This call starts a nonblocking variant of MPI_ALLTOALL (see Section 6.8).

```

```

20
21
22 MPI_IALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcoun, rdispls,
23               recvtpe, comm, request)
24
25    IN     sendbuf           starting address of send buffer (choice)
26    IN     sendcounts       nonnegative integer array (of length group size)
27                               specifying the number of elements to send to each
28                               rank
29    IN     sdispls          integer array (of length group size). Entry j specifies
30                               the displacement (relative to sendbuf) from which to
31                               take the outgoing data destined for process j
32
33    IN     sendtype         datatype of send buffer elements (handle)
34    OUT    recvbuf          address of receive buffer (choice)
35    IN     recvcoun         nonnegative integer array (of length group size)
36                               specifying the number of elements that can be
37                               received from each rank
38
39    IN     rdispls          integer array (of length group size). Entry i specifies
40                               the displacement (relative to recvbuf) at which to
41                               place the incoming data from process i
42
43    IN     recvtpe          datatype of receive buffer elements (handle)
44    IN     comm             communicator (handle)
45    OUT    request          communication request (handle)

```

C binding

```

46
47 int MPI_Ialltoallv(const void *sendbuf, const int sendcounts[],
48

```

```

const int sdispls[], MPI_Datatype sendtype, void *recvbuf,
const int recvcnts[], const int rdispls[],
MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
int MPI_Ialltoallv_c(const void *sendbuf, const MPI_Count sendcounts[],
const MPI_Aint sdispls[], MPI_Datatype sendtype, void *recvbuf,
const MPI_Count recvcnts[], const MPI_Aint rdispls[],
MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)

```

Fortran 2008 binding

```

MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcnts,
rdispls, recvtype, comm, request, ierror)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
recvcounts(*), rdispls(*)
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcnts,
rdispls, recvtype, comm, request, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*),
recvcounts(*)
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
rdispls(*)
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran 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 6.8).

```

MPI_IALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcnts, 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)

1	IN	sdispls	integer array (of length group size). Entry j specifies the displacement in bytes (relative to <code>sendbuf</code>) from which to take the outgoing data destined for process j (array of integers)
2			
3			
4			
5	IN	sendtypes	array of datatypes (of length group size). Entry j specifies the type of data to send to process j (array of handles)
6			
7			
8			
9	OUT	recvbuf	address of receive buffer (choice)
10	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)
11			
12			
13	IN	rdispls	integer array (of length group size). Entry i specifies the displacement in bytes (relative to <code>recvbuf</code>) at which to place the incoming data from process i (array of integers)
14			
15			
16			
17			
18	IN	recvtypes	array of datatypes (of length group size). Entry i specifies the type of data received from process i (array of handles)
19			
20			
21	IN	comm	communicator (handle)
22	OUT	request	communication request (handle)
23			

C binding

```

26 int MPI_Ialltoallw(const void *sendbuf, const int sendcounts[],
27                 const int sdispls[], const MPI_Datatype sendtypes[],
28                 void *recvbuf, const int recvcounts[], const int rdispls[],
29                 const MPI_Datatype recvtypes[], MPI_Comm comm,
30                 MPI_Request *request)

```

```

31 int MPI_Ialltoallw_c(const void *sendbuf, const MPI_Count sendcounts[],
32                    const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],
33                    void *recvbuf, const MPI_Count recvcounts[],
34                    const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],
35                    MPI_Comm comm, MPI_Request *request)

```

Fortran 2008 binding

```

38 MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
39              rdispls, recvtypes, comm, request, ierror)
40 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
41 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
42   recvcounts(*), rdispls(*)
43 TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*), recvtypes(*)
44 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
45 TYPE(MPI_Comm), INTENT(IN) :: comm
46 TYPE(MPI_Request), INTENT(OUT) :: request
47 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

48

```

MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
               rdispls, recvtypes, comm, request, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*),
               recvcounts(*)
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
               rdispls(*)
TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*), recvtypes(*)
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran 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 6.8).

6.12.7 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     datatype 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)
int MPI_Ireduce_c(const void *sendbuf, void *recvbuf, MPI_Count count,
                 MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
                 MPI_Request *request)

```

Fortran 2008 binding

```

1 MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request, ierror)
2   TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
3   TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
4   INTEGER, INTENT(IN) :: count, root
5   TYPE(MPI_Datatype), INTENT(IN) :: datatype
6   TYPE(MPI_Op), INTENT(IN) :: op
7   TYPE(MPI_Comm), INTENT(IN) :: comm
8   TYPE(MPI_Request), INTENT(OUT) :: request
9   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10

```

```

11 MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request, ierror)
12   !(_c)
13   TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
14   TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
15   INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
16   TYPE(MPI_Datatype), INTENT(IN) :: datatype
17   TYPE(MPI_Op), INTENT(IN) :: op
18   INTEGER, INTENT(IN) :: root
19   TYPE(MPI_Comm), INTENT(IN) :: comm
20   TYPE(MPI_Request), INTENT(OUT) :: request
21   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22

```

Fortran binding

```

23 MPI_IREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, IERROR)
24 <type> SENDBUF(*), RECVBUF(*)
25 INTEGER COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, IERROR
26

```

This call starts a nonblocking variant of MPI_REDUCE (see Section 6.9.1).

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 that 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.*)

6.12.8 Nonblocking All-Reduce

```

42 MPI_IALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm, request)
43

```

```

44   IN          sendbuf          starting address of send buffer (choice)
45

```

OUT	recvbuf	starting address of receive buffer (choice)	1
IN	count	number of elements in send buffer (non-negative integer)	2
			3
			4
IN	datatype	datatype of elements of send buffer (handle)	5
IN	op	operation (handle)	6
IN	comm	communicator (handle)	7
			8
OUT	request	communication request (handle)	9
			10
C binding			11
int	MPI_Iallreduce(const void *sendbuf, void *recvbuf, int count,		12
	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,		13
	MPI_Request *request)		14
			15
int	MPI_Iallreduce_c(const void *sendbuf, void *recvbuf, MPI_Count count,		16
	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,		17
	MPI_Request *request)		18
			19
Fortran 2008 binding			20
MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)			21
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf			22
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf			23
INTEGER, INTENT(IN) :: count			24
TYPE(MPI_Datatype), INTENT(IN) :: datatype			25
TYPE(MPI_Op), INTENT(IN) :: op			26
TYPE(MPI_Comm), INTENT(IN) :: comm			27
TYPE(MPI_Request), INTENT(OUT) :: request			28
INTEGER, OPTIONAL, INTENT(OUT) :: ierror			29
			30
MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)			31
!(<u>c</u>)			32
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf			33
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf			34
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count			35
TYPE(MPI_Datatype), INTENT(IN) :: datatype			36
TYPE(MPI_Op), INTENT(IN) :: op			37
TYPE(MPI_Comm), INTENT(IN) :: comm			38
TYPE(MPI_Request), INTENT(OUT) :: request			39
INTEGER, OPTIONAL, INTENT(OUT) :: ierror			40
			41
Fortran binding			42
MPI_IALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)			43
<type> SENDBUF(*), RECVBUF(*)			44
INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR			45
			46
This call starts a nonblocking variant of MPI_ALLREDUCE (see Section 6.9.6).			47
			48

6.12.9 Nonblocking Reduce-Scatter with Equal Blocks

```

MPI_IREDUCE_SCATTER_BLOCK(sendbuf, recvbuf, recvcnt, datatype, op, comm,
                           request)

```

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcnt	element count per block (non-negative integer)
IN	datatype	datatype 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 recvcnt, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
                             MPI_Request *request)

```

```

int MPI_Ireduce_scatter_block_c(const void *sendbuf, void *recvbuf,
                               MPI_Count recvcnt, MPI_Datatype datatype, MPI_Op op,
                               MPI_Comm comm, MPI_Request *request)

```

Fortran 2008 binding

```

MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcnt, datatype, op, comm,
                          request, ierror)

```

```

TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER, INTENT(IN) :: recvcnt
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, recvcnt, datatype, op, comm,
                          request, ierror) !(_c)

```

```

TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcnt
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

```


Fortran binding

```

MPI_IREDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, REVCOUNT, DATATYPE, OP, COMM,
    REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER REVCOUNT, DATATYPE, OP, COMM, REQUEST, IERROR

```

This call starts a nonblocking variant of `MPI_REDUCE_SCATTER_BLOCK` (see Section 6.10.1).

6.12.10 Nonblocking Reduce-Scatter

```

MPI_IREDUCE_SCATTER(sendbuf, recvbuf, recvcnts, datatype, op, comm, request)

```

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcnts	nonnegative integer array specifying the number of elements in result distributed to each process. This array must be identical on all calling processes.
IN	datatype	datatype 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 recvcnts[], MPI_Datatype datatype, MPI_Op op,
    MPI_Comm comm, MPI_Request *request)

int MPI_Ireduce_scatter_c(const void *sendbuf, void *recvbuf,
    const MPI_Count recvcnts[], MPI_Datatype datatype, MPI_Op op,
    MPI_Comm comm, MPI_Request *request)

```

Fortran 2008 binding

```

MPI_Ireduce_scatter(sendbuf, recvbuf, recvcnts, datatype, op, comm, request,
    ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcnts(*)
    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, recvcnts, datatype, op, comm, request,
    ierror) !(_c)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf

```

```

1     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
2     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcnts(*)
3     TYPE(MPI_Datatype), INTENT(IN) :: datatype
4     TYPE(MPI_Op), INTENT(IN) :: op
5     TYPE(MPI_Comm), INTENT(IN) :: comm
6     TYPE(MPI_Request), INTENT(OUT) :: request
7     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

9     MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, REQUEST,
10                        IERROR)
11     <type> SENDBUF(*), RECVBUF(*)
12     INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, REQUEST, IERROR

```

14 This call starts a nonblocking variant of MPI_REDUCE_SCATTER (see Section 6.10.2).

6.12.11 Nonblocking Inclusive Scan

```

19     MPI_ISCAN(sendbuf, recvbuf, count, datatype, op, comm, request)

```

21	IN	sendbuf	starting address of send buffer (choice)
22	OUT	recvbuf	starting address of receive buffer (choice)
23	IN	count	number of elements in input buffer (non-negative integer)
24			
25			
26	IN	datatype	datatype of elements of input buffer (handle)
27	IN	op	operation (handle)
28	IN	comm	communicator (handle)
29			
30	OUT	request	communication request (handle)

C binding

```

33     int MPI_Iscan(const void *sendbuf, void *recvbuf, int count,
34                 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
35                 MPI_Request *request)

```

```

36     int MPI_Iscan_c(const void *sendbuf, void *recvbuf, MPI_Count count,
37                   MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
38                   MPI_Request *request)

```

Fortran 2008 binding

```

41     MPI_Iscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
42     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
43     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
44     INTEGER, INTENT(IN) :: count
45     TYPE(MPI_Datatype), INTENT(IN) :: datatype
46     TYPE(MPI_Op), INTENT(IN) :: op
47     TYPE(MPI_Comm), INTENT(IN) :: comm
48     TYPE(MPI_Request), INTENT(OUT) :: request

```

```

    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Iscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror) !(_c)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER(KIND=MPI_COUNT_KIND), 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

```

Fortran 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 6.11).

6.12.12 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	datatype of elements of input buffer (handle)
IN	op	operation (handle)
IN	comm	intra-communicator (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)

```

```

int MPI_Iexscan_c(const void *sendbuf, void *recvbuf, MPI_Count count,
                  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
                  MPI_Request *request)

```

Fortran 2008 binding

```

MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype

```

```

1     TYPE(MPI_Op), INTENT(IN) :: op
2     TYPE(MPI_Comm), INTENT(IN) :: comm
3     TYPE(MPI_Request), INTENT(OUT) :: request
4     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
6 MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror) !(_c)
7     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
8     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
9     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
10    TYPE(MPI_Datatype), INTENT(IN) :: datatype
11    TYPE(MPI_Op), INTENT(IN) :: op
12    TYPE(MPI_Comm), INTENT(IN) :: comm
13    TYPE(MPI_Request), INTENT(OUT) :: request
14    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

16 MPI_IEXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)
17     <type> SENDBUF(*), RECVBUF(*)
18     INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR

```

This call starts a nonblocking variant of `MPI_EXSCAN` (see Section 6.11.2).

6.13 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 [52, 41] 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 intra-communicators and inter-communicators after completion. Likewise, upon completion, persistent collective reduction operations perform the same operation as their blocking and nonblocking counterparts, and the same restrictions and recommendations on reduction orders apply (see also Section 6.9.1).

Initialization calls for MPI persistent collective operations are nonlocal and follow all the existing rules for collective operations, in particular ordering; programs that do not conform to these restrictions are erroneous. After initialization, all arrays associated with input arguments (such as arrays of counts, displacements, and datatypes in the vector versions of the collectives) must not be modified until the corresponding persistent request is freed with `MPI_REQUEST_FREE`.

According to the definitions in Section 2.4.2, the persistent collective initialization procedures are incomplete. They are also nonlocal procedures because they may or may not return before they are called in all MPI processes of the process group associated with the specified communicator.

Advice to users. This is one of the exceptions in which incomplete procedures are nonlocal and therefore blocking. (*End of advice to users.*)

The `request` argument is an output argument that can be used zero or more times with `MPI_START` or `MPI_STARTALL` in order to start the collective operation. The `request` is initially inactive after the initialization call. Once initialized, persistent collective operations can be started in any order and the order can differ among processes in the communicator.

Rationale. All ordering requirements that an implementation may need to match up collective operations across the communicator are achieved through the ordering requirements of the initialization functions. This enables out-of-order starts for the persistent operations, and particularly supports their use in `MPI_STARTALL`. (*End of rationale.*)

Advice to implementors. An MPI implementation should do no worse than duplicating the communicator during the initialization function, caching the input arguments, and calling the appropriate nonblocking collective function, using the cached arguments, during `MPI_START`. High-quality implementations should be able to amortize setup costs and further optimize by taking advantage of early-binding, such as efficient and effective pre-allocation of certain resources and algorithm selection. (*End of advice to implementors.*)

A request must be inactive when it is started. Starting the operation makes the request active. Once any process starts a persistent collective operation, it must complete that operation and all other processes in the communicator must eventually start (and complete) the same persistent collective operation. Persistent collective operations cannot be matched with blocking or nonblocking collective operations. Completion of a persistent collective operation makes the corresponding request inactive. After starting a persistent collective operation, all associated send buffers must not be modified and all associated receive buffers must not be accessed until the corresponding persistent request is completed.

Completing a persistent collective request, for example using `MPI_TEST` or `MPI_WAIT`, makes it inactive, but does not free the request. This is the same behavior as for persistent point-to-point requests. Inactive persistent collective requests can be freed using `MPI_REQUEST_FREE`. It is erroneous to free an active persistent collective request. Persistent collective operations cannot be canceled; it is erroneous to use `MPI_CANCEL` on a persistent collective request.

For every nonblocking collective communication operation in MPI, there is a corresponding persistent collective operation with the analogous API signature.

The collective persistent API signatures include an `info` object in order to support optimization hints and other information that may be nonstandard. Persistent collective operations may be optimized during communicator creation or by the initialization operation of an individual persistent collective. Note that communicator-scoped hints should be provided using `MPI_COMM_SET_INFO` while, for operation-scoped hints, they are supplied to the persistent collective communication initialization functions using the `info` argument.

6.13.1 Persistent Barrier Synchronization

MPI_BARRIER_INIT(comm, info, request)

IN	comm	communicator (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)

C binding

```
int MPI_Barrier_init(MPI_Comm comm, MPI_Info info, MPI_Request *request)
```

Fortran 2008 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
```

Fortran binding

```
MPI_BARRIER_INIT(COMM, INFO, REQUEST, IERROR)
  INTEGER COMM, INFO, REQUEST, IERROR
```

Creates a persistent collective communication request for the barrier operation.

6.13.2 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	datatype 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)
```

```
int MPI_Bcast_init_c(void *buffer, MPI_Count count, MPI_Datatype datatype,
  int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)
```

Fortran 2008 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

MPI_Bcast_init(buffer, count, datatype, root, comm, info, request, ierror)
    !(_c)
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
INTEGER, INTENT(IN) :: root
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Info), INTENT(IN) :: info
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran 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.

6.13.3 Persistent Gather

```

MPI_GATHER_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcnt, recvtpe, 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	datatype of send buffer elements (handle)	
OUT	recvbuf	address of receive buffer (choice, significant only at root)	
IN	recvcnt	number of elements for any single receive (non-negative integer, significant only at root)	
IN	recvtpe	datatype of recv buffer elements (handle, significant only at root)	
IN	root	rank of receiving process (integer)	
IN	comm	communicator (handle)	
IN	info	info argument (handle)	
OUT	request	communication request (handle)	

C binding

```

1 int MPI_Gather_init(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
2                   void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
3                   MPI_Comm comm, MPI_Info info, MPI_Request *request)
4
5
6
7
8
9

```

```

10 int MPI_Gather_init_c(const void *sendbuf, MPI_Count sendcount,
11                    MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
12                    MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,
13                    MPI_Request *request)
14
15
16
17
18
19
20
21

```

Fortran 2008 binding

```

22 MPI_Gather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
23               root, comm, info, request, ierror)
24
25 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
26 INTEGER, INTENT(IN) :: sendcount, recvcount, root
27 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
28 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
29 TYPE(MPI_Comm), INTENT(IN) :: comm
30 TYPE(MPI_Info), INTENT(IN) :: info
31 TYPE(MPI_Request), INTENT(OUT) :: request
32 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
35
36
37
38
39
40

```

```

41 MPI_Gather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
42               root, comm, info, request, ierror) !(_c)
43
44 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
45 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
46 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
47 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
48 INTEGER, INTENT(IN) :: root
49 TYPE(MPI_Comm), INTENT(IN) :: comm
50 TYPE(MPI_Info), INTENT(IN) :: info
51 TYPE(MPI_Request), INTENT(OUT) :: request
52 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
53
54
55
56
57
58
59
60

```

Fortran binding

```

61 MPI_GATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
62               ROOT, COMM, INFO, REQUEST, IERROR)
63
64 <type> SENDBUF(*), RECVBUF(*)
65 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO,
66 REQUEST, IERROR
67
68
69
70
71
72
73
74

```

Creates a persistent collective communication request for the gather operation.

```

75 MPI_GATHERV_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcoun, displs, recvtype,
76                root, comm, info, request)
77
78
79
80
81
82
83
84

```

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative integer)

IN	sendtype	datatype of send buffer elements (handle)	1
OUT	recvbuf	address of receive buffer (choice, significant only at root)	2
			3
			4
IN	recvcounts	nonnegative integer array (of length group size) containing the number of elements that are received from each process (significant only at root)	5
			6
			7
IN	displs	integer array (of length group size). Entry <i>i</i> specifies the displacement relative to <code>recvbuf</code> at which to place the incoming data from process <i>i</i> (significant only at root)	8
			9
			10
			11
IN	recvtype	datatype of recv buffer elements (handle, significant only at root)	12
			13
			14
IN	root	rank of receiving process (integer)	15
IN	comm	communicator (handle)	16
IN	info	info argument (handle)	17
			18
OUT	request	communication request (handle)	19
			20
			21
C binding			21
<code>int MPI_Gatherv_init(const void *sendbuf, int sendcount, MPI_Datatype sendtype,</code>			22
<code>void *recvbuf, const int recvcounts[], const int displs[],</code>			23
<code>MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,</code>			24
<code>MPI_Request *request)</code>			25
			26
<code>int MPI_Gatherv_init_c(const void *sendbuf, MPI_Count sendcount,</code>			27
<code>MPI_Datatype sendtype, void *recvbuf,</code>			28
<code>const MPI_Count recvcounts[], const MPI_Aint displs[],</code>			29
<code>MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,</code>			30
<code>MPI_Request *request)</code>			31
			32
Fortran 2008 binding			32
<code>MPI_Gatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,</code>			33
<code>recvtype, root, comm, info, request, ierror)</code>			34
<code>TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf</code>			35
<code>INTEGER, INTENT(IN) :: sendcount, root</code>			36
<code>TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype</code>			37
<code>TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf</code>			38
<code>INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)</code>			39
<code>TYPE(MPI_Comm), INTENT(IN) :: comm</code>			40
<code>TYPE(MPI_Info), INTENT(IN) :: info</code>			41
<code>TYPE(MPI_Request), INTENT(OUT) :: request</code>			42
<code>INTEGER, OPTIONAL, INTENT(OUT) :: ierror</code>			43
			44
<code>MPI_Gatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,</code>			45
<code>recvtype, root, comm, info, request, ierror) !(_c)</code>			46
<code>TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf</code>			47
<code>INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount</code>			48

```

1     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
2     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
3     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcnts(*)
4     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
5     INTEGER, INTENT(IN) :: root
6     TYPE(MPI_Comm), INTENT(IN) :: comm
7     TYPE(MPI_Info), INTENT(IN) :: info
8     TYPE(MPI_Request), INTENT(OUT) :: request
9     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

12 MPI_GATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
13                 RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)
14 <type> SENDBUF(*), RECVBUF(*)
15 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
16     COMM, INFO, REQUEST, IERROR

```

Creates a persistent collective communication request for the gather operation.

6.13.4 Persistent Scatter

```

23 MPI_SCATTER_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcnt, recvtype, root,
24                 comm, info, request)

```

25	IN	sendbuf	address of send buffer (choice, significant only at root)
27	IN	sendcount	number of elements sent to each process (non-negative integer, significant only at root)
29	IN	sendtype	datatype of send buffer elements (handle, significant only at root)
32	OUT	recvbuf	address of receive buffer (choice)
33	IN	recvcnt	number of elements in receive buffer (non-negative integer)
36	IN	recvtype	datatype of receive buffer elements (handle)
37	IN	root	rank of sending process (integer)
38	IN	comm	communicator (handle)
40	IN	info	info argument (handle)
41	OUT	request	communication request (handle)

C binding

```

44 int MPI_Scatter_init(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
45                    void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root,
46                    MPI_Comm comm, MPI_Info info, MPI_Request *request)

```

```

int MPI_Scatter_init_c(const void *sendbuf, MPI_Count sendcount,
                      MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcnt,
                      MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,
                      MPI_Request *request)

```

Fortran 2008 binding

```

MPI_Scatter_init(sendbuf, sendcount, sendtype, recvbuf, recvcnt, recvtype,
                root, comm, info, request, ierror)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
INTEGER, INTENT(IN) :: sendcount, recvcnt, root
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Info), INTENT(IN) :: info
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

MPI_Scatter_init(sendbuf, sendcount, sendtype, recvbuf, recvcnt, recvtype,
                root, comm, info, request, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcnt
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER, INTENT(IN) :: root
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Info), INTENT(IN) :: info
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

MPI_SCATTER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, REVCOUNT, RECVTYPE,
                ROOT, COMM, INFO, REQUEST, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, REVCOUNT, RECVTYPE, ROOT, COMM, INFO,
                REQUEST, IERROR

```

Creates a persistent collective communication request for the scatter operation.

```

MPI_SCATTERV_INIT(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcnt, recvtype,
                 root, comm, info, request)

```

IN	sendbuf	address of send buffer (choice, significant only at root)
IN	sendcounts	nonnegative integer array (of length group size) specifying the number of elements to send to each rank (significant only at root)

1	IN	displs	integer array (of length group size). Entry <i>i</i> specifies the displacement (relative to <code>sendbuf</code>) from which to take the outgoing data to process <i>i</i> (significant only at root)
2			
3			
4			
5	IN	sendtype	datatype of send buffer elements (handle, significant only at root)
6			
7			
8	OUT	recvbuf	address of receive buffer (choice, significant only at root)
9			
10	IN	recvcount	number of elements in receive buffer (non-negative integer)
11			
12	IN	recvtype	datatype of receive buffer elements (handle)
13			
14	IN	root	rank of sending process (integer)
15	IN	comm	communicator (handle)
16	IN	info	info argument (handle)
17			
18	OUT	request	communication request (handle)
19			

C binding

```

21 int MPI_Scatterv_init(const void *sendbuf, const int sendcounts[],
22                     const int displs[], MPI_Datatype sendtype, void *recvbuf,
23                     int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm,
24                     MPI_Info info, MPI_Request *request)
25
26 int MPI_Scatterv_init_c(const void *sendbuf, const MPI_Count sendcounts[],
27                        const MPI_Aint displs[], MPI_Datatype sendtype, void *recvbuf,
28                        MPI_Count recvcount, MPI_Datatype recvtype, int root,
29                        MPI_Comm comm, MPI_Info info, MPI_Request *request)

```

Fortran 2008 binding

```

31 MPI_Scatterv_init(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
32                  recvtype, root, comm, info, request, ierror)
33     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
34     INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
35     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
36     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
37     INTEGER, INTENT(IN) :: recvcount, root
38     TYPE(MPI_Comm), INTENT(IN) :: comm
39     TYPE(MPI_Info), INTENT(IN) :: info
40     TYPE(MPI_Request), INTENT(OUT) :: request
41     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43 MPI_Scatterv_init(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
44                  recvtype, root, comm, info, request, ierror) !(_c)
45     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
46     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*)
47     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
48     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype

```

```

TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
INTEGER, INTENT(IN) :: root
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Info), INTENT(IN) :: info
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran 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.

6.13.5 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	datatype of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements received from any process (non-negative integer)
IN	recvtype	datatype of receive buffer elements (handle)
IN	comm	communicator (handle)
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)
int MPI_Allgather_init_c(const void *sendbuf, MPI_Count sendcount,
                        MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
                        MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
                        MPI_Request *request)

```

Fortran 2008 binding

```

1 MPI_Allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
2                   comm, info, request, ierror)
3

```

```

4     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
5

```

```

6     INTEGER, INTENT(IN) :: sendcount, recvcount
7

```

```

8     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
9

```

```

10    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
11

```

```

12    TYPE(MPI_Comm), INTENT(IN) :: comm
13

```

```

14    TYPE(MPI_Info), INTENT(IN) :: info
15

```

```

16    TYPE(MPI_Request), INTENT(OUT) :: request
17

```

```

18    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19

```

```

20 MPI_Allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
21                   comm, info, request, ierror) !(_c)
22

```

```

23    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
24

```

```

25    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
26

```

```

27    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
28

```

```

29    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
30

```

```

31    TYPE(MPI_Comm), INTENT(IN) :: comm
32

```

```

33    TYPE(MPI_Info), INTENT(IN) :: info
34

```

```

35    TYPE(MPI_Request), INTENT(OUT) :: request
36

```

```

37    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38

```

Fortran binding

```

39 MPI_ALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
40                   COMM, INFO, REQUEST, IERROR)
41

```

```

42    <type> SENDBUF(*), RECVBUF(*)
43

```

```

44    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
45

```

```

46    IERROR
47

```

Creates a persistent collective communication request for the allgather operation.

```

48 MPI_ALLGATHERV_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcoun
49                    t, displs, recvtype,
50                    comm, info, request)
51

```

```

52    IN    sendbuf          starting address of send buffer (choice)
53

```

```

54    IN    sendcount       number of elements in send buffer (non-negative
55                          integer)
56

```

```

57    IN    sendtype        datatype of send buffer elements (handle)
58

```

```

59    OUT   recvbuf         address of receive buffer (choice)
60

```

```

61    IN    recvcoun
62          t               nonnegative integer array (of length group size)
63                          containing the number of elements that are received
64                          from each process
65

```

```

66    IN    displs          integer array (of length group size). Entry i specifies
67                          the displacement (relative to recvbuf) at which to
68                          place the incoming data from process i
69

```

```

70    IN    recvtype        datatype of receive buffer elements (handle)
71

```

IN	comm	communicator (handle)	1
IN	info	info argument (handle)	2
OUT	request	communication request (handle)	3
			4
			5

C binding

```

int MPI_Allgatherv_init(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf, const int recvcnts[],
    const int displs[], MPI_Datatype recvtype, MPI_Comm comm,
    MPI_Info info, MPI_Request *request)

```

```

int MPI_Allgatherv_init_c(const void *sendbuf, MPI_Count sendcount,
    MPI_Datatype sendtype, void *recvbuf,
    const MPI_Count recvcnts[], const MPI_Aint displs[],
    MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
    MPI_Request *request)

```

Fortran 2008 binding

```

MPI_Allgatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcnts, displs,
    recvtype, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    INTEGER, INTENT(IN) :: sendcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcnts(*), displs(*)
    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_init(sendbuf, sendcount, sendtype, recvbuf, recvcnts, displs,
    recvtype, comm, info, request, ierror) !(_c)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcnts(*)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran 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

```

1 Creates a persistent collective communication request for the allgather operation.
2

3 6.13.6 Persistent All-to-All Scatter/Gather 4

5
6 MPI_ALLTOALL_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm,
7 info, request)
8

9	IN	sendbuf	starting address of send buffer (choice)
10	IN	sendcount	number of elements sent to each process (non-negative integer)
11			
12	IN	sendtype	datatype of send buffer elements (handle)
13	OUT	recvbuf	address of receive buffer (choice)
14	IN	recvcount	number of elements received from any process (non-negative integer)
15			
16	IN	recvtype	datatype of receive buffer elements (handle)
17	IN	comm	communicator (handle)
18	IN	info	info argument (handle)
19			
20	IN	request	communication request (handle)
21	OUT	request	communication request (handle)
22			

23 C binding

24
25 int MPI_Alltoall_init(const void *sendbuf, int sendcount,
26 MPI_Datatype sendtype, void *recvbuf, int recvcount,
27 MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
28 MPI_Request *request)

29 int MPI_Alltoall_init_c(const void *sendbuf, MPI_Count sendcount,
30 MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
31 MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
32 MPI_Request *request)

33 Fortran 2008 binding

34 MPI_Alltoall_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
35 comm, info, request, ierror)

36 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
37 INTEGER, INTENT(IN) :: sendcount, recvcount
38 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
39 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
40 TYPE(MPI_Comm), INTENT(IN) :: comm
41 TYPE(MPI_Info), INTENT(IN) :: info
42 TYPE(MPI_Request), INTENT(OUT) :: request
43 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44

45 MPI_Alltoall_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
46 comm, info, request, ierror) !(_c)

47 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
48 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount


```

TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype           1
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf             2
TYPE(MPI_Comm), INTENT(IN) :: comm                          3
TYPE(MPI_Info), INTENT(IN) :: info                          4
TYPE(MPI_Request), INTENT(OUT) :: request                   5
INTEGER, OPTIONAL, INTENT(OUT) :: ierror                    6

```

Fortran binding

```

MPI_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
                  COMM, INFO, REQUEST, IERROR)                8
<type> SENDBUF(*), RECVBUF(*)                                11
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
IERROR                                                        13

```

Creates a persistent collective communication request for the alltoall operation.

```

MPI_ALLTOALLV_INIT(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcoun-
                    t, rdispls, recvtype, comm, info, request) 17

```

IN	sendbuf	starting address of send buffer (choice)	19
IN	sendcounts	nonnegative integer array (of length group size) specifying the number of elements to send to each rank	21
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	24
IN	sendtype	datatype of send buffer elements (handle)	27
OUT	recvbuf	address of receive buffer (choice)	29
IN	recvcoun-	nonnegative integer array (of length group size) specifying the number of elements that can be received from each rank	30
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	33
IN	recvtype	datatype of receive buffer elements (handle)	37
IN	comm	communicator (handle)	38
IN	info	info argument (handle)	39
OUT	request	communication request (handle)	41

C binding

```

int MPI_Alltoallv_init(const void *sendbuf, const int sendcounts[],
                      const int sdispls[], MPI_Datatype sendtype, void *recvbuf,
                      const int recvcoun-                      45
                      t, const int rdispls[],
                      MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
                      MPI_Request *request)                    48

```

```

1 int MPI_Alltoallv_init_c(const void *sendbuf, const MPI_Count sendcounts[],
2     const MPI_Aint sdispls[], MPI_Datatype sendtype, void *recvbuf,
3     const MPI_Count recvcnts[], const MPI_Aint rdispls[],
4     MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
5     MPI_Request *request)

```

Fortran 2008 binding

```

6 MPI_Alltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcnts,
7     rdispls, recvtype, comm, info, request, ierror)
8     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
9     INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
10    recvcnts(*), rdispls(*)
11    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
12    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
13    TYPE(MPI_Comm), INTENT(IN) :: comm
14    TYPE(MPI_Info), INTENT(IN) :: info
15    TYPE(MPI_Request), INTENT(OUT) :: request
16    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

17 MPI_Alltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcnts,
18    rdispls, recvtype, comm, info, request, ierror) !(_c)
19    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
20    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*),
21    recvcnts(*)
22    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
23    rdispls(*)
24    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
25    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
26    TYPE(MPI_Comm), INTENT(IN) :: comm
27    TYPE(MPI_Info), INTENT(IN) :: info
28    TYPE(MPI_Request), INTENT(OUT) :: request
29    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

30 MPI_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,
31    RDISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)
32    <type> SENDBUF(*), RECVBUF(*)
33    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
34    RECVTYPE, COMM, INFO, REQUEST, IERROR

```

Creates a persistent collective communication request for the alltoallv operation.

```

35 MPI_ALLTOALLW_INIT(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcnts, rdispls,
36    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 <code>sendbuf</code>) from which to take the outgoing data destined for process j (array of integers)	1 2 3 4
IN	sendtypes	Array of datatypes (of length group size). Entry j specifies the type of data to send to process j (array of handles)	5 6 7 8
OUT	recvbuf	address of receive buffer (choice)	9
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)	10 11 12
IN	rdispls	integer array (of length group size). Entry i specifies the displacement in bytes (relative to <code>recvbuf</code>) at which to place the incoming data from process i (array of integers)	13 14 15 16 17
IN	recvtypes	array of datatypes (of length group size). Entry i specifies the type of data received from process i (array of handles)	18 19 20
IN	comm	communicator (handle)	21
IN	info	info argument (handle)	22 23
OUT	request	communication request (handle)	24 25
C binding			26
<code>int MPI_Alltoallw_init(const void *sendbuf, const int sendcounts[],</code>			27
<code>const int sdispls[], const MPI_Datatype sendtypes[],</code>			28
<code>void *recvbuf, const int recvcounts[], const int rdispls[],</code>			29
<code>const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info,</code>			30
<code>MPI_Request *request)</code>			31
<code>int MPI_Alltoallw_init_c(const void *sendbuf, const MPI_Count sendcounts[],</code>			32
<code>const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],</code>			33
<code>void *recvbuf, const MPI_Count recvcounts[],</code>			34
<code>const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],</code>			35
<code>MPI_Comm comm, MPI_Info info, MPI_Request *request)</code>			36 37
Fortran 2008 binding			38
<code>MPI_Alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,</code>			39
<code>recvcounts, rdispls, recvtypes, comm, info, request, ierror)</code>			40
<code>TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf</code>			41
<code>INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),</code>			42
<code>recvcounts(*), rdispls(*)</code>			43
<code>TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*), recvtypes(*)</code>			44
<code>TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf</code>			45
<code>TYPE(MPI_Comm), INTENT(IN) :: comm</code>			46
<code>TYPE(MPI_Info), INTENT(IN) :: info</code>			47
<code>TYPE(MPI_Request), INTENT(OUT) :: request</code>			48

```

1     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
2
3 MPI_Alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
4                   recvcounts, rdispls, recvtypes, comm, info, request, ierror)
5                   !(_c)
6     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
7     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*),
8                   recvcounts(*)
9     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
10                   rdispls(*)
11    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*), recvtypes(*)
12    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
13    TYPE(MPI_Comm), INTENT(IN) :: comm
14    TYPE(MPI_Info), INTENT(IN) :: info
15    TYPE(MPI_Request), INTENT(OUT) :: request
16    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

17 MPI_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
18                   RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST, IERROR)
19 <type> SENDBUF(*), RECVBUF(*)
20 INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*), RDISPLS(*),
21                   RECVTYPES(*), COMM, INFO, REQUEST, IERROR

```

22
23
24 Creates a persistent collective communication request for the alltoallw operation.

6.13.7 Persistent Reduce

```

25
26 MPI_REDUCE_INIT(sendbuf, recvbuf, count, datatype, op, root, comm, info, request)
27
28
29
30 IN     sendbuf      address of send buffer (choice)
31
32 OUT    recvbuf      address of receive buffer (choice, significant only at
33                   root)
34
35 IN     count        number of elements in send buffer (non-negative
36                   integer)
37
38 IN     datatype     datatype of elements of send buffer (handle)
39
40 IN     op           reduce operation (handle)
41
42 IN     root         rank of root process (integer)
43
44 IN     comm         communicator (handle)
45
46 IN     info         info argument (handle)
47
48 OUT    request      communication request (handle)

```

C binding

```

46 int MPI_Reduce_init(const void *sendbuf, void *recvbuf, int count,
47                   MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
48                   MPI_Info info, MPI_Request *request)

```

```

int MPI_Reduce_init_c(const void *sendbuf, void *recvbuf, MPI_Count count,
                    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
                    MPI_Info info, MPI_Request *request)

```

Fortran 2008 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

```

```

MPI_Reduce_init(sendbuf, recvbuf, count, datatype, op, root, comm, info,
               request, ierror) !(_c)

```

```

TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf

```

```

TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf

```

```

INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count

```

```

TYPE(MPI_Datatype), INTENT(IN) :: datatype

```

```

TYPE(MPI_Op), INTENT(IN) :: op

```

```

INTEGER, INTENT(IN) :: root

```

```

TYPE(MPI_Comm), INTENT(IN) :: comm

```

```

TYPE(MPI_Info), INTENT(IN) :: info

```

```

TYPE(MPI_Request), INTENT(OUT) :: request

```

```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran 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.

6.13.8 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     datatype of elements of send buffer (handle)

```

```

IN      op           operation (handle)

```

```

1      IN      comm      communicator (handle)
2      IN      info      info argument (handle)
3
4      OUT     request    communication request (handle)
5

```

C binding

```

7      int MPI_Allreduce_init(const void *sendbuf, void *recvbuf, int count,
8                          MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info,
9                          MPI_Request *request)
10
11     int MPI_Allreduce_init_c(const void *sendbuf, void *recvbuf, MPI_Count count,
12                             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info,
13                             MPI_Request *request)
14

```

Fortran 2008 binding

```

15     MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
16                       ierror)
17     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
18     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
19     INTEGER, INTENT(IN) :: count
20     TYPE(MPI_Datatype), INTENT(IN) :: datatype
21     TYPE(MPI_Op), INTENT(IN) :: op
22     TYPE(MPI_Comm), INTENT(IN) :: comm
23     TYPE(MPI_Info), INTENT(IN) :: info
24     TYPE(MPI_Request), INTENT(OUT) :: request
25     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27     MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
28                       ierror) !(_c)
29     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
30     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
31     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
32     TYPE(MPI_Datatype), INTENT(IN) :: datatype
33     TYPE(MPI_Op), INTENT(IN) :: op
34     TYPE(MPI_Comm), INTENT(IN) :: comm
35     TYPE(MPI_Info), INTENT(IN) :: info
36     TYPE(MPI_Request), INTENT(OUT) :: request
37     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38

```

Fortran binding

```

39     MPI_ALLREDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,
40                       IERROR)
41     <type> SENDBUF(*), RECVBUF(*)
42     INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
43

```

Creates a persistent collective communication request for the allreduce operation.

```

44
45
46
47
48

```

6.13.9 Persistent Reduce-Scatter with Equal Blocks

MPI_REDUCE_SCATTER_BLOCK_INIT(sendbuf, recvbuf, recvcnt, datatype, op, comm, info, request)

IN	sendbuf	starting address of send buffer (choice)	1
OUT	recvbuf	starting address of receive buffer (choice)	2
IN	recvcnt	element count per block (non-negative integer)	3
IN	datatype	datatype of elements of send and receive buffers (handle)	4
IN	op	operation (handle)	5
IN	comm	communicator (handle)	6
IN	info	info argument (handle)	7
OUT	request	communication request (handle)	8

C binding

```
int MPI_Reduce_scatter_block_init(const void *sendbuf, void *recvbuf,
    int recvcnt, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
    MPI_Info info, MPI_Request *request)
```

```
int MPI_Reduce_scatter_block_init_c(const void *sendbuf, void *recvbuf,
    MPI_Count recvcnt, MPI_Datatype datatype, MPI_Op op,
    MPI_Comm comm, MPI_Info info, MPI_Request *request)
```

Fortran 2008 binding

```
MPI_Reduce_scatter_block_init(sendbuf, recvbuf, recvcnt, datatype, op, comm,
    info, request, ierror)
```

```
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
```

```
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
```

```
INTEGER, INTENT(IN) :: recvcnt
```

```
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_block_init(sendbuf, recvbuf, recvcnt, datatype, op, comm,
    info, request, ierror) !(_c)
```

```
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
```

```
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
```

```
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcnt
```

```
TYPE(MPI_Datatype), INTENT(IN) :: datatype
```

```
TYPE(MPI_Op), INTENT(IN) :: op
```

```
TYPE(MPI_Comm), INTENT(IN) :: comm
```

```
TYPE(MPI_Info), INTENT(IN) :: info
```

```

1     TYPE(MPI_Request), INTENT(OUT) :: request
2     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

4 MPI_REDUCE_SCATTER_BLOCK_INIT(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
5                               INFO, REQUEST, IERROR)
6     <type> SENDBUF(*), RECVBUF(*)
7     INTEGER RECVCOUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
8

```

9 Creates a persistent collective communication request for the reduce-scatter with equal
10 blocks operation.

6.13.10 Persistent Reduce-Scatter

```

15 MPI_REDUCE_SCATTER_INIT(sendbuf, recvbuf, recvcounts, datatype, op, comm, info,
16                          request)

```

18	IN	sendbuf	starting address of send buffer (choice)
19	OUT	recvbuf	starting address of receive buffer (choice)
20	IN	recvcounts	nonnegative integer array specifying the number of
21			elements in result distributed to each process. This
22			array must be identical on all calling processes.
23			
24	IN	datatype	datatype of elements of input buffer (handle)
25	IN	op	operation (handle)
26	IN	comm	communicator (handle)
27	IN	info	info argument (handle)
28	IN	request	communication request (handle)
29	OUT	request	communication request (handle)

C binding

```

32 int MPI_Reduce_scatter_init(const void *sendbuf, void *recvbuf,
33                            const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
34                            MPI_Comm comm, MPI_Info info, MPI_Request *request)
35
36 int MPI_Reduce_scatter_init_c(const void *sendbuf, void *recvbuf,
37                              const MPI_Count recvcounts[], MPI_Datatype datatype, MPI_Op op,
38                              MPI_Comm comm, MPI_Info info, MPI_Request *request)

```

Fortran 2008 binding

```

40 MPI_Reduce_scatter_init(sendbuf, recvbuf, recvcounts, datatype, op, comm, info,
41                          request, ierror)
42     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
43     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
44     INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
45     TYPE(MPI_Datatype), INTENT(IN) :: datatype
46     TYPE(MPI_Op), INTENT(IN) :: op
47     TYPE(MPI_Comm), INTENT(IN) :: comm
48

```



```

TYPE(MPI_Info), INTENT(IN) :: info
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Reduce_scatter_init(sendbuf, recvbuf, recvcnts, datatype, op, comm, info,
    request, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcnts(*)
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

```

Fortran 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.

6.13.11 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   datatype 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)

```

```

int MPI_Scan_init_c(const void *sendbuf, void *recvbuf, MPI_Count count,
    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info,
    MPI_Request *request)

```

Fortran 2008 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

```

```

MPI_Scan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
              ierror) !(_c)

```

```

TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf

```

```

TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf

```

```

INTEGER(KIND=MPI_COUNT_KIND), 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

```

Fortran 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.

6.13.12 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     datatype of elements of input buffer (handle)

```

```

IN      op           operation (handle)

```

```

IN      comm         intra-communicator (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)
```

```
int MPI_Exscan_init_c(const void *sendbuf, void *recvbuf, MPI_Count count,
                     MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info,
                     MPI_Request *request)
```

Fortran 2008 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
```

```
MPI_Exscan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
                ierror) !(_c)
```

```
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
```

```
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
```

```
INTEGER(KIND=MPI_COUNT_KIND), 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
```

Fortran 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.

6.14 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 intra-communicators.

Example 6.25. The following is erroneous.

```

1  /* ----- THIS EXAMPLE IS ERRONEOUS ----- */
2  switch(rank) {
3
4      case 0:
5          MPI_Bcast(buf1, count, type, 0, comm);
6          MPI_Bcast(buf2, count, type, 1, comm);
7          break;
8
9      case 1:
10         MPI_Bcast(buf2, count, type, 1, comm);
11         MPI_Bcast(buf1, count, type, 0, comm);
12         break;
13     }

```

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 6.26. The following is erroneous.

```

18 /* ----- THIS EXAMPLE IS ERRONEOUS ----- */
19 switch(rank) {
20
21     case 0:
22         MPI_Bcast(buf1, count, type, 0, comm0);
23         MPI_Bcast(buf2, count, type, 2, comm2);
24         break;
25
26     case 1:
27         MPI_Bcast(buf1, count, type, 1, comm1);
28         MPI_Bcast(buf2, count, type, 0, comm0);
29         break;
30
31     case 2:
32         MPI_Bcast(buf1, count, type, 2, comm2);
33         MPI_Bcast(buf2, count, type, 1, comm1);
34         break;
35     }

```

Assume that the group of `comm0` is $\{0,1\}$, of `comm1` is $\{1, 2\}$ and of `comm2` is $\{2,0\}$. If the broadcast is a synchronizing operation, then there is a cyclic dependency: the broadcast in `comm2` completes only after the broadcast in `comm0`; the broadcast in `comm0` completes only after the broadcast in `comm1`; and the broadcast in `comm1` completes only after the broadcast in `comm2`. Thus, the code will deadlock. Collective operations must be executed in an order so that no cyclic dependencies occur. Nonblocking collective operations can alleviate this issue.

Example 6.27. The following is erroneous.

```

42 /* ----- THIS EXAMPLE IS ERRONEOUS ----- */
43 switch(rank) {
44
45     case 0:
46         MPI_Bcast(buf1, count, type, 0, comm);
47         MPI_Send(buf2, count, type, 1, tag, comm);
48     }

```

```

        break;
    case 1:
        MPI_Recv(buf2, count, type, 0, tag, comm, status);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}

```

Process zero executes a broadcast, followed by a blocking send operation. Process one first executes a blocking receive that matches the send, followed by broadcast call that matches the broadcast of process zero. This program may deadlock. The broadcast call on process zero *may* block until process one executes the matching broadcast call, so that the send is not executed. Process one will definitely block on the receive and so, in this case, never executes the broadcast.

The relative order of execution of collective operations and point-to-point operations should be such, so that even if the collective operations and the point-to-point operations are synchronizing, no deadlock will occur.

Example 6.28. An unsafe, nondeterministic program.

```

switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
        break;
    case 1:
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
        break;
    case 2:
        MPI_Send(buf2, count, type, 1, tag, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}

```

All three processes participate in a broadcast. Process 0 sends a message to process 1 after the broadcast, and process 2 sends a message to process 1 before the broadcast. Process 1 receives before and after the broadcast, with a wildcard source argument.

Two possible executions of this program, with different matchings of sends and receives, are illustrated in Figure 6.12. Note that the second execution has the peculiar effect that a send executed after the broadcast is received at another node before the broadcast. This example illustrates the fact that one should not rely on collective communication functions to have particular synchronization effects. A program that works correctly only when the first execution occurs (only when broadcast is synchronizing) is erroneous.

Finally, in multithreaded implementations, one can have more than one, concurrently executing, collective communication initialization call at an MPI process. In these situations, it is the user's responsibility to ensure that the same communicator is not used concurrently by two different collective communication initialization calls at the same MPI process. Collective communication initialization calls include all calls for blocking collective operations, all initiation calls for nonblocking collective operations, and all initialization

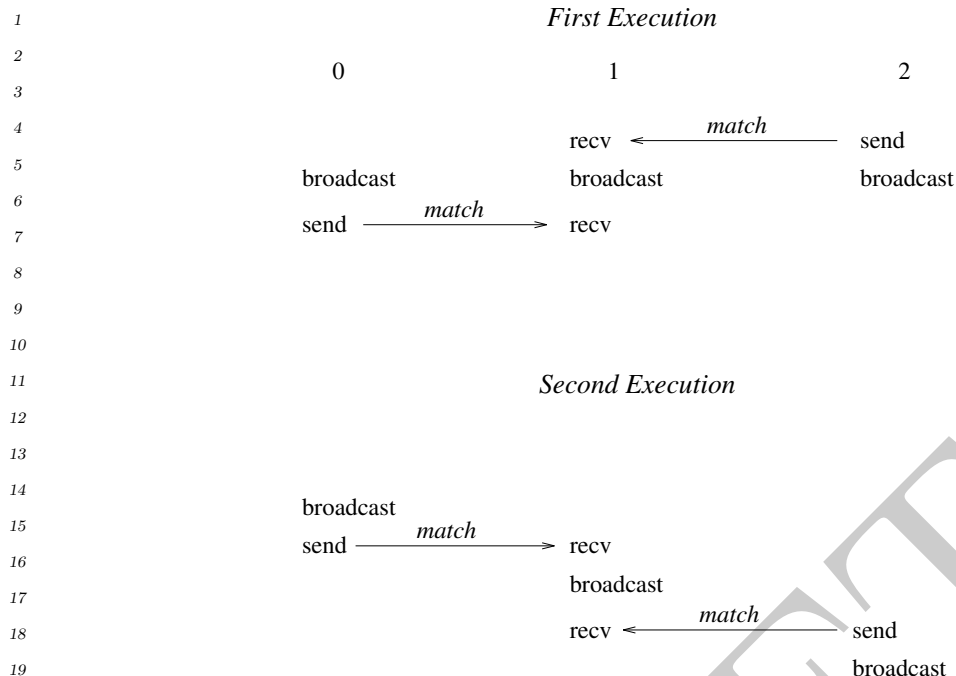


Figure 6.12: A race condition causes nondeterministic matching of sends and receives. One cannot rely on synchronization from a broadcast to make the program deterministic.

calls for persistent collective operations.

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 6.29. 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 6.30. The starting order of collective operations on a particular communicator defines their matching. The following example shows an erroneous matching of different collective operations on the same communicator.

```

/* ----- THIS EXAMPLE IS ERRONEOUS ----- */
MPI_Request req;
switch(rank) {
  case 0:
    /* erroneous matching */
    MPI_Ibarrier(comm, &req);
    MPI_Bcast(buf1, count, type, 0, comm);
    MPI_Wait(&req, MPI_STATUS_IGNORE);
    break;
  case 1:
    /* erroneous matching */
    MPI_Bcast(buf1, count, type, 0, comm);
    MPI_Ibarrier(comm, &req);
    MPI_Wait(&req, MPI_STATUS_IGNORE);
    break;
}

```

This ordering would match MPI_Ibarrier on rank 0 with MPI_Bcast on rank 1, which is erroneous and the program behavior is undefined. However, if such an order is required, the user must create different duplicate communicators and perform the operations on them. If started with two processes, the following program would be correct:

```

MPI_Request req;
MPI_Comm dupcomm;
MPI_Comm_dup(comm, &dupcomm);
switch(rank) {
  case 0:
    MPI_Ibarrier(comm, &req);
    MPI_Bcast(buf1, count, type, 0, dupcomm);
    MPI_Wait(&req, MPI_STATUS_IGNORE);
    break;
  case 1:
    MPI_Bcast(buf1, count, type, 0, dupcomm);
    MPI_Ibarrier(comm, &req);
    MPI_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

1 be used as an equivalent to tags. However, communicator construction might induce
 2 overheads so that this should be used carefully. (*End of advice to users.*)
 3

4 **Example 6.31.** Nonblocking collective operations can rely on similar progress rules as
 5 nonblocking point-to-point operations. Thus, if started with two processes, the following
 6 program is a valid MPI program and is guaranteed to terminate:
 7

```
8 MPI_Request req;
9
10 switch(rank) {
11     case 0:
12         MPI_Ibarrier(comm, &req);
13         MPI_Wait(&req, MPI_STATUS_IGNORE);
14         MPI_Send(buf, count, dtype, 1, tag, comm);
15         break;
16     case 1:
17         MPI_Ibarrier(comm, &req);
18         MPI_Recv(buf, count, dtype, 0, tag, comm, MPI_STATUS_IGNORE);
19         MPI_Wait(&req, MPI_STATUS_IGNORE);
20         break;
21 }
22
```

23 The MPI library must *progress* the barrier in the MPI_Recv call. Thus, the MPI_Wait call in
 24 rank 0 will eventually complete, which enables the matching MPI_Send so all calls eventually
 25 return.
 26

27 **Example 6.32.** Blocking and nonblocking collective operations do not match. The
 28 following example is erroneous.
 29

```
30 /* ----- THIS EXAMPLE IS ERRONEOUS ----- */
31 MPI_Request req;
32
33 switch(rank) {
34     case 0:
35         /* erroneous false matching of Alltoall and Ialltoall */
36         MPI_Ialltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm, &req);
37         MPI_Wait(&req, MPI_STATUS_IGNORE);
38         break;
39     case 1:
40         /* erroneous false matching of Alltoall and Ialltoall */
41         MPI_Alltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm);
42         break;
43 }
44
```

45 **Example 6.33.** Collective and point-to-point requests can be mixed in functions that
 46 enable multiple completions. If started with two processes, the following program is valid.
 47

```
48 MPI_Request reqs[2];
49
50 switch(rank) {
51     case 0:
52         MPI_Ibarrier(comm, &reqs[0]);
53
```



```

    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 6.34. Multiple nonblocking collective operations can be outstanding on a single communicator and match in order.

```

MPI_Request reqs[3];

compute(buf1);
MPI_Ibcast(buf1, count, type, 0, comm, &reqs[0]);
compute(buf2);
MPI_Ibcast(buf2, count, type, 0, comm, &reqs[1]);
compute(buf3);
MPI_Ibcast(buf3, count, type, 0, comm, &reqs[2]);
MPI_Waitall(3, reqs, MPI_STATUSES_IGNORE);

```

Advice to users. Pipelining and double-buffering techniques can efficiently be used to overlap computation and communication. However, having too many outstanding requests might have a negative impact on performance. (*End of advice to users.*)

Advice to implementors. The use of pipelining may generate many outstanding requests. A high-quality hardware-supported implementation with limited resources should be able to fall back to a software implementation if its resources are exhausted. In this way, the implementation could limit the number of outstanding requests only by the available memory. (*End of advice to implementors.*)

Example 6.35. Nonblocking collective operations can also be used to enable simultaneous collective operations on multiple overlapping communicators (see Figure 6.13). The following example is started with three processes and three communicators. The first communicator comm1 includes ranks 0 and 1, comm2 includes ranks 1 and 2, and comm3 spans ranks 0 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]);

```

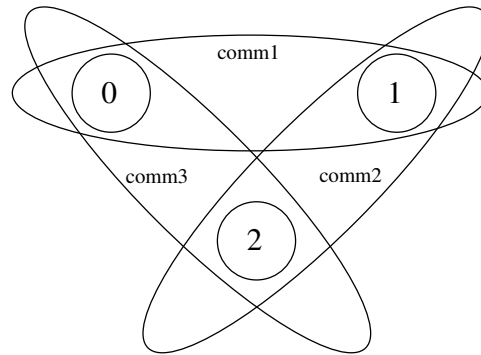


Figure 6.13: Example with overlapping communicators.

```

14 MPI_Iallreduce(sbuf2, rbuf2, count, dtype, MPI_SUM, comm2, &reqs[1]);
15 break;
16 case 2:
17 MPI_Iallreduce(sbuf2, rbuf2, count, dtype, MPI_SUM, comm2, &reqs[0]);
18 MPI_Iallreduce(sbuf3, rbuf3, count, dtype, MPI_SUM, comm3, &reqs[1]);
19 break;
20 }
21 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 6.36. The *progress* of multiple outstanding nonblocking collective operations is completely independent.

```

31 MPI_Request reqs[2];
32
33 compute(buf1);
34 MPI_Ibcast(buf1, count, type, 0, comm, &reqs[0]);
35 compute(buf2);
36 MPI_Ibcast(buf2, count, type, 0, comm, &reqs[1]);
37 MPI_Wait(&reqs[1], MPI_STATUS_IGNORE);
38 /* nothing is known about the status of the first bcast here */
39 MPI_Wait(&reqs[0], MPI_STATUS_IGNORE);

```

Finishing the second MPI_IBCAST is completely independent of the first one. This means that it is not guaranteed that the first broadcast operation is finished or even started after the second one is completed via reqs[1].

Chapter 7

Groups, Contexts, Communicators, and Caching

7.1 Introduction

This chapter introduces MPI features that support the development of parallel libraries. Parallel libraries are needed to encapsulate the distracting complications inherent in parallel implementations of key algorithms. They help to ensure consistent correctness of such procedures, and provide a “higher level” of portability than MPI itself can provide. As such, libraries prevent each programmer from repeating the work of defining consistent data structures, data layouts, and methods that implement key algorithms (such as matrix operations). Since the best libraries come with several variations on parallel systems (different data layouts, different strategies depending on the size of the system or problem, or type of floating point), this too needs to be hidden from the user.

We refer the reader to [4] and [62] for further information on writing libraries in MPI, using the features described in this chapter.

7.1.1 Features Needed to Support Libraries

The key features needed to support the creation of robust parallel libraries are as follows:

- Safe communication space, that guarantees that libraries can communicate as they need to, without conflicting with communication extraneous to the library,
- Group scope for collective operations, that allow libraries to avoid unnecessarily synchronizing uninvolved MPI processes (potentially running unrelated code),
- Abstract naming of MPI processes 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 MPI 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 MPI processes, to house abstract naming of MPI processes, and to store adornments.

7.1.2 MPI’s Support for Libraries

The corresponding concepts that MPI provides, specifically to support robust libraries, are as follows:

- 1 • **Contexts** of communication,
- 2
- 3 • **Groups** of MPI processes,
- 4
- 5 • **Virtual topologies**,
- 6
- 7 • **Attribute caching**,
- 8
- 9 • **Communicators**.

10 **Communicators** (see [22, 60, 64]) encapsulate all of these ideas in order to provide the
11 appropriate scope for all communication operations in MPI. Communicators are divided
12 into two kinds: intra-communicators for operations within a single group of MPI processes
13 and inter-communicators for operations between two groups of MPI processes.

14 **Caching.** Communicators (see below) provide a “caching” mechanism that allows one
15 to associate new attributes with communicators, on par with MPI built-in features. This
16 can be used by advanced users to adorn communicators further, and by MPI to implement
17 some communicator functions. For example, the virtual-topology functions described in
18 Chapter 8 are likely to be supported this way.

19

20 **Groups.** Groups define an ordered collection of MPI processes, each with a rank, and it
21 is this group that defines the low-level names (ranks) for communication. Thus, groups
22 define a scope for MPI process names in point-to-point communication. In addition, groups
23 define the scope of collective operations. Groups may be manipulated separately from
24 communicators in MPI, but only communicators can be used in communication operations.

25

26 **Intra-Communicators.** The most commonly used means for message-passing in MPI is
27 via intra-communicators. Intra-communicators contain an instance of a group, contexts of
28 communication for both point-to-point and collective communication, and the ability to
29 include virtual topology and other attributes. These features work as follows:

- 30
- 31 • **Contexts** provide the ability to have separate safe “universes” of message-passing in
32 MPI. A context is akin to an additional tag that differentiates messages. The system
33 manages this differentiation process. The use of separate communication contexts
34 by distinct libraries (or distinct library invocations) insulates communication internal
35 to the library execution from external communication. This allows the invocation of
36 the library even if there are pending communications on “other” communicators, and
37 avoids the need to synchronize entry or exit into library code. Pending point-to-point
38 communications are also guaranteed not to interfere with collective communications
39 within a single communicator.
- 40
- 41 • **Groups** define the participants in the communication (see above) of a communicator.
- 42
- 43 • A **virtual topology** defines a special mapping of the MPI processes ranks in a group to
44 and from a topology. Special constructors for communicators are defined in Chapter 8
45 to provide this feature. Intra-communicators as described in this chapter do not have
46 topologies.
- 47 • **Attributes** define the local information that the user or library has added to a com-
48 municator for later reference.

Advice to users. The practice in many communication libraries is that there is a unique, predefined communication universe that includes all MPI processes available when the parallel program is initiated; the MPI 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 MPI processes. When using the World Model (Section 11.2), this practice can be followed in MPI by using the predefined communicator `MPI_COMM_WORLD`. (*End of advice to users.*)

Inter-Communicators. The discussion has dealt so far with **intra-communication**: communication within a group. MPI also supports **inter-communication**: communication between two nonoverlapping 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 MPI 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 operation in the local group is always matched by a receive operation in the remote group, and vice versa. The system manages this differentiation process. The use of separate communication contexts by distinct libraries (or distinct library invocations) insulates communication internal to the library execution from external communication. This allows the invocation of the library even if there are pending communications on “other” communicators, and avoids the need to synchronize entry or exit into library code.
- A local and remote group specify the recipients and destinations for an inter-communicator.
- Virtual topology is undefined for an inter-communicator.
- As before, attributes cache defines the local information that the user or library has added to a communicator for later reference.

MPI provides mechanisms for creating and manipulating inter-communicators. They are used for point-to-point and collective communication in a 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.

7.2 Basic Concepts

In this section, we turn to a more formal definition of the concepts introduced above.

7.2.1 Groups

A **group** is an ordered set of MPI process identifiers (henceforth MPI processes); MPI processes are implementation-dependent objects. Each MPI process in a group is associated with an integer **rank**. Ranks are consecutive and start from zero. Groups are represented by opaque **group objects**, and hence cannot be directly transferred from one MPI 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 procedures; the latter is not a valid input value for an input argument. (*End of advice to users.*)

Advice to implementors. 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 MPI processes. Such implementations are possible with MPI. (*End of advice to implementors.*)

7.2.2 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 MPI 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 do not 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.*)

7.2.3 Intra-Communicators

Intra-communicators bring together the concepts of group and context. To support implementation-specific optimizations, and application topologies (defined in the next chapter, Chapter 8), communicators may also “cache” additional information (see Section 7.7). MPI communication operations reference communicators to determine the scope and the “communication universe” in which a point-to-point or collective operation is to operate.

Each communicator contains a group of valid participants; this group always includes the local MPI process. The source and destination of a message are identified by MPI process ranks within that group.

For collective communication, the intra-communicator specifies the set of MPI 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 MPI process addressing through ranks.

Intra-communicators are represented by opaque **intra-communicator objects**, and hence cannot be directly transferred from one MPI process to another.

7.2.4 Predefined Intra-Communicators

When using the World Model (Section 11.2) for MPI initialization, an initial intra-communicator `MPI_COMM_WORLD` of all MPI processes the local MPI 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 MPI process itself. When using the Sessions Model (Section 11.3) for initialization of MPI resources, `MPI_COMM_WORLD` and `MPI_COMM_SELF` are not valid for use as a communicator. See the discussion concerning use of MPI named constants in 2.5.4 for valid uses of `MPI_COMM_WORLD` and `MPI_COMM_SELF` prior to initialization of MPI. See also the discussion concerning interoperability of the World Model and Sessions Model in Section 11.1.

The predefined constant `MPI_COMM_NULL` is the value used for invalid communicator handles.

In a static-process-model implementation of MPI, all MPI processes that participate in the computation are available after MPI is initialized. For this case, `MPI_COMM_WORLD` is a communicator of all MPI processes available for the computation; this communicator has the same value in all MPI processes. In an implementation of MPI where MPI processes can dynamically join an MPI execution, it may be the case that an MPI process starts an MPI computation without having access to all other MPI processes. In such situations, `MPI_COMM_WORLD` is a communicator incorporating all MPI processes with which the joining MPI process can immediately communicate. Therefore, `MPI_COMM_WORLD` may simultaneously represent disjoint groups in different MPI processes.

All MPI implementations are required to provide the `MPI_COMM_WORLD` communicator. It cannot be deallocated during the life of an MPI 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 MPI process rank in `MPI_COMM_WORLD` and its (machine-dependent) absolute address. Neither does MPI specify the function of the host MPI process, if any. Other implementation-dependent, predefined communicators may also be provided.

7.3 Group Management

This section describes the manipulation of MPI process groups. These operations are local.

7.3.1 Group Accessors

MPI_GROUP_SIZE(group, size)

IN	group	group (handle)
OUT	size	number of MPI processes in the group (integer)

C binding

```
int MPI_Group_size(MPI_Group group, int *size)
```

Fortran 2008 binding

```
MPI_Group_size(group, size, ierror)
  TYPE(MPI_Group), INTENT(IN) :: group
  INTEGER, INTENT(OUT) :: size
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran 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 MPI process in group, or MPI_UNDEFINED if the MPI process is not a member (integer)

C binding

```
int MPI_Group_rank(MPI_Group group, int *rank)
```

Fortran 2008 binding

```
MPI_Group_rank(group, rank, ierror)
  TYPE(MPI_Group), INTENT(IN) :: group
  INTEGER, INTENT(OUT) :: rank
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_GROUP_RANK(GROUP, RANK, IERROR)
  INTEGER GROUP, RANK, IERROR
```


MPI_GROUP_TRANSLATE_RANKS(group1, n, ranks1, group2, ranks2)			1
IN	group1	group1 (handle)	2
IN	n	number of elements in ranks1 and ranks2 arrays (integer)	3
IN	ranks1	array of zero or more valid ranks in group1	4
IN	group2	group2 (handle)	5
OUT	ranks2	array of corresponding ranks in group2, MPI_UNDEFINED when no correspondence exists.	6

C binding

```
int MPI_Group_translate_ranks(MPI_Group group1, int n, const int ranks1[],
                             MPI_Group group2, int ranks2[])
```

Fortran 2008 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
```

Fortran 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 MPI processes in two different groups. For instance, if one knows the ranks of certain MPI 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)
```

Fortran 2008 binding

```
MPI_Group_compare(group1, group2, result, ierror)
  TYPE(MPI_Group), INTENT(IN) :: group1, group2
  INTEGER, INTENT(OUT) :: result
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```

MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR)
    INTEGER GROUP1, GROUP2, RESULT, IERROR

```

MPI_IDENT results if the group members and group order are exactly the same in both groups. This happens for instance if `group1` and `group2` are the same handle. MPI_SIMILAR results if the group members are the same but the order is different. MPI_UNEQUAL results otherwise.

7.3.2 Group Constructors

MPI provides two approaches to constructing groups. In the first approach, MPI procedures are provided to subset and superset existing groups. These constructors construct new groups from existing groups. In the second approach, a group is created using a session handle and associated process set. This second approach is available when using the Sessions Model. With both approaches, these are local operations, and distinct groups may be defined on different MPI processes; an MPI process may also define a group that does not include itself. Consistent definitions are required when groups are used as arguments in communicator creation functions. When using the World Model (Section 11.2) for MPI initialization, the base group, upon which all other groups are defined, is the group associated with the initial communicator MPI_COMM_WORLD (accessible through the function MPI_COMM_GROUP).

Rationale. In what follows, there is no group duplication function analogous to MPI_COMM_DUP, defined later in this chapter. There is no need for a group duplicator. A group, once created, can have several references to it by making copies of the handle. The following constructors address the need for subsets and supersets of existing groups. (*End of rationale.*)

Advice to implementors. Each group constructor behaves as if it returned a new group object. When this new group is a copy of an existing group, then one can avoid creating such new objects, using a reference-count mechanism. (*End of advice to implementors.*)

```

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)

```

Fortran 2008 binding

```

MPI_Comm_group(comm, group, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Group), INTENT(OUT) :: group
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran 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)
```

Fortran 2008 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
```

Fortran 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)
```

Fortran 2008 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
```

Fortran binding

```
MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
```

```

1 MPI_GROUP_DIFFERENCE(group1, group2, newgroup)
2     IN      group1          first group (handle)
3
4     IN      group2          second group (handle)
5
6     OUT     newgroup        difference group (handle)

```

C binding

```

8 int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
9                          MPI_Group *newgroup)

```

Fortran 2008 binding

```

12 MPI_Group_difference(group1, group2, newgroup, ierror)
13     TYPE(MPI_Group), INTENT(IN) :: group1, group2
14     TYPE(MPI_Group), INTENT(OUT) :: newgroup
15     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

17 MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)
18     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 MPI 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`.

```

35 MPI_GROUP_INCL(group, n, ranks, newgroup)

```

```

36     IN      group          group (handle)
37
38     IN      n              number of elements in array ranks (and size of
39                          newgroup) (integer)
40
41     IN      ranks          ranks of processes in group to appear in newgroup
42                          (array of integers)
43
44     OUT     newgroup        new group derived from above, in the order defined
45                          by ranks (handle)

```

C binding

```

47 int MPI_Group_incl(MPI_Group group, int n, const int ranks[],
48                   MPI_Group *newgroup)

```

Fortran 2008 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

```

Fortran 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` MPI processes in `group` with ranks `ranks[0], ..., ranks[n-1]`; the MPI process with rank `i` in `newgroup` is the MPI 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 <code>ranks</code> (integer)
IN	ranks	array of integer ranks of MPI processes in <code>group</code> not to appear in <code>newgroup</code>
OUT	newgroup	new group derived from above, preserving the order defined by <code>group</code> (handle)

C binding

```

int MPI_Group_excl(MPI_Group group, int n, const int ranks[],
  MPI_Group *newgroup)

```

Fortran 2008 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

```

Fortran binding

```

MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)
  INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR

```

The function `MPI_GROUP_EXCL` creates a group of MPI processes `newgroup` that is obtained by deleting from `group` those MPI processes with ranks `ranks[0], ..., ranks[n-1]`. The ordering of MPI 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`.

```

1 MPI_GROUP_RANGE_INCL(group, n, ranges, newgroup)
2   IN      group          group (handle)
3
4   IN      n              number of triplets in array ranges (integer)
5
6   IN      ranges         a one-dimensional array of integer triplets, of the
7                          form (first rank, last rank, stride) indicating ranks in
8                          group of MPI processes to be included in newgroup
9
10  OUT     newgroup       new group derived from above, in the order defined
11                          by ranges (handle)

```

C binding

```

12 int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],
13                          MPI_Group *newgroup)
14

```

Fortran 2008 binding

```

15 MPI_Group_range_incl(group, n, ranges, newgroup, ierror)
16   TYPE(MPI_Group), INTENT(IN) :: group
17   INTEGER, INTENT(IN) :: n, ranges(3, n)
18   TYPE(MPI_Group), INTENT(OUT) :: newgroup
19   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20

```

Fortran binding

```

21 MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)
22   INTEGER GROUP, N, RANGES(3, *), NEWGROUP, IERROR
23

```

If *ranges* consists of the triplets

$$(first_1, last_1, stride_1), \dots, (first_n, last_n, stride_n)$$

then *newgroup* consists of the sequence of MPI processes in *group* with ranks

$$first_1, first_1 + stride_1, \dots, first_1 + \left\lfloor \frac{last_1 - first_1}{stride_1} \right\rfloor stride_1, \dots,$$

$$first_n, first_n + stride_n, \dots, first_n + \left\lfloor \frac{last_n - first_n}{stride_n} \right\rfloor stride_n.$$

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)			1
IN	group	group (handle)	2
IN	n	number of triplets in array ranges (integer)	3
IN	ranges	a one-dimensional array of integer triplets, of the form (first rank, last rank, stride) indicating ranks in group of MPI processes to be excluded from the output group newgroup (array of integers)	4
OUT	newgroup	new group derived from above, preserving the order in group (handle)	5

C binding

```
int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],
                        MPI_Group *newgroup)
```

Fortran 2008 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
```

Fortran 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 whenever possible, as high-quality implementations will take advantage of this fact. (*End of advice to users.*)

Advice to implementors. The range operations should be implemented, if possible, without enumerating the group members, in order to obtain better scalability (time and space). (*End of advice to implementors.*)

```
MPI_GROUP_FROM_SESSION_PSET(session, pset_name, newgroup)
```

IN	session	session (handle)	45
IN	pset_name	name of process set to use to create the new group (string)	46

1 OUT newgroup new group derived from supplied session and process
2 set (handle)
3

4 **C binding**

5 int MPI_Group_from_session_pset(MPI_Session session, const char *pset_name,
6 MPI_Group *newgroup)
7

8 **Fortran 2008 binding**

9 MPI_Group_from_session_pset(session, pset_name, newgroup, ierror)
10 TYPE(MPI_Session), INTENT(IN) :: session
11 CHARACTER(LEN=*) , INTENT(IN) :: pset_name
12 TYPE(MPI_Group), INTENT(OUT) :: newgroup
13 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14

14 **Fortran binding**

15 MPI_GROUP_FROM_SESSION_PSET(SESSION, PSET_NAME, NEWGROUP, IERROR)
16 INTEGER SESSION, NEWGROUP, IERROR
17 CHARACTER*(*) PSET_NAME
18

19 The function MPI_GROUP_FROM_SESSION_PSET creates a group newgroup using the
20 provided session handle and process set. The process set name must be one returned from
21 an invocation of MPI_SESSION_GET_NTH_PSET using the supplied session handle. If the
22 pset_name does not exist, MPI_GROUP_NULL will be returned in the newgroup argument.
23 As with other group constructors, MPI_GROUP_FROM_SESSION_PSET is a local function.
24 See Section 11.3 for more information on sessions and process sets.
25

26 7.3.3 Group Destructors

29 MPI_GROUP_FREE(group)

30 INOUT group group (handle)
31

33 **C binding**

34 int MPI_Group_free(MPI_Group *group)

35 **Fortran 2008 binding**

36 MPI_Group_free(group, ierror)
37 TYPE(MPI_Group), INTENT(INOUT) :: group
38 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39

40 **Fortran binding**

41 MPI_GROUP_FREE(GROUP, IERROR)
42 INTEGER GROUP, IERROR
43

44 This operation marks a group object for deallocation. The handle
45 group is set to MPI_GROUP_NULL by the call. Any on-going operation using this group will
46 complete normally.

47 *Advice to implementors.* One can keep a reference count that is incremented
48 for each call to MPI_COMM_GROUP, MPI_COMM_CREATE,

MPI_COMM_DUP, MPI_COMM_IDUP, MPI_COMM_DUP_WITH_INFO, MPI_COMM_IDUP_WITH_INFO, MPI_COMM_SPLIT, MPI_COMM_SPLIT_TYPE, MPI_COMM_CREATE_GROUP, MPI_COMM_CREATE_FROM_GROUP, MPI_INTERCOMM_CREATE, and MPI_INTERCOMM_CREATE_FROM_GROUPS, and decremented for each call to MPI_GROUP_FREE or MPI_COMM_FREE; the group object is ultimately deallocated when the reference count drops to zero. (*End of advice to implementors.*)

7.4 Communicator Management

This section describes the manipulation of communicators in MPI. Operations that access communicators are local. Operations that create communicators are collective.

Advice to implementors. High-quality implementations should amortize the overheads associated with the creation of communicators (for the same group, or subsets thereof) over several calls, by allocating multiple contexts with one collective communication. (*End of advice to implementors.*)

7.4.1 Communicator Accessors

The following are all local operations.

MPI_COMM_SIZE(comm, size)

IN	comm	communicator (handle)
OUT	size	number of MPI processes in the group of comm (integer)

C binding

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

Fortran 2008 binding

```
MPI_Comm_size(comm, size, ierror)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, INTENT(OUT) :: size
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_COMM_SIZE(COMM, SIZE, IERROR)
  INTEGER COMM, SIZE, IERROR
```

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 MPI processes involved in a communicator. For MPI_COMM_WORLD, it indicates the total number of MPI

processes available unless the number of MPI processes has been changed by using the functions described in Chapter 11; note that the number of MPI 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 MPI process that calls it in the range from 0, . . . , size-1, where size is the return value of MPI_COMM_SIZE. (*End of advice to users.*)

MPI_COMM_RANK(comm, rank)

IN	comm	communicator (handle)
OUT	rank	rank of the calling MPI process in group of comm (integer)

C binding

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Fortran 2008 binding

```
MPI_Comm_rank(comm, rank, ierror)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, INTENT(OUT) :: rank
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran 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 MPI 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 supervisor/executor or manager/worker model, where one MPI process will play a supervisory role, and the other MPI processes will serve as compute nodes. In this framework, the two preceding calls are useful for determining the roles of the various MPI processes of a communicator. (*End of advice to users.*)

MPI_COMM_COMPARE(comm1, comm2, result)			1
IN	comm1	first communicator (handle)	2
IN	comm2	second communicator (handle)	3
OUT	result	result (integer)	4

C binding

```
int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)
```

Fortran 2008 binding

```
MPI_Comm_compare(comm1, comm2, result, ierror)
```

```
TYPE(MPI_Comm), INTENT(IN) :: comm1, comm2
```

```
INTEGER, INTENT(OUT) :: result
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)
```

```
INTEGER COMM1, COMM2, RESULT, IERROR
```

MPI_IDENT results if and only if comm1 and comm2 are handles for the same object (identical groups and same contexts). MPI_CONGRUENT results if the underlying groups are identical in constituents and rank order; these communicators differ only by context. MPI_SIMILAR results if the group members of both communicators are the same but the rank order differs. MPI_UNEQUAL results otherwise.

7.4.2 Communicator Constructors

The following are collective functions that are invoked by all MPI processes in the group or groups associated with comm, with the exception of MPI_COMM_CREATE_GROUP, MPI_COMM_CREATE_FROM_GROUP, and MPI_INTERCOMM_CREATE_FROM_GROUPS. MPI_COMM_CREATE_GROUP and MPI_COMM_CREATE_FROM_GROUP are invoked only by the MPI processes in the group of the new communicator being constructed. MPI_INTERCOMM_CREATE_FROM_GROUPS is invoked by all the MPI processes in the local and remote groups of the new communicator being constructed. See the discussion below for the definition of local and remote groups.

Rationale. Note that, when using the World Model, there is a chicken-and-egg aspect to MPI in that a communicator is needed to create a new communicator. In the World Model, the base communicator for all MPI communicators is predefined outside of MPI, and is MPI_COMM_WORLD. The World Model was arrived at after considerable debate, and was chosen to increase “safety” of programs written in MPI. (*End of rationale.*)

This chapter presents the following communicator construction routines:

```
MPI_COMM_CREATE, MPI_COMM_DUP, MPI_COMM_IDUP,  
MPI_COMM_DUP_WITH_INFO, MPI_COMM_IDUP_WITH_INFO, MPI_COMM_SPLIT  
and MPI_COMM_SPLIT_TYPE can be used to create both intra-communicators and inter-communicators; MPI_COMM_CREATE_GROUP, MPI_COMM_CREATE_FROM_GROUP and MPI_INTERCOMM_MERGE (see Section 7.6.2) can be used to create intra-communicators;
```

1 MPI_INTERCOMM_CREATE and MPI_INTERCOMM_CREATE_FROM_GROUPS (see Sec-
 2 tion 7.6.2) can be used to create inter-communicators.

3 An intra-communicator involves a single group while an inter-communicator involves
 4 two groups. Where the following discussions address inter-communicator semantics, the
 5 two groups in an inter-communicator are called the *left* and *right* groups. An MPI process
 6 in an inter-communicator is a member of either the left or the right group. From the point
 7 of view of that MPI process, the group that the MPI process is a member of is called the
 8 *local group*; the other group (relative to that MPI process) is the *remote group*. The left
 9 and right group labels give us a way to describe the two groups in an inter-communicator
 10 that is not relative to any particular MPI process (as the local and remote groups are).

11
 12
 13 MPI_COMM_DUP(comm, newcomm)

14	IN	comm	communicator (handle)
15	OUT	newcomm	copy of comm (handle)

17 C binding

18 int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)

20 Fortran 2008 binding

21 MPI_Comm_dup(comm, newcomm, ierror)
 22 TYPE(MPI_Comm), INTENT(IN) :: comm
 23 TYPE(MPI_Comm), INTENT(OUT) :: newcomm
 24 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

26 Fortran binding

27 MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
 28 INTEGER COMM, NEWCOMM, IERROR

29 MPI_COMM_DUP duplicates the existing communicator `comm` with associated key
 30 values and topology information. For each key value, the respective copy callback function
 31 determines the attribute value associated with this key in the new communicator; one
 32 particular action that a copy callback may take is to delete the attribute from the new
 33 communicator. MPI_COMM_DUP returns in `newcomm` a new communicator with the same
 34 group or groups, same topology, and any copied cached information, but a new context (see
 35 Section 7.7.1).

37 *Advice to users.* This operation is used to provide a parallel library with a duplicate
 38 communication space that has the same properties as the original communicator. This
 39 includes any attributes (see below) and topologies (see Chapter 8). This call is valid
 40 even if there are pending point-to-point communications involving the communicator
 41 `comm`. A typical call might involve a MPI_COMM_DUP at the beginning of the
 42 parallel call, and an MPI_COMM_FREE of that duplicated communicator at the end
 43 of the call. Other models of communicator management are also possible.

44 This call applies to both intra- and inter-communicators. (*End of advice to users.*)

46 *Advice to implementors.* One need not actually copy the group information, but only
 47 add a new reference and increment the reference count. Copy on write can be used
 48 for the cached information. (*End of advice to implementors.*)

MPI_COMM_DUP_WITH_INFO(comm, info, newcomm)			1
IN	comm	communicator (handle)	2
IN	info	info object (handle)	3
OUT	newcomm	copy of comm (handle)	4
			5
			6

C binding

```
int MPI_Comm_dup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm)
```

Fortran 2008 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
```

Fortran 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)
```

Fortran 2008 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
```

Fortran binding

```
MPI_COMM_IDUP(COMM, NEWCOMM, REQUEST, IERROR)
```

```
INTEGER COMM, NEWCOMM, REQUEST, IERROR
```

1 MPI_COMM_IDUP is a nonblocking variant of MPI_COMM_DUP. With the exception
 2 of its nonblocking behavior, the semantics of MPI_COMM_IDUP are as if MPI_COMM_DUP
 3 was executed at the time that MPI_COMM_IDUP is called. For example, attributes changed
 4 after MPI_COMM_IDUP will not be copied to the new communicator. All restrictions and
 5 assumptions for nonblocking collective operations (see Section 6.12) apply to
 6 MPI_COMM_IDUP and the returned request.

7 It is erroneous to use the communicator `newcomm` as an input argument to other MPI
 8 functions before the MPI_COMM_IDUP operation completes.

10 MPI_COMM_IDUP_WITH_INFO(comm, info, newcomm, request)

12	IN	comm	communicator (handle)
13	IN	info	info object (handle)
14	OUT	newcomm	copy of comm (handle)
15	OUT	request	communication request (handle)

18 C binding

```
19 int MPI_Comm_idup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm,
20                             MPI_Request *request)
21
```

22 Fortran 2008 binding

```
23 MPI_Comm_idup_with_info(comm, info, newcomm, request, ierror)
24     TYPE(MPI_Comm), INTENT(IN) :: comm
25     TYPE(MPI_Info), INTENT(IN) :: info
26     TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS :: newcomm
27     TYPE(MPI_Request), INTENT(OUT) :: request
28     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
```

29 Fortran binding

```
30 MPI_COMM_IDUP_WITH_INFO(COMM, INFO, NEWCOMM, REQUEST, IERROR)
31     INTEGER COMM, INFO, NEWCOMM, REQUEST, IERROR
32
```

33 MPI_COMM_IDUP_WITH_INFO is a nonblocking variant of
 34 MPI_COMM_DUP_WITH_INFO. With the exception of its nonblocking behavior, the se-
 35 mantics of MPI_COMM_IDUP_WITH_INFO are as if MPI_COMM_DUP_WITH_INFO was
 36 executed at the time that MPI_COMM_IDUP_WITH_INFO is called. For example, attributes
 37 or info hints changed after MPI_COMM_IDUP_WITH_INFO will not be copied to the new
 38 communicator. All restrictions and assumptions for nonblocking collective operations (see
 39 Section 6.12) apply to MPI_COMM_IDUP_WITH_INFO and the returned request.

40 It is erroneous to use the communicator `newcomm` as an input argument to other MPI
 41 functions before the MPI_COMM_IDUP_WITH_INFO operation completes.

42
 43 *Rationale.* The MPI_COMM_IDUP and MPI_COMM_IDUP_WITH_INFO functions
 44 are crucial for the development of purely nonblocking libraries (see [40]). (*End of*
 45 *rationale.*)

MPI_COMM_CREATE(comm, group, newcomm)			1
IN	comm	communicator (handle)	2
			3
IN	group	group, which is a subset of the group of comm (handle)	4
			5
OUT	newcomm	new communicator (handle)	6
			7

C binding

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
```

Fortran 2008 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
```

Fortran binding

```
MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
  INTEGER COMM, GROUP, NEWCOMM, IERROR
```

If `comm` is an intra-communicator, this function returns a new communicator `newcomm` with communication group defined by the `group` argument. No cached information propagates from `comm` to `newcomm` and no virtual topology information is added to the created communicator. Each MPI 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 MPI processes may specify different values for the `group` argument. If an MPI process calls with a nonempty `group` then all MPI processes in that `group` must call the function with the same `group` as argument, that is the same MPI processes in the same order. Otherwise, the call is erroneous. This implies that the set of groups specified across the MPI processes must be disjoint. If the calling MPI 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 an MPI 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 MPI processes in the group of `comm`.

Rationale. The interface supports the original mechanism from MPI-1.1, which required the same `group` in all MPI 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 MPI 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 MPI processes call with the same `group` (the `group` associated with the communicator). When calling MPI_COMM_CREATE, the MPI processes provide the same `group` or disjoint sub-groups. 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 MPI 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 MPI process. (*End of advice to implementors.*)

If `comm` is an inter-communicator, then the output communicator is also an inter-communicator where the local group consists only of those MPI processes contained in `group` (see Figure 7.1). The `group` argument should only contain those MPI processes in the local group of the input inter-communicator that are to be a part of `newcomm`. All MPI 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 MPI process in the local group of the inter-communicator, or if the calling MPI 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 inter-communicator with MPI_GROUP_EMPTY because the side with the empty group must return MPI_COMM_NULL. (*End of rationale.*)

Example 7.1. Inter-communicator creation.

The following example illustrates how the first node in the left side of an inter-communicator could be joined with all members on the right side of an inter-communicator to form a new inter-communicator.

```
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 inter-communicator: "inter_comm" */
...
```

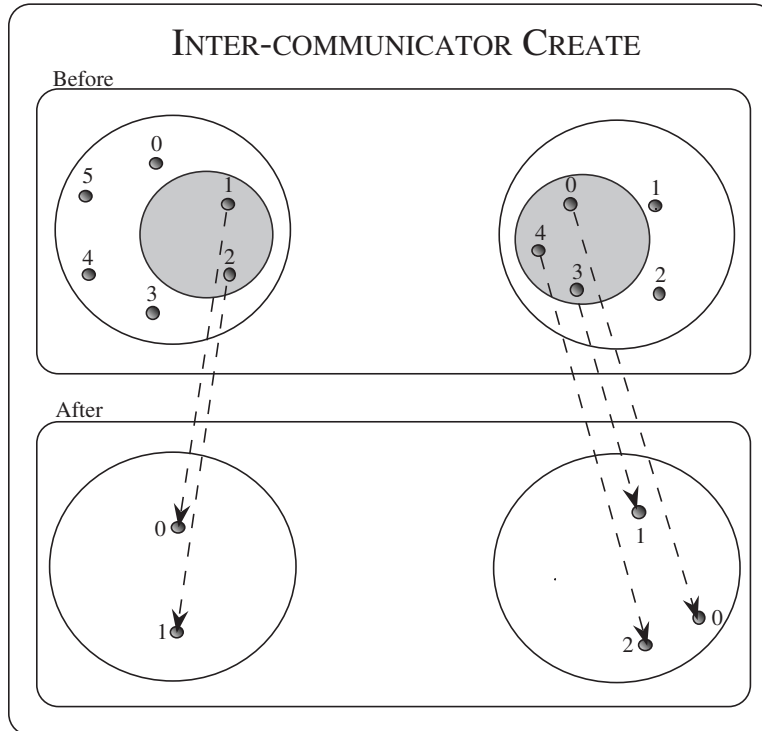



Figure 7.1: Inter-communicator creation using `MPI_COMM_CREATE` extended to inter-communicators. The input groups are those in the grey circle.

```

/* Construct the group of MPI processes to be in new
   inter-communicator */
if (/* I'm on the left side of the inter-communicator */) {
    MPI_Comm_group(inter_comm, &local_group);
    MPI_Group_incl(local_group, 1, &rank, &group);
    MPI_Group_free(&local_group);
}
else
    MPI_Comm_group(inter_comm, &group);

MPI_Comm_create(inter_comm, group, &new_inter_comm);
MPI_Group_free(&group);

```

`MPI_COMM_CREATE_GROUP(comm, group, tag, newcomm)`

IN	comm	intra-communicator (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

```

1 int MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag,
2                           MPI_Comm *newcomm)
3
4

```

Fortran 2008 binding

```

5 MPI_Comm_create_group(comm, group, tag, newcomm, ierror)
6     TYPE(MPI_Comm), INTENT(IN) :: comm
7     TYPE(MPI_Group), INTENT(IN) :: group
8     INTEGER, INTENT(IN) :: tag
9     TYPE(MPI_Comm), INTENT(OUT) :: newcomm
10    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11

```

Fortran binding

```

12 MPI_COMM_CREATE_GROUP(COMM, GROUP, TAG, NEWCOMM, IERROR)
13     INTEGER COMM, GROUP, TAG, NEWCOMM, IERROR
14
15

```

16 MPI_COMM_CREATE_GROUP is similar to MPI_COMM_CREATE; however,
17 MPI_COMM_CREATE must be called by all MPI processes in the group of `comm`, whereas
18 MPI_COMM_CREATE_GROUP must be called by all MPI processes in `group`, which is
19 a subgroup of the group of `comm`. In addition, MPI_COMM_CREATE_GROUP requires
20 that `comm` is an intra-communicator. MPI_COMM_CREATE_GROUP returns a new intra-
21 communicator, `newcomm`, for which the `group` argument defines the communication group.
22 No cached information propagates from `comm` to `newcomm` and no virtual topology infor-
23 mation is added to the created communicator. Each MPI process must provide a `group`
24 argument that is a subgroup of the group associated with `comm`; this could be
25 MPI_GROUP_EMPTY. If a nonempty group is specified, then all MPI processes in that group
26 must call the function, and each of these MPI processes must provide the same arguments,
27 including a group that contains the same members with the same ordering. Otherwise the
28 call is erroneous. If the calling MPI process is a member of the group given as the `group`
29 argument, then `newcomm` is a communicator with `group` as its associated group. If the
30 calling MPI process is not a member of `group`, e.g., `group` is MPI_GROUP_EMPTY, then the
31 call is a local operation and MPI_COMM_NULL is returned as `newcomm`.

32 *Rationale.* Functionality similar to MPI_COMM_CREATE_GROUP can be imple-
33 mented through repeated MPI_INTERCOMM_CREATE and
34 MPI_INTERCOMM_MERGE calls that start with the MPI_COMM_SELF communica-
35 tors at each MPI process in `group` and build up an intra-communicator with group
36 `group` [17]. Such an algorithm requires the creation of many intermediate communica-
37 tors; MPI_COMM_CREATE_GROUP can provide a more efficient implementation that
38 avoids this overhead. (*End of rationale.*)

39 *Advice to users.* An inter-communicator can be created collectively over MPI pro-
40 cesses in the union of the local and remote groups by creating the local communicator
41 using MPI_COMM_CREATE_GROUP and using that communicator as the local com-
42 municator argument to MPI_INTERCOMM_CREATE. (*End of advice to users.*)

43 The `tag` argument does not conflict with tags used in point-to-point communication
44 and is not permitted to be a wildcard. If multiple threads at a given MPI process per-
45 form concurrent MPI_COMM_CREATE_GROUP operations, the user must distinguish these
46 operations by providing different `tag` or `comm` arguments.

Advice to users. MPI_COMM_CREATE may provide lower overhead than MPI_COMM_CREATE_GROUP because it can take advantage of collective communication on comm when constructing newcomm. (*End of advice to users.*)

MPI_COMM_SPLIT(comm, color, key, newcomm)

IN	comm	communicator (handle)
IN	color	control of subset assignment (integer)
IN	key	control of rank assignment (integer)
OUT	newcomm	new communicator (handle)

C binding

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
```

Fortran 2008 binding

```
MPI_Comm_split(comm, color, key, newcomm, ierror)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, INTENT(IN) :: color, key
  TYPE(MPI_Comm), INTENT(OUT) :: newcomm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
  INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR
```

This function partitions the group associated with comm into disjoint subgroups, one for each value of color. Each subgroup contains all MPI processes of the same color. Within each subgroup, the MPI processes are ranked in the order defined by the value of the argument key, with ties broken according to their rank in the old group. A new communicator is created for each subgroup and returned in newcomm. An MPI process may supply the color value MPI_UNDEFINED, in which case newcomm returns MPI_COMM_NULL. This is a collective call, but each MPI process is permitted to provide different values for color and key. No cached information propagates from comm to newcomm and no virtual topology information is added to the created communicators.

With an intra-communicator comm, a call to MPI_COMM_CREATE(comm, group, newcomm) is equivalent to a call to MPI_COMM_SPLIT(comm, color, key, newcomm), where MPI 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 MPI processes that are not members of their group argument provide color = MPI_UNDEFINED. The value of color must be nonnegative or MPI_UNDEFINED.

Advice to users. This is an extremely powerful mechanism for dividing a single communicating group of MPI processes into k subgroups, with k chosen implicitly by the user (by the number of colors asserted over all the MPI processes). Each resulting communicator will be nonoverlapping. Such a division could be useful for defining a hierarchy of computations, such as for multigrid, or linear algebra. For intra-communicators, MPI_COMM_SPLIT provides similar capability as MPI_COMM_CREATE to split a

communicating group into disjoint subgroups. `MPI_COMM_SPLIT` is useful when some MPI processes do not have complete information of the other members in their group, but all MPI 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 MPI 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 MPI processes in a given group have complete information of the members of their group and synchronization with MPI 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 MPI 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 MPI 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 MPI processes in a given color will have the relative rank order as they did in their parent group.

(End of advice to users.)

Rationale. color is restricted to be nonnegative, so as not to conflict with the value assigned to `MPI_UNDEFINED`. *(End of rationale.)*

The result of `MPI_COMM_SPLIT` on an inter-communicator is that those MPI processes on the left with the same color as those MPI processes on the right combine to create a new inter-communicator. The key argument describes the relative rank of MPI processes on each side of the inter-communicator (see Figure 7.2). For those colors that are specified only on one side of the inter-communicator, `MPI_COMM_NULL` is returned. `MPI_COMM_NULL` is also returned to those MPI processes that specify `MPI_UNDEFINED` as the color.

Advice to users. For inter-communicators, `MPI_COMM_SPLIT` is more general than `MPI_COMM_CREATE`. A single call to `MPI_COMM_SPLIT` can create a set of disjoint inter-communicators, while a call to `MPI_COMM_CREATE` creates only one. *(End of advice to users.)*

Example 7.2. Parallel client-server model.

The following client code illustrates how clients on the left side of an inter-communicator could be assigned to a single server from a pool of servers on the right side of an inter-communicator.

```

41 /* Client code */
42 MPI_Comm multiple_server_comm;
43 MPI_Comm single_server_comm;
44 int      color, rank, num_servers;
45
46 /* Create inter-communicator with clients and servers:
47    multiple_server_comm */
48 ...

```

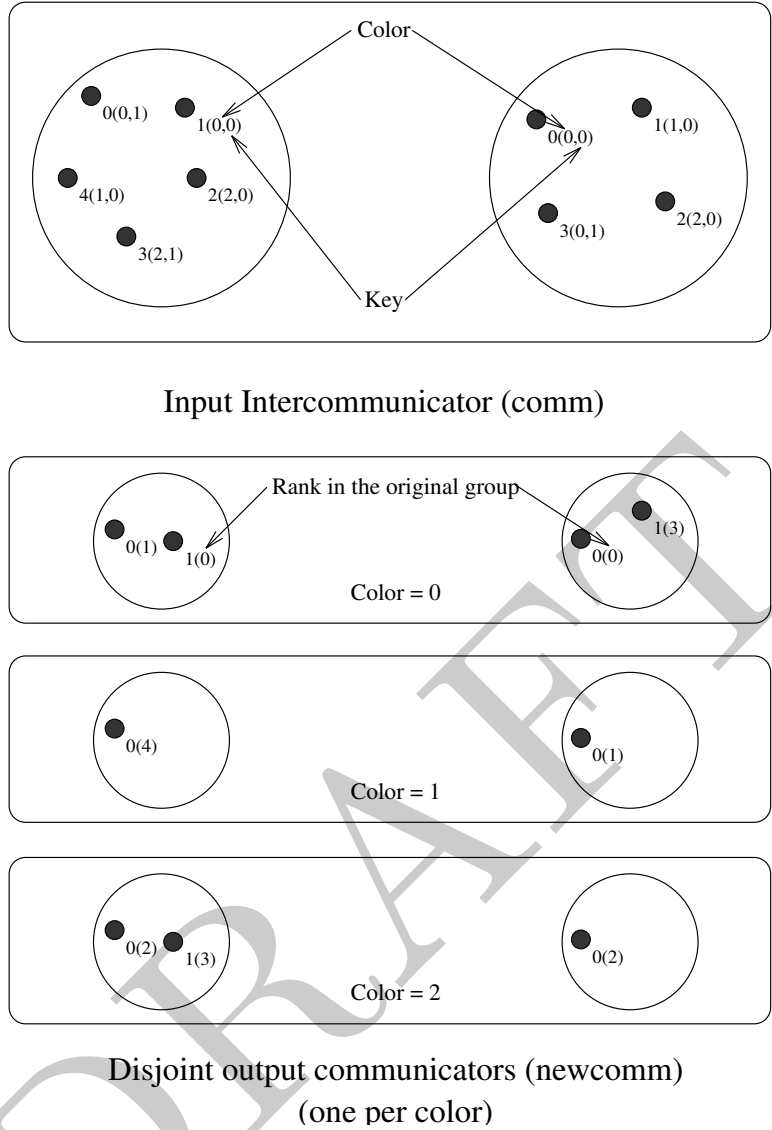


Figure 7.2: Inter-communicator construction achieved by splitting an existing inter-communicator with MPI_COMM_SPLIT extended to inter-communicators.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

```

1
2  /* Find out the number of servers available */
3  MPI_Comm_remote_size(multiple_server_comm, &num_servers);
4
5  /* Determine my color */
6  MPI_Comm_rank(multiple_server_comm, &rank);
7  color = rank % num_servers;
8
9  /* Split the inter-communicator */
10 MPI_Comm_split(multiple_server_comm, color, rank,
11                &single_server_comm);

```

The following is the corresponding server code:

```

13 /* Server code */
14 MPI_Comm multiple_client_comm;
15 MPI_Comm single_server_comm;
16 int rank;
17
18 /* Create inter-communicator with clients and servers:
19 multiple_client_comm */
20 ...
21
22 /* Split the inter-communicator for a single server per group
23 of clients */
24 MPI_Comm_rank(multiple_client_comm, &rank);
25 MPI_Comm_split(multiple_client_comm, rank, 0,
26                &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)
INOUT	info	info argument (handle)
OUT	newcomm	new communicator (handle)

C binding

```

38 int MPI_Comm_split_type(MPI_Comm comm, int split_type, int key, MPI_Info info,
39                        MPI_Comm *newcomm)

```

Fortran 2008 binding

```

42 MPI_Comm_split_type(comm, split_type, key, info, newcomm, ierror)
43   TYPE(MPI_Comm), INTENT(IN) :: comm
44   INTEGER, INTENT(IN) :: split_type, key
45   TYPE(MPI_Info), INTENT(IN) :: info
46   TYPE(MPI_Comm), INTENT(OUT) :: newcomm
47   INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran 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 such that each subgroup contains all MPI processes in the same grouping referred to by `split_type`. Within each subgroup, the MPI processes are ranked in the order defined by the value of the argument `key`, with ties broken according to their rank in the old group. A new communicator is created for each subgroup and returned in `newcomm`. This is a collective call. All MPI processes in the group associated with `comm` must provide the same `split_type`, but each MPI process is permitted to provide different values for `key`. An exception to this rule is that an MPI process may supply the type value `MPI_UNDEFINED`, in which case `MPI_COMM_NULL` is returned in `newcomm` for such MPI process. No cached information propagates from `comm` to `newcomm` and no virtual topology information is added to the created communicators.

For `split_type`, the following values are defined by MPI:

MPI_COMM_TYPE_SHARED: all MPI processes in `newcomm` can create a shared memory segment (e.g., with a successful call to `MPI_WIN_ALLOCATE_SHARED`). This segment can subsequently be used for load/store accesses by all MPI processes in `newcomm`.

Advice to users. Since the location of some of the MPI processes may change during the application execution, the communicators created with the value `MPI_COMM_TYPE_SHARED` before this change may not reflect an actual ability to share memory between MPI processes after this change. (*End of advice to users.*)

MPI_COMM_TYPE_HW_GUIDED: this value specifies that the communicator `comm` is split according to a **hardware resource type** (for example a computing core or an L3 cache) specified by the "mpi_hw_resource_type" info key. Each output communicator `newcomm` corresponds to a single instance of the specified hardware resource type. The MPI processes in the group associated with the output communicator `newcomm` utilize that specific hardware resource type instance, and no other instance of the same hardware resource type.

If an MPI process does not meet the above criteria, then `MPI_COMM_NULL` is returned in `newcomm` for such MPI process.

`MPI_COMM_NULL` is also returned in `newcomm` in the following cases:

- `MPI_INFO_NULL` is provided.
- The info handle does not include the key "mpi_hw_resource_type".
- The MPI implementation neither recognizes nor supports the info key "mpi_hw_resource_type".
- The MPI implementation does not recognize the value associated with the info key "mpi_hw_resource_type".

The MPI implementation will return in the group of the output communicator `newcomm` the largest subset of MPI processes that match the splitting criterion.

The MPI processes in the group associated with `newcomm` are ranked in the order defined by the value of the argument `key` with ties broken according to their rank in the group associated with `comm`.

Advice to users. The set of hardware resources that an MPI process is able to utilize may change during the application execution (e.g., because of the relocation of an MPI process), in which case the communicators created with the value `MPI_COMM_TYPE_HW_GUIDED` before this change may not reflect the utilization of hardware resources of such MPI process at any time after the communicator creation. (*End of advice to users.*)

The user explicitly constrains with the `info` argument the splitting of the input communicator `comm`. To this end, the `info` key `"mpi_hw_resource_type"` is reserved and its associated value is an implementation-defined string designating the type of the requested hardware resource (e.g., `"NUMANode"`, `"Package"` or `"L3Cache"`).

The value `"mpi_shared_memory"` is reserved and its use is equivalent to using `MPI_COMM_TYPE_SHARED` for the `split_type` parameter.

Rationale. The value `"mpi_shared_memory"` is defined in order to ensure consistency between the use of `MPI_COMM_TYPE_SHARED` and the use of `MPI_COMM_TYPE_HW_GUIDED`. (*End of rationale.*)

All MPI processes must provide the same value for the `info` key `"mpi_hw_resource_type"`.

Example 7.3. Splitting `MPI_COMM_WORLD` into `NUMANode` subcommunicators.

```
MPI_Info info;
MPI_Comm hwcomm;
int rank;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Info_create(&info);
MPI_Info_set(info, "mpi_hw_resource_type", "NUMANode");
MPI_Comm_split_type(MPI_COMM_WORLD,
                   MPI_COMM_TYPE_HW_GUIDED,
                   rank, info, &hwcomm);
```

`MPI_COMM_TYPE_HW_UNGUIDED`: the group of MPI processes associated with `newcomm` must be a *strict* subset of the group associated with `comm` and each `newcomm` corresponds to a single instance of a **hardware resource type** (for example a computing core or an L3 cache).

All MPI processes in the group associated with `comm` that utilize that specific hardware resource type instance—and no other instance of the same hardware resource type—are included in the group of `newcomm`.

If a given MPI process cannot be a member of a communicator that forms such a strict subset, or does not meet the above criteria, then `MPI_COMM_NULL` is returned in `newcomm` for this process.

Advice to implementors. In a high-quality MPI implementation, the number of different new valid communicators `newcomm` produced by this splitting operation should be minimal unless the user provides a key/value pair that modifies this behavior. The sets of hardware resource types used for the splitting operation are implementation-dependent, but should reflect the hardware of the actual system on which the application is currently executing. (*End of advice to implementors.*)

Rationale. If the hardware resources are hierarchically organized, calling this routine several times using as its input communicator `comm` the output communicator `newcomm` of the previous call creates a sequence of `newcomm` communicators in each MPI process, which exposes a hierarchical view of the hardware platform, as shown in Example 7.4. This sequence of returned `newcomm` communicators may differ from the sets of hardware resource types, as shown in the second splitting operation in Figure 7.3. (*End of rationale.*)

Advice to users. Each output communicator `newcomm` can represent a different hardware resource type (see Figure 7.3 for an example). The set of hardware resources an MPI process utilizes may change during the application execution (e.g., because of MPI process relocation), in which case the communicators created with the value `MPI_COMM_TYPE_HW_UNGUIDED` before this change may not reflect the utilization of hardware resources for such MPI process at any time after the communicator creation. (*End of advice to users.*)

If a valid info handle is provided as an argument, the MPI implementation sets the info key "mpi_hw_resource_type" for each MPI process in the group associated with a returned `newcomm` communicator and the info key value is an implementation-defined string that indicates the hardware resource type represented by `newcomm`. The same hardware resource type must be set in all MPI processes in the group associated with `newcomm`.

Example 7.4. Recursive splitting of `MPI_COMM_WORLD`.

```
#define MAX_NUM_LEVELS 32

MPI_Comm hwcomm[MAX_NUM_LEVELS];
int      rank, level_num = 0;

hwcomm[level_num] = MPI_COMM_WORLD;

while((hwcomm[level_num] != MPI_COMM_NULL)
      && (level_num < MAX_NUM_LEVELS-1)){
    MPI_Comm_rank(hwcomm[level_num], &rank);
    MPI_Comm_split_type(hwcomm[level_num],
                        MPI_COMM_TYPE_HW_UNGUIDED,
                        rank,
                        MPI_INFO_NULL,
                        &hwcomm[level_num+1]);
    level_num++;
}
```

Advice to implementors. Implementations can define their own `split_type` values, or use the `info` argument, to assist in creating communicators that help expose platform-specific information to the application. The concept of hardware-based communicators was first described by Träff [67] for SMP systems. Guided and unguided modes description as well as an implementation path are introduced by Goglin *et al.* [27]. (*End of advice to implementors.*)

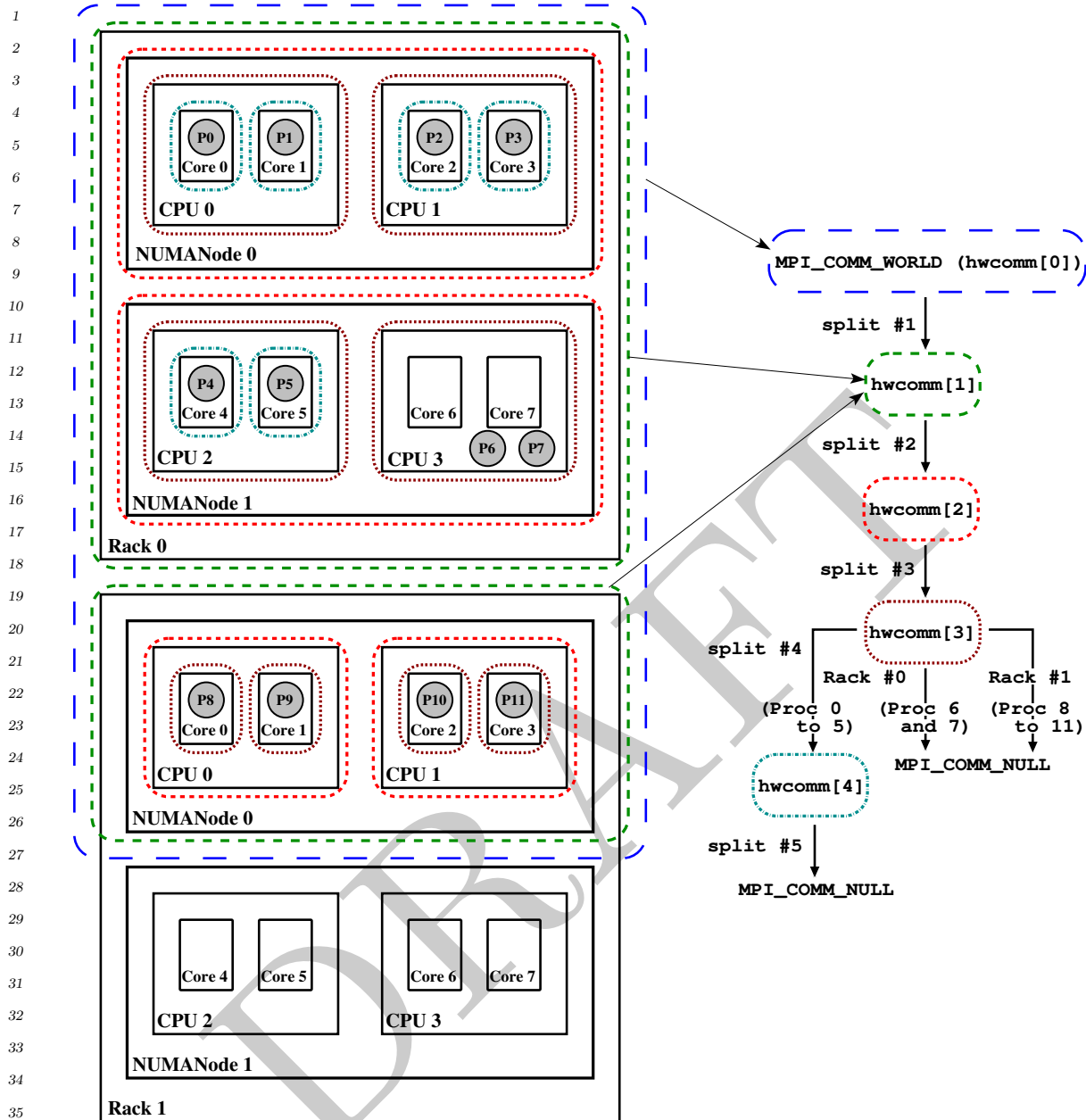


Figure 7.3: Recursive splitting of `MPI_COMM_WORLD` with `MPI_COMM_SPLIT_TYPE` and `MPI_COMM_TYPE_HW_UNGUIDED`. Dashed lines represent communicators whilst solid lines represent hardware resources. MPI processes (P0 to P11) utilize exclusively their respective core, except for P6 and P7, which utilize CPU #3 of Rack #0 and can therefore use Cores #6 and #7 indifferently. The second splitting operation yields two subcommunicators corresponding to NUMANodes in Rack #0 and to CPUs in Rack #1 because Rack #1 features only one NUMANode, which corresponds to the whole portion of the Rack that is included in `MPI_COMM_WORLD` and `hwcomm[1]`. For the first splitting operation, the hardware resource type returned in the info argument is “Rack” on the MPI processes on Rack #0, whereas on Rack #1, it can be either “Rack” or “NUMANode”.

<code>MPI_COMM_CREATE_FROM_GROUP(group, stringtag, info, errhandler, newcomm)</code>			1
IN	<code>group</code>	group (handle)	2
IN	<code>stringtag</code>	unique identifier for this operation (string)	3
IN	<code>info</code>	info object (handle)	4
IN	<code>errhandler</code>	error handler to be attached to new intra-communicator (handle)	5
OUT	<code>newcomm</code>	new communicator (handle)	6

C binding

```
int MPI_Comm_create_from_group(MPI_Group group, const char *stringtag,
                              MPI_Info info, MPI_Errhandler errhandler, MPI_Comm *newcomm)
```

Fortran 2008 binding

```
MPI_Comm_create_from_group(group, stringtag, info, errhandler, newcomm, ierror)
  TYPE(MPI_Group), INTENT(IN) :: group
  CHARACTER(LEN=*), INTENT(IN) :: stringtag
  TYPE(MPI_Info), INTENT(IN) :: info
  TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
  TYPE(MPI_Comm), INTENT(OUT) :: newcomm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_COMM_CREATE_FROM_GROUP(GROUP, STRINGTAG, INFO, ERRHANDLER, NEWCOMM, IERROR)
  INTEGER GROUP, INFO, ERRHANDLER, NEWCOMM, IERROR
  CHARACTER*(*) STRINGTAG
```

`MPI_COMM_CREATE_FROM_GROUP` is similar to `MPI_COMM_CREATE_GROUP`, except that the set of MPI processes involved in the creation of the new intra-communicator is specified by a `group` argument, rather than the group associated with a pre-existing communicator. If a nonempty group is specified, then all MPI processes in that group must call the function and each of these MPI processes must provide the same arguments, including a group that contains the same members with the same ordering, and identical `stringtag` value. In the event that `MPI_GROUP_EMPTY` is supplied as the `group` argument, then the call is a local operation and `MPI_COMM_NULL` is returned as `newcomm`. The `stringtag` argument is analogous to the `tag` used for `MPI_COMM_CREATE_GROUP`. If multiple threads at a given MPI process perform concurrent `MPI_COMM_CREATE_FROM_GROUP` operations, the user must distinguish these operations by providing different `stringtag` arguments. The `stringtag` shall not exceed `MPI_MAX_STRINGTAG_LEN` characters in length. For C, this includes space for a null terminating character. `MPI_MAX_STRINGTAG_LEN` shall have a value of at least 63.

The `errhandler` argument specifies an error handler to be attached to the new intra-communicator. Section 9.3 specifies the error handler to be invoked if an error is encountered during the invocation of `MPI_COMM_CREATE_FROM_GROUP`.

The `info` argument provides hints and assertions, possibly MPI implementation dependent, which indicate desired characteristics and guide communicator creation.

Advice to users. The `stringtag` argument is used to distinguish concurrent communicator construction operations issued by different entities. As such, it is important

1 to ensure that this argument is unique for each concurrent call to
 2 MPI_COMM_CREATE_FROM_GROUP. Reverse domain name notation convention [1]
 3 is one approach to constructing unique stringtag arguments. See also example 11.10.
 4 (*End of advice to users.*)

6 7.4.3 Communicator Destructors

9 MPI_COMM_FREE(comm)

11 INOUT comm communicator to be destroyed (handle)

13 C binding

14 int MPI_Comm_free(MPI_Comm *comm)

16 Fortran 2008 binding

17 MPI_Comm_free(comm, ierror)

18 TYPE(MPI_Comm), INTENT(INOUT) :: comm

19 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

20 Fortran binding

21 MPI_COMM_FREE(COMM, IERROR)

22 INTEGER COMM, IERROR

23
 24 This collective operation marks the communication object for deallocation. The handle
 25 is set to MPI_COMM_NULL. Any pending operations that use this communicator will com-
 26 plete normally; the object is actually deallocated only if there are no other active references
 27 to it. This call applies to intra- and inter-communicators. The delete callback functions for
 28 all cached attributes (see Section 7.7) are called in arbitrary order.

29 *Advice to implementors.* Though collective, it is anticipated that this operation will
 30 normally be implemented to be local, though a debugging version of an MPI library
 31 might choose to synchronize. (*End of advice to implementors.*)

34 7.4.4 Communicator Info

35 Hints specified via info (see Chapter 10) allow a user to provide information to direct
 36 optimization. Providing hints may enable an implementation to deliver increased per-
 37 formance or minimize use of system resources. An implementation is free to ignore all
 38 hints; however, applications must comply with any info hints they provide that are used
 39 by the MPI implementation (i.e., are returned by a call to MPI_COMM_GET_INFO) and
 40 that place a restriction on the behavior of the application. Hints are specified on a per
 41 communicator basis, in MPI_COMM_DUP_WITH_INFO, MPI_COMM_IDUP_WITH_INFO,
 42 MPI_COMM_SET_INFO, MPI_COMM_SPLIT_TYPE, MPI_DIST_GRAPH_CREATE, and
 43 MPI_DIST_GRAPH_CREATE_ADJACENT, via the opaque info object. When an info object
 44 that specifies a subset of valid hints is passed to MPI_COMM_SET_INFO, there will be no
 45 effect on previously set or defaulted hints that the info does not specify.

47 *Advice to implementors.* It may happen that a program is coded with hints for one
 48 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 MPI 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 MPI 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 MPI 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 nonovertaking 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 MPI 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)
```

Fortran 2008 binding

```
MPI_Comm_set_info(comm, info, ierror)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Info), INTENT(IN) :: info
```

1 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

2 **Fortran binding**

3 MPI_COMM_SET_INFO(COMM, INFO, IERROR)

4 INTEGER COMM, INFO, IERROR

5
6 MPI_COMM_SET_INFO updates the hints of the communicator associated with `comm`
7 using the hints provided in `info`. This operation has no effect on previously set or defaulted
8 hints that are not specified by `info`. It also has no effect on previously set or defaulted
9 hints that are specified by `info`, but are ignored by the MPI implementation in this call to
10 MPI_COMM_SET_INFO. MPI_COMM_SET_INFO is a collective routine. The `info` object
11 may be different on each MPI process, but any `info` entries that an implementation requires
12 to be the same on all MPI processes must appear with the same value in each MPI process's
13 `info` object.

14
15 *Advice to users.* Some `info` items that an implementation can use when it creates
16 a communicator cannot easily be changed once the communicator has been created.
17 Thus, an implementation may ignore hints issued in this call that it would have
18 accepted in a creation call. An implementation may also be unable to update certain
19 `info` hints in a call to MPI_COMM_SET_INFO. MPI_COMM_GET_INFO can be used to
20 determine whether updates to existing `info` hints were ignored by the implementation.
21 (*End of advice to users.*)

22 *Advice to users.* Setting `info` hints on the predefined communicators
23 MPI_COMM_WORLD and MPI_COMM_SELF may have unintended effects, as changes to
24 these global objects may affect all components of the application, including libraries
25 and tools. Users must ensure that all components of the application that use a given
26 communicator, including libraries and tools, can comply with any `info` hints associated
27 with that communicator. (*End of advice to users.*)

28
29
30 MPI_COMM_GET_INFO(comm, info_used)

31 IN `comm` communicator object (handle)

32 OUT `info_used` new `info` object (handle)

33 **C binding**

34 int MPI_Comm_get_info(MPI_Comm comm, MPI_Info *info_used)

35 **Fortran 2008 binding**

36 MPI_Comm_get_info(comm, info_used, ierror)

37 TYPE(MPI_Comm), INTENT(IN) :: comm

38 TYPE(MPI_Info), INTENT(OUT) :: info_used

39 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

40 **Fortran binding**

41 MPI_COMM_GET_INFO(COMM, INFO_USED, IERROR)

42 INTEGER COMM, INFO_USED, IERROR

43
44 MPI_COMM_GET_INFO returns a new `info` object containing the hints of the commu-
45 nicator associated with `comm`. The current setting of all hints related to this communicator
46
47
48

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`.

7.5 Motivating Examples

7.5.1 Current Practice #1

Example 7.5. Parallel output of a message

```
int main(int argc, char *argv[])
{
    int me, size;
    ...
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    (void)printf("MPI process %d size %d\n", me, size);
    ...
    MPI_Finalize();
    return 0;
}
```

Example 7.5 is a do-nothing program that initializes itself, and refers to the “all” communicator, and prints a message. It terminates itself too. This example does not imply that MPI supports `printf`-like communication itself.

Example 7.6. Message exchange (supposing that `size` is even)

```
int main(int argc, char *argv[])
{
    int me, size;
    int SOME_TAG = 0;
    ...
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &me); /* local */
    MPI_Comm_size(MPI_COMM_WORLD, &size); /* local */

    if((me % 2) == 0)
    {
        /* send unless highest-numbered MPI 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);
}
```

```

1
2
3     ...
4     MPI_Finalize();
5     return 0;
6 }

```

Example 7.6 schematically illustrates message exchanges between “even” and “odd” MPI processes in the “all” communicator.

7.5.2 Current Practice #2

Example 7.7.

```

13 int main(int argc, char *argv[])
14 {
15     int me, count;
16     void *data;
17     ...
18
19     MPI_Init(&argc, &argv);
20     MPI_Comm_rank(MPI_COMM_WORLD, &me);
21
22     if(me == 0)
23     {
24         /* get input, create buffer “data” */
25         ...
26     }
27
28     MPI_Bcast(data, count, MPI_BYTE, 0, MPI_COMM_WORLD);
29
30     ...
31     MPI_Finalize();
32     return 0;
33 }

```

Example 7.7 illustrates the use of a collective communication.

7.5.3 (Approximate) Current Practice #3

Example 7.8.

```

38 int main(int argc, char *argv[])
39 {
40     int me, count, count2;
41     void *send_buf, *recv_buf, *send_buf2, *recv_buf2;
42     MPI_Group group_world, grpem;
43     MPI_Comm commWorker;
44     static int ranks[] = {0};
45     ...
46     MPI_Init(&argc, &argv);
47     MPI_Comm_group(MPI_COMM_WORLD, &group_world);
48     MPI_Comm_rank(MPI_COMM_WORLD, &me); /* local */

```



```

1 MPI_Group_excl(group_world, 1, ranks, &grpem); /* local */
2 MPI_Comm_create(MPI_COMM_WORLD, grpem, &commWorker);
3
4 if(me != 0)
5 {
6     /* compute on worker */
7     ...
8     MPI_Reduce(send_buf, recv_buf, count, MPI_INT, MPI_SUM, 1, commWorker);
9     ...
10    MPI_Comm_free(&commWorker);
11 }
12 /* zero falls through immediately to this reduce, others do later... */
13 MPI_Reduce(send_buf2, recv_buf2, count2,
14            MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
15
16 MPI_Group_free(&group_world);
17 MPI_Group_free(&grpem);
18 MPI_Finalize();
19 return 0;
20 }

```

Example 7.8 illustrates how a group consisting of all but the zeroth MPI process of the “all” group is created, and then how a communicator is formed (`commWorker`) for that new group. The new communicator is used in a collective call, and all MPI 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 `commWorker`, and vice versa.

In summary, “group safety” is achieved via communicators because distinct contexts within communicators are enforced to be unique on any MPI process.

7.5.4 Communication Safety Example

The following example (7.9) is meant to illustrate “safety” between point-to-point and collective communication. MPI guarantees that a single communicator can do safe point-to-point and collective communication.

Example 7.9.

```

35 #define TAG_ARBITRARY 12345
36 #define SOME_COUNT 50
37
38 int main(int argc, char *argv[])
39 {
40     int me;
41     MPI_Request request[2];
42     MPI_Status status[2];
43     MPI_Group group_world, subgroup;
44     int ranks[] = {2, 4, 6, 8};
45     MPI_Comm the_comm;
46     ...
47     MPI_Init(&argc, &argv);
48     MPI_Comm_group(MPI_COMM_WORLD, &group_world);

```

```

1  MPI_Group_incl(group_world, 4, ranks, &subgroup); /* local */
2  MPI_Group_rank(subgroup, &me); /* local */
3
4  MPI_Comm_create(MPI_COMM_WORLD, subgroup, &the_comm);
5
6  if(me != MPI_UNDEFINED)
7  {
8      MPI_Irecv(buff1, count, MPI_DOUBLE, MPI_ANY_SOURCE,
9              TAG_ARBITRARY, the_comm, request);
10     MPI_Isend(buff2, count, MPI_DOUBLE, (me+1)%4, TAG_ARBITRARY,
11             the_comm, request+1);
12     for(i = 0; i < SOME_COUNT; i++)
13         MPI_Reduce(..., the_comm);
14     MPI_Waitall(2, request, status);
15
16     MPI_Comm_free(&the_comm);
17 }
18
19 MPI_Group_free(&group_world);
20 MPI_Group_free(&subgroup);
21 MPI_Finalize();
22 return 0;
23 }

```

7.5.5 Library Example #1

Example 7.10. First library example
The main program:

```

28 int main(int argc, char *argv[])
29 {
30     int done = 0;
31     user_lib_t *libh_a, *libh_b;
32     void *dataset1, *dataset2;
33     ...
34     MPI_Init(&argc, &argv);
35     ...
36     init_user_lib(MPI_COMM_WORLD, &libh_a);
37     init_user_lib(MPI_COMM_WORLD, &libh_b);
38     ...
39     user_start_op(libh_a, dataset1);
40     user_start_op(libh_b, dataset2);
41     ...
42     while(!done)
43     {
44         /* work */
45         ...
46         MPI_Reduce(..., MPI_COMM_WORLD);
47         ...
48         /* see if done */
49         ...

```

```

}
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);
}

```

7.5.6 Library Example #2

Example 7.11. Second library example

The main program:

```

1  int main(int argc, char *argv[])
2  {
3      int ma, mb;
4      MPI_Group group_world, group_a, group_b;
5      MPI_Comm comm_a, comm_b;
6
7      static int list_a[] = {0, 1};
8      #if defined(EXAMPLE_2B) || defined(EXAMPLE_2C)
9          static int list_b[] = {0, 2, 3};
10     #else /* EXAMPLE_2A */
11         static int list_b[] = {0, 2};
12     #endif
13     int size_list_a = sizeof(list_a)/sizeof(int);
14     int size_list_b = sizeof(list_b)/sizeof(int);
15
16     ...
17     MPI_Init(&argc, &argv);
18     MPI_Comm_group(MPI_COMM_WORLD, &group_world);
19
20     MPI_Group_incl(group_world, size_list_a, list_a, &group_a);
21     MPI_Group_incl(group_world, size_list_b, list_b, &group_b);
22
23     MPI_Comm_create(MPI_COMM_WORLD, group_a, &comm_a);
24     MPI_Comm_create(MPI_COMM_WORLD, group_b, &comm_b);
25
26     if(comm_a != MPI_COMM_NULL)
27         MPI_Comm_rank(comm_a, &ma);
28     if(comm_b != MPI_COMM_NULL)
29         MPI_Comm_rank(comm_b, &mb);
30
31     if(comm_a != MPI_COMM_NULL)
32         lib_call(comm_a);
33
34     if(comm_b != MPI_COMM_NULL)
35     {
36         lib_call(comm_b);
37         lib_call(comm_b);
38     }
39
40     if(comm_a != MPI_COMM_NULL)
41         MPI_Comm_free(&comm_a);
42     if(comm_b != MPI_COMM_NULL)
43         MPI_Comm_free(&comm_b);
44     MPI_Group_free(&group_a);
45     MPI_Group_free(&group_b);
46     MPI_Group_free(&group_world);
47     MPI_Finalize();
48     return 0;
49 }

```

The library:

```

46 void lib_call(MPI_Comm comm)
47 {
48

```

```

1  int me, done = 0;
2  MPI_Status status;
3  MPI_Comm_rank(comm, &me);
4  if(me == 0)
5      while(!done)
6      {
7          MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
8          ...
9      }
10 else
11 {
12     /* work */
13     MPI_Send(..., 0, ARBITRARY_TAG, comm);
14     ...
15 }
16 #ifdef EXAMPLE_2C
17     /* include (resp, exclude) for safety (resp, no safety): */
18     MPI_Barrier(comm);
19 #endif
20 }

```

The above example is really three examples, depending on whether or not one includes rank 3 in `list_b`, and whether or not a synchronization operation 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 a call to `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 synchronization operation 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 [64]). Here we rely on two guarantees of MPI: pairwise ordering of messages between MPI 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 nondeterministic broadcasts or other calls that include wildcard operations will not generally have the good properties of the deterministic implementations of “reduce,” “allreduce,” and “broadcast.” Such algorithms would have to utilize the monotonically increasing tags (within a communicator scope) to keep things straight.

All of the foregoing is a supposition of “collective calls” implemented with point-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 7.9.

7.6 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 MPI processes that are members of the same group. This type of communication is called “**intra-**

1 **communication**” and the communicator used is called an “**intra-communicator**,” as we
2 have noted earlier in the chapter.

3 In modular and multi-disciplinary applications, different MPI process groups execute
4 distinct modules and MPI processes within different modules communicate with one another
5 in a pipeline or a more general module graph. In these applications, the most natural way for
6 a MPI process to specify a target MPI process is by the rank of the target MPI process within
7 the target group. In applications that contain internal user-level servers, each server may be
8 a MPI process group that provides services to one or more clients, and each client may be a
9 MPI process group that uses the services of one or more servers. It is again most natural to
10 specify the target MPI process by rank within the target group in these applications. This
11 type of communication is called “**inter-communication**” and the communicator used is
12 called an “**inter-communicator**,” as introduced earlier.

13 An **inter-communication** is a point-to-point communication between MPI processes
14 in different groups. The group containing an MPI process that initiates an inter-communi-
15 cation operation is called the “local group,” that is, the sender in a send and the receiver in
16 a receive. The group containing the target MPI process is called the “remote group,” that
17 is, the receiver in a send and the sender in a receive. As in intra-communication, the target
18 MPI process is specified using a (communicator, rank) pair. Unlike intra-communication, the
19 rank is relative to a second, remote group.

20 All inter-communicator constructors are blocking except for `MPI_COMM_IDUP` and
21 require that the local and remote groups be disjoint.

22
23 *Advice to users.* The groups must be disjoint for several reasons. Primarily, this
24 is the intent of the inter-communicators—to provide a communicator for communi-
25 cation between disjoint groups. This is reflected in the definition of
26 `MPI_INTERCOMM_MERGE`, which allows the user to control the ranking of the MPI
27 processes in the created intra-communicator; this ranking makes little sense if the
28 groups are not disjoint. In addition, the natural extension of collective operations
29 to inter-communicators makes the most sense when the groups are disjoint. (*End of*
30 *advice to users.*)

31 Here is a summary of the properties of inter-communication and inter-communicators:

- 32
33 • The syntax of point-to-point and collective communication is the same for both inter-
34 and intra-communication. The same communicator can be used both for send and for
35 receive operations.
- 36
37 • A target MPI process is addressed by its rank in the remote group, both for sends and
38 for receives.
- 39
40 • Communications using an inter-communicator are guaranteed not to conflict with any
41 communications that use a different communicator.
- 42
43 • A communicator will provide either intra- or inter-communication, never both.

44 The routine `MPI_COMM_TEST_INTER` may be used to determine if a communicator is an
45 inter- or intra-communicator. Inter-communicators can be used as arguments to some of the
46 other communicator access routines. Inter-communicators cannot be used as input to some
47 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 MPI process in the local group. For intra-communicators, *group* is the communicator group (remote=local), *source* is the rank of the MPI 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 MPI processes in both the local and remote groups. Imagine an MPI process **P** in group \mathcal{P} , which has an inter-communicator $C_{\mathcal{P}}$, and an MPI process **Q** in group \mathcal{Q} , which has an inter-communicator $C_{\mathcal{Q}}$. Then

- $C_{\mathcal{P}}$.**group** describes the group \mathcal{Q} and $C_{\mathcal{Q}}$.**group** describes the group \mathcal{P} .
- $C_{\mathcal{P}}$.**send_context** = $C_{\mathcal{Q}}$.**receive_context** and the context is unique in \mathcal{Q} ;
 $C_{\mathcal{P}}$.**receive_context** = $C_{\mathcal{Q}}$.**send_context** and this context is unique in \mathcal{P} .
- $C_{\mathcal{P}}$.**source** is rank of **P** in \mathcal{P} and $C_{\mathcal{Q}}$.**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.*)

7.6.1 Inter-Communicator Accessors

MPI_COMM_TEST_INTER(comm, flag)

IN	comm	communicator (handle)
OUT	flag	true if comm is an inter-communicator (logical)

C binding

int MPI_Comm_test_inter(MPI_Comm comm, int *flag)

Fortran 2008 binding

MPI_Comm_test_inter(comm, flag, ierror)

Table 7.1: MPI_COMM_* function behavior (in inter-communication mode)

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

```

TYPE(MPI_Comm), INTENT(IN) :: comm
LOGICAL, INTENT(OUT) :: flag
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)
    INTEGER COMM, IERROR
    LOGICAL FLAG

```

This local routine allows the calling MPI 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. 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 MPI processes in the remote group of comm (integer)

C binding

```

int MPI_Comm_remote_size(MPI_Comm comm, int *size)

```

Fortran 2008 binding

```

MPI_Comm_remote_size(comm, size, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR

```


MPI_COMM_REMOTE_GROUP(comm, group)			1
IN	comm	inter-communicator (handle)	2
			3
OUT	group	remote group corresponding to comm (handle)	4
			5

C binding

```
int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
```

Fortran 2008 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
```

Fortran 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.*)

7.6.2 Inter-Communicator Operations

This section introduces five blocking inter-communicator operations.

MPI_INTERCOMM_CREATE is used to bind two intra-communicators into an inter-communicator; the function MPI_INTERCOMM_CREATE_FROM_GROUPS constructs an inter-communicator from two previously defined disjoint groups; 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.

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 an MPI process in the local group and an MPI process in the remote group.

When using the World Model (Section 11.2), the MPI_COMM_WORLD communicator (or preferably a dedicated duplicate thereof) can be this peer communicator. For applications that use the Sessions Model, or the spawn or join operations, it may be necessary to first create an intra-communicator to be used as the peer communicator.

The application topology functions described in Chapter 8 do not apply to inter-communicators. Users that require this capability should utilize

1 MPI_INTERCOMM_MERGE to build an intra-communicator, then apply the graph or car-
 2 tesian topology capabilities to that intra-communicator, creating an appropriate topology-
 3 oriented intra-communicator. Alternatively, it may be reasonable to devise one’s own ap-
 4 plication topology mechanisms for this case, without loss of generality.

5
 6
 7 MPI_INTERCOMM_CREATE(local_comm, local_leader, peer_comm, remote_leader, tag,
 8 newintercomm)

9 IN local_comm local intra-communicator (handle)
 10 IN local_leader rank of local group leader in local_comm (integer)
 11 IN peer_comm “peer” communicator; significant only at the
 12 local_leader (handle)
 13
 14 IN remote_leader rank of remote group leader in peer_comm;
 15 significant only at the local_leader (integer)
 16 IN tag tag (integer)
 17
 18 OUT newintercomm new inter-communicator (handle)

19 C binding

20
 21 int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,
 22 MPI_Comm peer_comm, int remote_leader, int tag,
 23 MPI_Comm *newintercomm)

24 Fortran 2008 binding

25 MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader, tag,
 26 newintercomm, ierror)
 27 TYPE(MPI_Comm), INTENT(IN) :: local_comm, peer_comm
 28 INTEGER, INTENT(IN) :: local_leader, remote_leader, tag
 29 TYPE(MPI_Comm), INTENT(OUT) :: newintercomm
 30 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

32 Fortran binding

33 MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
 34 NEWINTERCOMM, IERROR)
 35 INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
 36 NEWINTERCOMM, IERROR

37 This call creates an inter-communicator. It is collective over the union of the local and
 38 remote groups. MPI processes should provide identical local_comm and
 39 local_leader arguments within each group. Wildcards are not permitted for remote_leader,
 40 local_leader, and tag.

41
 42
 43 MPI_INTERCOMM_CREATE_FROM_GROUPS(local_group, local_leader, remote_group,
 44 remote_leader, stringtag, info, errhandler, newintercomm)

45 IN local_group local group (handle)
 46 IN local_leader rank of local group leader in local_group (integer)
 47
 48 IN remote_group remote group, significant only at local_leader (handle)

IN	remote_leader	rank of remote group leader in remote_group, significant only at local_leader (integer)	1 2
IN	stringtag	unique identifier for this operation (string)	3 4
IN	info	info object (handle)	5
IN	errhandler	error handler to be attached to new inter-communicator (handle)	6 7
OUT	newintercomm	new inter-communicator (handle)	8 9

C binding

```
int MPI_Intercomm_create_from_groups(MPI_Group local_group, int local_leader,
    MPI_Group remote_group, int remote_leader, const char *stringtag,
    MPI_Info info, MPI_Errhandler errhandler, MPI_Comm *newintercomm)
```

Fortran 2008 binding

```
MPI_Intercomm_create_from_groups(local_group, local_leader, remote_group,
    remote_leader, stringtag, info, errhandler, newintercomm, ierror)
TYPE(MPI_Group), INTENT(IN) :: local_group, remote_group
INTEGER, INTENT(IN) :: local_leader, remote_leader
CHARACTER(LEN=*), INTENT(IN) :: stringtag
TYPE(MPI_Info), INTENT(IN) :: info
TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
TYPE(MPI_Comm), INTENT(OUT) :: newintercomm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_INTERCOMM_CREATE_FROM_GROUPS(LOCAL_GROUP, LOCAL_LEADER, REMOTE_GROUP,
    REMOTE_LEADER, STRINGTAG, INFO, ERRHANDLER, NEWINTERCOMM, IERROR)
INTEGER LOCAL_GROUP, LOCAL_LEADER, REMOTE_GROUP, REMOTE_LEADER, INFO,
    ERRHANDLER, NEWINTERCOMM, IERROR
CHARACTER*(*) STRINGTAG
```

This call creates an inter-communicator. Unlike MPI_INTERCOMM_CREATE, this function uses as input previously defined, disjoint local and remote groups. The calling MPI process must be a member of the local group. The call is collective over the union of the local and remote groups. All involved MPI processes shall provide an identical value for the stringtag argument. Within each group, all MPI processes shall provide identical local_group, local_leader arguments. Wildcards are not permitted for the remote_leader or local_leader arguments. The stringtag argument serves the same purpose as the stringtag used in the MPI_COMM_CREATE_FROM_GROUP function; it differentiates concurrent calls in a multithreaded environment. The stringtag shall not exceed MPI_MAX_STRINGTAG_LEN characters in length. For C, this includes space for a null terminating character. MPI_MAX_STRINGTAG_LEN shall have a value of at least 63. In the event that MPI_GROUP_EMPTY is supplied as the local_group or remote_group or both, then the call is a local operation and MPI_COMM_NULL is returned as the newintercomm.

The errhandler argument specifies an error handler to be attached to the new inter-communicator. Section 9.3 specifies the error handler to be invoked if an error is encountered during the invocation of MPI_INTERCOMM_CREATE_FROM_GROUPS.

The info argument provides hints and assertions, possibly MPI implementation dependent, which indicate desired characteristics and guide communicator creation.

`MPI_INTERCOMM_MERGE(intercomm, high, newintracomm)`

IN	intercomm	inter-communicator (handle)
IN	high	ordering of the local and remote groups in the new intra-communicator (logical)
OUT	newintracomm	new intra-communicator (handle)

C binding

```
int MPI_Intercomm_merge(MPI_Comm intercomm, int high, MPI_Comm *newintracomm)
```

Fortran 2008 binding

```
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
```

Fortran binding

```
MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR)
  INTEGER INTERCOMM, NEWINTRACOMM, IERROR
  LOGICAL HIGH
```

This function creates an intra-communicator from the union of the two groups that are associated with `intercomm`. All MPI processes should provide the same `high` value within each of the two groups. If MPI processes in one group provided the value `high = false` and MPI processes in the other group provided the value `high = true` then the union orders the “low” group before the “high” group. If all MPI 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 inter-communicator in each MPI process is inherited from the communicator that contributes the local group. Note that this can result in different MPI 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.*)

7.6.3 Inter-Communication Examples

Example 1: Three-Group “Pipeline”

As shown in Figure 7.4, 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.

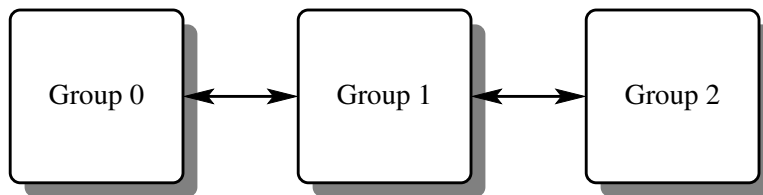


Figure 7.4: Three-group pipeline

Example 7.12.

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
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);
    }
}

```

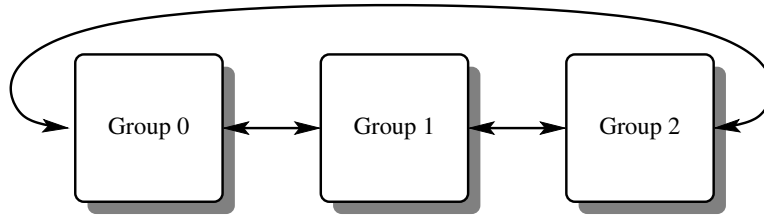


Figure 7.5: Three-group ring

```

11     break;
12 }
13 MPI_Finalize();
14 return 0;
15 }
  
```

Example 2: Three-Group “Ring”

As shown in Figure 7.5, groups 0 and 1 communicate. Groups 1 and 2 communicate. Groups 0 and 2 communicate. Therefore, each requires two inter-communicators.

Example 7.13.

```

23 int main(int argc, char *argv[])
24 {
25     MPI_Comm    myComm;        /* intra-communicator of local sub-group */
26     MPI_Comm    myFirstComm; /* inter-communicators */
27     MPI_Comm    mySecondComm;
28     int         membershipKey;
29     int         rank;
30
31     MPI_Init(&argc, &argv);
32     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
33     ...
34
35     /* User code must generate membershipKey in the range [0, 1, 2] */
36     membershipKey = rank % 3;
37
38     /* Build intra-communicator for local sub-group */
39     MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);
40
41     /* Build inter-communicators. Tags are hard-coded. */
42     if (membershipKey == 0)
43     {
44         /* Group 0 communicates with groups 1 and 2. */
45         MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1,
46                             1, &myFirstComm);
47         MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 2,
48                             2, &mySecondComm);
49     }
50     else if (membershipKey == 1)
51     {
52         /* 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;
}

```

7.7 Caching

MPI provides a “caching” facility that allows an application to attach arbitrary pieces of information, called **attributes**, to three kinds of MPI objects: communicators, windows, and datatypes. More precisely, the caching facility allows a portable library to do the following:

- pass information between calls by associating it with an MPI intra- or inter-communicator, window, or datatype,
- quickly retrieve that information, and
- be guaranteed that out-of-date information is never retrieved, even if the object is freed and its handle subsequently reused by MPI.

The caching capabilities, in some form, are required by built-in MPI routines such as collective communication and application topology. Defining an interface to these capabilities as part of the MPI standard is valuable because it permits routines like collective communication and application topologies to be implemented as portable code, and also because it makes MPI more extensible by allowing user-written routines to use standard MPI calling sequences.

Advice to users. The communicator `MPI_COMM_SELF` is a suitable choice for posting MPI 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

1 and should only be allowed when it is clearly needed and the increased cost is modest.
2 This is the reason that windows and datatypes were added but not other handles.
3 (*End of rationale.*)
4

5 One difficulty is the potential for size differences between Fortran integers and C
6 pointers. For this reason, the Fortran versions of these routines use integers of kind
7 MPI_ADDRESS_KIND.

8 *Advice to implementors.* High-quality implementations should raise an error when
9 a keyval that was created by a call to MPI_XXX_CREATE_KEYVAL is used with an
10 object of the wrong type with a call to MPI_YYY_GET_ATTR, MPI_YYY_SET_ATTR,
11 MPI_YYY_DELETE_ATTR, or MPI_YYY_FREE_KEYVAL. To do so, it is necessary to
12 maintain, with each keyval, information on the type of the associated user function.
13 (*End of advice to implementors.*)
14

15 7.7.1 Functionality

16 Attributes can be attached to communicators, windows, and datatypes. Attributes are
17 local to the MPI process and specific to the communicator to which they are attached.
18 Attributes are not propagated by MPI from one communicator to another except when the
19 communicator is duplicated using MPI_COMM_DUP, MPI_COMM_IDUP,
20 MPI_COMM_DUP_WITH_INFO, and MPI_COMM_IDUP_WITH_INFO (and even then the
21 application must give specific permission through callback functions for the attribute to be
22 copied. Please refer to Section 7.4.2 and Section 7.7.2 for attributes propagation rules).
23
24

25 *Advice to users.* Attributes in C are of type void*. Typically, such an attribute will
26 be a pointer to a structure that contains further information, or a handle to an MPI
27 object. In Fortran, attributes are of type INTEGER. Such attribute can be a handle to
28 an MPI object, or just an integer-valued attribute. (*End of advice to users.*)
29

30 *Advice to implementors.* Attributes are scalar values, equal in size to, or larger than
31 a C-language pointer. Attributes can always hold an MPI handle. (*End of advice to
32 implementors.*)
33

34 The caching interface defined here requires that attributes be stored by MPI opaquely
35 within a communicator, window, or datatype. Accessor functions include the following:

- 36 • obtain a key value (used to identify an attribute); the user specifies “callback” func-
37 tions by which MPI informs the application when the communicator is destroyed or
38 copied.
39
- 40 • store and retrieve the value of an attribute;
41

42 *Advice to implementors.* Caching and callback functions are only called synchronously,
43 in response to explicit application requests. This avoids problems that result from re-
44 peated crossings between user and system space. (This synchronous calling rule is a
45 general property of MPI.)

46 The choice of key values is under control of MPI. This allows MPI to optimize its
47 implementation of attribute sets. It also avoids conflict between independent modules
48 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 MPI process local.

7.7.2 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)
```

Fortran 2008 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
```

Fortran 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 an MPI 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:

```

1  typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm, int comm_keyval,
2      void *extra_state, void *attribute_val_in,
3      void *attribute_val_out, int *flag);
4

```

and

```

5  typedef int MPI_Comm_delete_attr_function(MPI_Comm comm, int comm_keyval,
6      void *attribute_val, void *extra_state);
7

```

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:

```

10 ABSTRACT INTERFACE

```

```

11   SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,
12       attribute_val_in, attribute_val_out, flag, ierror)
13   TYPE(MPI_Comm) :: oldcomm
14   INTEGER :: comm_keyval, ierror
15   INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
16       attribute_val_out
17   LOGICAL :: flag
18

```

and

```

19 ABSTRACT INTERFACE

```

```

20   SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval, attribute_val,
21       extra_state, ierror)
22   TYPE(MPI_Comm) :: comm
23   INTEGER :: comm_keyval, ierror
24   INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
25

```

With the `mpi` module and `mpif.h`, the Fortran callback functions are:

```

26 SUBROUTINE COMM_COPY_ATTR_FUNCTION(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
27     ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
28   INTEGER OLDCOMM, COMM_KEYVAL, IERROR
29   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
30     ATTRIBUTE_VAL_OUT
31   LOGICAL FLAG
32

```

and

```

33
34 SUBROUTINE COMM_DELETE_ATTR_FUNCTION(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
35     EXTRA_STATE, IERROR)
36   INTEGER COMM, COMM_KEYVAL, IERROR
37   INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
38

```

The `comm_copy_attr_fn` function is invoked when a communicator is duplicated by `MPI_COMM_DUP`, `MPI_COMM_IDUP`, `MPI_COMM_DUP_WITH_INFO` or `MPI_COMM_IDUP_WITH_INFO`. `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 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

1 MPI_COMM_NULL_DELETE_FN are defined in the `mpi` module (and `mpif.h`) and the
 2 `mpi_f08` module with the same name, but with different interfaces. Each function can
 3 coexist twice with the same name in the same MPI library, one routine as an implicit
 4 interface outside of the `mpi` module, i.e., declared as `EXTERNAL`, and the other routine
 5 within `mpi_f08` declared with `CONTAINS`. These routines have different link names,
 6 which are also different to the link names used for the routines used in C. (*End of*
 7 *advice to implementors.*)

8
 9 *Advice to users.* Callbacks, including the predefined Fortran functions
 10 `MPI_COMM_NULL_COPY_FN`, `MPI_COMM_DUP_FN`, and
 11 `MPI_COMM_NULL_DELETE_FN` should not be passed from one application routine
 12 that uses the `mpi_f08` module to another application routine that uses the `mpi` module
 13 or `mpif.h`, and vice versa; see also the advice to users on page 809. (*End of advice to*
 14 *users.*)

15
 16
 17 `MPI_COMM_FREE_KEYVAL(comm_keyval)`

18 INOUT `comm_keyval` key value (integer)

19
 20
 21 **C binding**

22 `int MPI_Comm_free_keyval(int *comm_keyval)`

23 **Fortran 2008 binding**

24 `MPI_Comm_free_keyval(comm_keyval, ierror)`

25 INTEGER, INTENT(INOUT) :: `comm_keyval`

26 INTEGER, OPTIONAL, INTENT(OUT) :: `ierror`

27
 28 **Fortran binding**

29 `MPI_COMM_FREE_KEYVAL(COMM_KEYVAL, IERROR)`

30 INTEGER COMM_KEYVAL, IERROR

31 Frees an extant attribute key. This function sets the value of `keyval` to
 32 `MPI_KEYVAL_INVALID`. Note that it is not erroneous to free an attribute key that is in use,
 33 because the actual free does not transpire until after all references (in other communicators
 34 on the MPI process) to the key have been freed. These references need to be explicitly freed
 35 by the program, either via calls to `MPI_COMM_DELETE_ATTR` that free one attribute
 36 instance, or by calls to `MPI_COMM_FREE` that free all attribute instances associated with
 37 the freed communicator.
 38

39
 40 `MPI_COMM_SET_ATTR(comm, comm_keyval, attribute_val)`

41 INOUT `comm` communicator to which attribute will be attached
 42 (handle)

43 IN `comm_keyval` key value (integer)

44 IN `attribute_val` attribute value

45
 46
 47 **C binding**

48 `int MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void *attribute_val)`

Fortran 2008 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

```

Fortran 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 <code>flag = false</code>
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)

```

Fortran 2008 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

```

Fortran 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)
```

Fortran 2008 binding

```
MPI_Comm_delete_attr(comm, comm_keyval, ierror)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, INTENT(IN) :: comm_keyval
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_COMM_DELETE_ATTR(COMM, COMM_KEYVAL, IERROR)
  INTEGER COMM, COMM_KEYVAL, IERROR
```

Delete attribute from cache by key. This function invokes the attribute delete function comm_delete_attr_fn specified when the keyval was created. The call will fail if the comm_delete_attr_fn function returns an error code other than MPI_SUCCESS.

Whenever a communicator is replicated using the function MPI_COMM_DUP, MPI_COMM_IDUP, MPI_COMM_DUP_WITH_INFO or MPI_COMM_IDUP_WITH_INFO, all call-back copy functions for attributes that are currently set are invoked (in arbitrary order). Whenever a communicator is deleted using the function MPI_COMM_FREE all callback delete functions for attributes that are currently set are invoked.

7.7.3 Windows

The functions for caching on windows are:

MPI_WIN_CREATE_KEYVAL(win_copy_attr_fn, win_delete_attr_fn, win_keyval, extra_state) 1

IN	win_copy_attr_fn	copy callback function for win_keyval (function)	3
IN	win_delete_attr_fn	delete callback function for win_keyval (function)	4
OUT	win_keyval	key value for future access (integer)	5
IN	extra_state	extra state for callback function	6

C binding 7

```
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) 8
```

Fortran 2008 binding 9

```
MPI_Win_create_keyval(win_copy_attr_fn, win_delete_attr_fn, win_keyval,
                      extra_state, ierror) 10
PROCEDURE(MPI_Win_copy_attr_function) :: win_copy_attr_fn 11
PROCEDURE(MPI_Win_delete_attr_function) :: win_delete_attr_fn 12
INTEGER, INTENT(OUT) :: win_keyval 13
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state 14
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 15
```

Fortran binding 16

```
MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,
                      EXTRA_STATE, IERROR) 17
EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN 18
INTEGER WIN_KEYVAL, IERROR 19
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE 20
```

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 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
```

```

1  SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state,
2      attribute_val_in, attribute_val_out, flag, ierror)
3      TYPE(MPI_Win) :: oldwin
4      INTEGER :: win_keyval, ierror
5      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
6          attribute_val_out
7      LOGICAL :: flag

```

8 and

9 ABSTRACT INTERFACE

```

10  SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,
11      extra_state, ierror)
12      TYPE(MPI_Win) :: win
13      INTEGER :: win_keyval, ierror
14      INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state

```

16 With the mpi module and mpif.h, the Fortran callback functions are:

```

17  SUBROUTINE WIN_COPY_ATTR_FUNCTION(OLDWIN, WIN_KEYVAL, EXTRA_STATE,
18      ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
19      INTEGER OLDWIN, WIN_KEYVAL, IERROR
20      INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
21          ATTRIBUTE_VAL_OUT
22      LOGICAL FLAG

```

23 and

```

24  SUBROUTINE WIN_DELETE_ATTR_FUNCTION(WIN, WIN_KEYVAL, ATTRIBUTE_VAL,
25      EXTRA_STATE, IERROR)
26      INTEGER WIN, WIN_KEYVAL, IERROR
27      INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE

```

29 If an attribute copy function or attribute delete function returns other than
30 MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_WIN_FREE), is
31 erroneous.

33 MPI_WIN_FREE_KEYVAL(win_keyval)

34 INOUT win_keyval key value (integer)

37 C binding

38 int MPI_Win_free_keyval(int *win_keyval)

40 Fortran 2008 binding

```

41  MPI_Win_free_keyval(win_keyval, ierror)
42      INTEGER, INTENT(INOUT) :: win_keyval
43      INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

44 Fortran binding

```

45  MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)
46      INTEGER WIN_KEYVAL, IERROR

```

48

MPI_WIN_SET_ATTR(win, win_keyval, attribute_val)			1
INOUT	win	window to which attribute will be attached (handle)	2
IN	win_keyval	key value (integer)	3
IN	attribute_val	attribute value	4
			5
			6

C binding

```
int MPI_Win_set_attr(MPI_Win win, int win_keyval, void *attribute_val)
```

Fortran 2008 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
```

Fortran binding

```
MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR)
  INTEGER WIN, WIN_KEYVAL, IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
```

MPI_WIN_GET_ATTR(win, win_keyval, attribute_val, flag)

IN	win	window to which the attribute is attached (handle)	24
IN	win_keyval	key value (integer)	25
OUT	attribute_val	attribute value, unless flag = false	26
OUT	flag	false if no attribute is associated with the key (logical)	27

C binding

```
int MPI_Win_get_attr(MPI_Win win, int win_keyval, void *attribute_val,
                    int *flag)
```

Fortran 2008 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
```

Fortran 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
```

```

1 MPI_WIN_DELETE_ATTR(win, win_keyval)
2     INOUT    win                window from which the attribute is deleted (handle)
3
4     IN      win_keyval          key value (integer)
5

```

C binding

```

7 int MPI_Win_delete_attr(MPI_Win win, int win_keyval)
8

```

Fortran 2008 binding

```

9 MPI_Win_delete_attr(win, win_keyval, ierror)
10     TYPE(MPI_Win), INTENT(IN) :: win
11     INTEGER, INTENT(IN) :: win_keyval
12     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13

```

Fortran binding

```

14 MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR)
15     INTEGER WIN, WIN_KEYVAL, IERROR
16
17

```

7.7.4 Datatypes

The new functions for caching on datatypes are:

```

23 MPI_TYPE_CREATE_KEYVAL(type_copy_attr_fn, type_delete_attr_fn, type_keyval,
24     extra_state)
25     IN      type_copy_attr_fn    copy callback function for type_keyval (function)
26     IN      type_delete_attr_fn  delete callback function for type_keyval (function)
27     OUT     type_keyval          key value for future access (integer)
28     IN      extra_state          extra state for callback function
29
30

```

C binding

```

32 int MPI_Type_create_keyval(MPI_Type_copy_attr_function *type_copy_attr_fn,
33     MPI_Type_delete_attr_function *type_delete_attr_fn,
34     int *type_keyval, void *extra_state)
35

```

Fortran 2008 binding

```

37 MPI_Type_create_keyval(type_copy_attr_fn, type_delete_attr_fn, type_keyval,
38     extra_state, ierror)
39     PROCEDURE(MPI_Type_copy_attr_function) :: type_copy_attr_fn
40     PROCEDURE(MPI_Type_delete_attr_function) :: type_delete_attr_fn
41     INTEGER, INTENT(OUT) :: type_keyval
42     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
43     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44

```

Fortran binding

```

45 MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL,
46     EXTRA_STATE, IERROR)
47     EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN
48

```

```

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 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

```

1 INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE

2

3 If an attribute copy function or attribute delete function returns other than
4 MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_TYPE_FREE),
5 is erroneous.

6

7 MPI_TYPE_FREE_KEYVAL(type_keyval)

8

9 INOUT type_keyval key value (integer)

10

11 **C binding**

12 int MPI_Type_free_keyval(int *type_keyval)

13

14 **Fortran 2008 binding**

15 MPI_Type_free_keyval(type_keyval, ierror)

16

17 INTEGER, INTENT(INOUT) :: type_keyval

18

19 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

20 **Fortran binding**

21 MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)

22

23 INTEGER TYPE_KEYVAL, IERROR

24

25 MPI_TYPE_SET_ATTR(datatype, type_keyval, attribute_val)

26

27 INOUT datatype datatype to which attribute will be attached (handle)

28

29 IN type_keyval key value (integer)

30

31 IN attribute_val attribute value

32

33 **C binding**

34 int MPI_Type_set_attr(MPI_Datatype datatype, int type_keyval,

35

36 void *attribute_val)

37

38 **Fortran 2008 binding**

39 MPI_Type_set_attr(datatype, type_keyval, attribute_val, ierror)

40

41 TYPE(MPI_Datatype), INTENT(IN) :: datatype

42

43 INTEGER, INTENT(IN) :: type_keyval

44

45 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val

46

47 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

48

49 **Fortran binding**

50 MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)

51

52 INTEGER DATATYPE, TYPE_KEYVAL, IERROR

53

54 INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL

55

56

57

58

59

60

61

62

MPI_TYPE_GET_ATTR(datatype, type_keyval, attribute_val, flag)				1
IN	datatype	datatype to which the attribute is attached (handle)		2
IN	type_keyval	key value (integer)		3
OUT	attribute_val	attribute value, unless flag = false		4
OUT	flag	false if no attribute is associated with the key (logical)		5
				6
				7
				8
				9

C binding

```
int MPI_Type_get_attr(MPI_Datatype datatype, int type_keyval,
                     void *attribute_val, int *flag)
```

Fortran 2008 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
```

Fortran 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)		28
IN	type_keyval	key value (integer)		29
				30
				31
				32

C binding

```
int MPI_Type_delete_attr(MPI_Datatype datatype, int type_keyval)
```

Fortran 2008 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
```

Fortran binding

```
MPI_TYPE_DELETE_ATTR(DATATYPE, TYPE_KEYVAL, IERROR)
  INTEGER DATATYPE, TYPE_KEYVAL, IERROR
```

7.7.5 Error Class for Invalid Keyval

Key values for attributes are system-allocated, by MPI_{XXX}_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_{XXX}_DELETE_ATTR`, `MPI_{XXX}_SET_ATTR`, `MPI_{XXX}_GET_ATTR`, `MPI_{XXX}_FREE_KEYVAL`, `MPI_COMM_DUP`, `MPI_COMM_IDUP`, `MPI_COMM_DUP_WITH_INFO`, `MPI_COMM_IDUP_WITH_INFO`, `MPI_COMM_DISCONNECT`, and `MPI_COMM_FREE`. The last six are included because `keyval` is an argument to the copy and delete functions for attributes.

7.7.6 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.*)

```

16 /* key for this module's stuff: */
17 static int gop_key = MPI_KEYVAL_INVALID;
18
19 typedef struct
20 {
21     int ref_count;          /* reference count */
22     /* other stuff, whatever else we want */
23 } gop_stuff_type;
24
25 void Efficient_Collective_Op(MPI_Comm comm, ...)
26 {
27     gop_stuff_type *gop_stuff;
28     MPI_Group      group;
29     int            foundflag;
30
31     MPI_Comm_group(comm, &group);
32
33     if (gop_key == MPI_KEYVAL_INVALID) /* get a key on first call ever */
34     {
35         if ( ! MPI_Comm_create_keyval(gop_stuff_copier,
36                                     gop_stuff_destructor,
37                                     &gop_key, NULL)) {
38             /* get the key while assigning its copy and delete callback
39              behavior. */
40         } else
41             MPI_Abort(comm, 99);
42     }
43
44     MPI_Comm_get_attr(comm, gop_key, &gop_stuff, &foundflag);
45     if (foundflag)
46     { /* This module has executed in this group before.
47        We will use the cached information */
48     }
49     else
50     { /* This is a group that we have not yet cached anything in.
51        We will now do so.
52        */

```

```

1
2  /* First, allocate storage for the stuff we want,
3     and initialize the reference count */
4
5  gop_stuff = (gop_stuff_type *) malloc(sizeof(gop_stuff_type));
6  if (gop_stuff == NULL) { /* abort on out-of-memory error */ }
7
8  gop_stuff->ref_count = 1;
9
10 /* Second, fill in *gop_stuff with whatever we want.
11    This part isn't shown here */
12
13 /* Third, store gop_stuff as the attribute value */
14 MPI_Comm_set_attr(comm, gop_key, gop_stuff);
15 }
16 /* Then, in any case, use contents of *gop_stuff
17    to do the global op ... */
18 }
19
20 /* The following routine is called by MPI when a group is freed */
21
22 int gop_stuff_destructor(MPI_Comm comm, int keyval, void *gop_stuffP,
23                        void *extra)
24 {
25     gop_stuff_type *gop_stuff = (gop_stuff_type *)gop_stuffP;
26     if (keyval != gop_key) { /* abort -- programming error */ }
27
28     /* The group's being freed removes one reference to gop_stuff */
29     gop_stuff->ref_count -= 1;
30
31     /* If no references remain, then free the storage */
32     if (gop_stuff->ref_count == 0) {
33         free((void *)gop_stuff);
34     }
35     return MPI_SUCCESS;
36 }
37
38 /* The following routine is called by MPI when a group is copied */
39
40 int gop_stuff_copier(MPI_Comm comm, int keyval, void *extra,
41                    void *gop_stuff_inP, void *gop_stuff_outP, int *flag)
42 {
43     gop_stuff_type *gop_stuff_in = (gop_stuff_type *)gop_stuff_inP;
44     gop_stuff_type **gop_stuff_out = (gop_stuff_type **)gop_stuff_outP;
45     if (keyval != gop_key) { /* abort -- programming error */ }
46
47     /* The new group adds one reference to this gop_stuff */
48     gop_stuff_in->ref_count += 1;
49     *gop_stuff_out = gop_stuff_in;
50     return MPI_SUCCESS;
51 }

```

7.8 Naming Objects

There are many occasions on which it would be useful to allow a user to associate a printable identifier with an MPI communicator, window, or datatype, for instance error reporting, debugging, and profiling. The names attached to opaque objects do not propagate when the object is duplicated or copied by MPI routines. For communicators this can be achieved using the following two functions.

```
MPI_COMM_SET_NAME(comm, comm_name)
```

INOUT	comm	communicator whose identifier is to be set (handle)
IN	comm_name	the character string that is remembered as the name (string)

C binding

```
int MPI_Comm_set_name(MPI_Comm comm, const char *comm_name)
```

Fortran 2008 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
```

Fortran 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 that 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 (noncollective) operation, which only affects the name of the communicator as seen in the MPI process that made the MPI_COMM_SET_NAME call. There is no requirement that the same (or any) name be assigned to a communicator in every MPI 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 MPI processes where it exists, to avoid confusion. (*End of advice to users.*)

The length of the name that 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 that 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 that 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	<code>comm</code>	communicator whose name is to be returned (handle)
OUT	<code>comm_name</code>	the name previously stored on the communicator, or an empty string if no such name exists (string)
OUT	<code>resultlen</code>	length of returned name (integer)

C binding

```
int MPI_Comm_get_name(MPI_Comm comm, char *comm_name, int *resultlen)
```

Fortran 2008 binding

```
MPI_Comm_get_name(comm, comm_name, resultlen, ierror)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: comm_name
  INTEGER, INTENT(OUT) :: resultlen
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran 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 that has previously been associated with the given communicator. The name may be set and retrieved from any language. The same name will be returned independent of the language used. `comm_name` should be allocated so that it can hold a resulting string of length `MPI_MAX_OBJECT_NAME` characters. `MPI_COMM_GET_NAME` returns a copy of the set name in `comm_name`.

In C, a null character is additionally stored at `comm_name[resultlen]`. The value of `resultlen` cannot be larger than `MPI_MAX_OBJECT_NAME-1`. In Fortran, `comm_name` is padded on the right with blank characters. The value of `resultlen` cannot be larger than `MPI_MAX_OBJECT_NAME`.

If the user has not associated a name with a communicator, or an error occurs, `MPI_COMM_GET_NAME` will return an empty string (all spaces in Fortran, "" in C). The three predefined communicators will have predefined names associated with them. Thus, the names of `MPI_COMM_WORLD`, `MPI_COMM_SELF`, and the communicator returned by `MPI_COMM_GET_PARENT` (if not `MPI_COMM_NULL`) will have the default of "MPI_COMM_WORLD", "MPI_COMM_SELF", and "MPI_COMM_PARENT". Passing `MPI_COMM_NULL` as `comm` will return the string "MPI_COMM_NULL". 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 that 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 that is remembered as the name (string)

C binding

```
int MPI_Type_set_name(MPI_Datatype datatype, const char *type_name)
```

Fortran 2008 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
```

Fortran binding

```
MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR)
  INTEGER DATATYPE, IERROR
  CHARACTER*(*) TYPE_NAME
```

MPI_TYPE_GET_NAME(datatype, type_name, resultlen)			1
IN	datatype	datatype whose name is to be returned (handle)	2
OUT	type_name	the name previously stored on the datatype, or an empty string if no such name exists (string)	3
OUT	resultlen	length of returned name (integer)	4
			5
			6
			7

C binding

```
int MPI_Type_get_name(MPI_Datatype datatype, char *type_name, int *resultlen)
```

Fortran 2008 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
```

Fortran 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". Passing MPI_DATATYPE_NULL as datatype will return the string "MPI_DATATYPE_NULL".

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 that is remembered as the name (string)
----	----------	--

C binding

```
int MPI_Win_set_name(MPI_Win win, const char *win_name)
```

Fortran 2008 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
```

Fortran binding

```
MPI_WIN_SET_NAME(WIN, WIN_NAME, IERROR)
```

```
INTEGER WIN, IERROR
```

```
CHARACTER*(*) WIN_NAME
```

```

1 MPI_WIN_GET_NAME(win, win_name, resultlen)
2     IN      win                window whose name is to be returned (handle)
3
4     OUT     win_name           the name previously stored on the window, or an
5                                 empty string if no such name exists (string)
6
7     OUT     resultlen          length of returned name (integer)

```

C binding

```

9 int MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)

```

Fortran 2008 binding

```

12 MPI_Win_get_name(win, win_name, resultlen, ierror)
13     TYPE(MPI_Win), INTENT(IN) :: win
14     CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: win_name
15     INTEGER, INTENT(OUT) :: resultlen
16     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

18 MPI_WIN_GET_NAME(WIN, WIN_NAME, RESULTLEN, IERROR)
19     INTEGER WIN, RESULTLEN, IERROR
20     CHARACTER*(*) WIN_NAME
21
22     Passing MPI_WIN_NULL as win will return the string "MPI_WIN_NULL".

```

7.9 Formalizing the Loosely Synchronous Model

In this section, we make further statements about the loosely synchronous model, with particular attention to intra-communication.

7.9.1 Basic Statements

When a caller passes a communicator (that contains a context and group) to a callee, that 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 MPI 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.

7.9.2 Models of Execution

In the loosely synchronous model, transfer of control to a **parallel procedure** is effected by having each executing MPI process invoke the procedure. The invocation is a collective

operation: it is executed by all MPI processes in the execution group, and invocations are similarly ordered at all MPI processes. However, the invocation need not be synchronized.

We say that a parallel procedure is *active* in an MPI process if the MPI 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 an MPI process, then this MPI 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 MPI process, and the group of executing MPI processes is fixed. For example, all invocations of parallel procedures involve all MPI processes, MPI 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, MPI 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

1 sure that, should two distinct parallel procedures be invoked concurrently on overlapping
2 sets of MPI processes, communicator creation is properly coordinated.
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

DRAFT

Chapter 8

Process Topologies

8.1 Introduction

This chapter discusses the MPI *virtual topology* mechanism. A *virtual topology* is an extra, optional attribute that one can give to an intra-communicator; *virtual topologies* cannot be added to inter-communicators. A *virtual topology* can provide a convenient naming mechanism for the MPI processes of a group (within a communicator), and additionally, may assist the runtime system in mapping the processes onto hardware.

As stated in Chapter 7, a group in MPI is an ordered set of n process identifiers (henceforth MPI processes). Each MPI process in the group is assigned a rank between 0 and $n-1$. In many parallel applications a linear ranking of MPI processes does not adequately reflect the logical communication pattern of the MPI processes (which is usually determined by the underlying problem geometry and the numerical algorithm used). Often the MPI processes are arranged in topological patterns such as two- or three-dimensional grids. More generally, the logical MPI process arrangement is described by a graph. In this chapter we will refer to this logical MPI process arrangement as the *virtual topology*.

A clear distinction must be made between the *virtual 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 topologies*.

Rationale. Though physical mapping is not discussed, the existence of the *virtual topology* information may be used as advice by the runtime system. There are well-known techniques for mapping grid/torus structures to hardware topologies such as hypercubes or grids. For more complicated graph structures good heuristics often yield nearly optimal results [49]. On the other hand, if there is no way for the user to specify the logical process arrangement as a *virtual topology*, a random mapping is most likely to result. On some machines, this will lead to unnecessary contention in the interconnection network. Some details about predicted and measured performance improvements that result from good process-to-processor mapping on modern wormhole-routing architectures can be found in [12, 13].

Besides possible performance benefits, the *virtual topology* can function as a convenient, process-naming structure, with significant benefits for program readability and notational power in message-passing programming. (*End of rationale.*)

8.2 Virtual Topologies

The communication pattern of a set of MPI processes can be represented by a graph. The nodes represent MPI processes, and the edges connect MPI processes that communicate with each other. MPI provides message-passing between any pair of MPI processes in a group. There is no requirement for opening a channel explicitly. Therefore, a “missing link” in the user-defined graph of MPI processes does not prevent the corresponding MPI processes from exchanging messages. It means rather that this connection is neglected in the *virtual topology*. This strategy implies that the *virtual 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 MPI 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 MPI 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.

The coordinates of MPI processes in a Cartesian structure begin their numbering at 0. Row-major numbering is always used for the MPI processes in a Cartesian structure. This means that, for example, for four MPI processes in a (2×2) grid, the relationship between their ranks in the group and their coordinates in the *virtual topology* is as follows:

```
coord (0,0):  rank 0
coord (0,1):  rank 1
coord (1,0):  rank 2
coord (1,1):  rank 3
```

8.3 Embedding in MPI

The support for *virtual topologies* as defined in this chapter is consistent with other parts of MPI, and, whenever possible, makes use of functions that are defined elsewhere. Topology information is associated with communicators. It is added to communicators using the caching mechanism described in Chapter 7.

Information representing a *virtual topology* may be added to a communicator at the time of its creation. If a communicator creation function adds information representing a *virtual topology* to the output communicator it creates, then it either propagates the topology representation from the input communicator to the output communicator, or adds a new topology representation generated from the input parameters that describe a *virtual topology*. The description of every MPI communicator creation function explicitly states how topology information is handled. Communicator creation functions that create new topology representations are described in Section 8.5.

8.4 Overview of the Functions

MPI supports three types of *virtual topology*: **Cartesian**, **graph**, and **distributed graph**. The function `MPI_CART_CREATE` can be used to create Cartesian topologies, the function `MPI_GRAPH_CREATE` can be used to create graph topologies, and the functions `MPI_DIST_GRAPH_CREATE_ADJACENT` and `MPI_DIST_GRAPH_CREATE` can be used to create distributed graph topologies. These topology creation functions are collective. As with other collective calls, the program must be written to work correctly, whether the call synchronizes or not.

The above topology creation functions take as input an existing communicator `comm_old`, which defines the set of MPI 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 MPI processes of the group of `comm_old`. When calling `MPI_GRAPH_CREATE`, each MPI 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 MPI process only specifies a subset of the edges in the graph such that the entire graph structure is defined collectively across the set of MPI processes. Therefore the MPI processes provide different values for the arguments specifying the graph. However, all MPI processes must give the same value for `reorder` and the `info` argument. In all cases, a new communicator `comm_topol` is created that carries the topological structure as cached information (see Chapter 7). In analogy to function `MPI_COMM_CREATE`, no cached information propagates from `comm_old` to `comm_topol`.

`MPI_CART_CREATE` can be used to describe Cartesian structures of arbitrary dimension. For each coordinate direction one specifies whether the MPI 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 MPI 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` retrieve the graph topology information that is associated with the communicator. 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 MPI process. For a Cartesian topology, the function `MPI_CARTDIM_GET` returns the number of dimensions and `MPI_CART_GET` returns the numbers of MPI processes in each dimension and periodicity of the associated Cartesian topology. Additionally, the functions `MPI_CART_RANK` and `MPI_CART_COORDS` translate Cartesian coordinates into a group rank, and vice-versa. The function `MPI_CART_SHIFT` provides the information needed to communicate with neighbors along a Cartesian dimension. All of these query functions are local.

For Cartesian topologies, the function `MPI_CART_SUB` can be used to extract a Cartesian subspace (analogous to `MPI_COMM_SPLIT`). This function is collective over the input communicator's group.

The two additional functions, `MPI_GRAPH_MAP` and `MPI_CART_MAP`, are, in gen-

eral, not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 7, they are sufficient to implement all other topology functions. Section 8.5.8 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`.

8.5 Topology Constructors

8.5.1 Cartesian Constructor

`MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)`

IN	<code>comm_old</code>	input communicator (handle)
IN	<code>ndims</code>	number of dimensions of Cartesian grid (integer)
IN	<code>dims</code>	integer array of size <code>ndims</code> specifying the number of processes in each dimension
IN	<code>periods</code>	logical array of size <code>ndims</code> specifying whether the grid is periodic (<code>true</code>) or not (<code>false</code>) in each dimension
IN	<code>reorder</code>	ranking may be reordered (<code>true</code>) or not (<code>false</code>) (logical)
OUT	<code>comm_cart</code>	new communicator with associated Cartesian topology (handle)

C binding

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[],
                  const int periods[], int reorder, MPI_Comm *comm_cart)
```

Fortran 2008 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
```

Fortran 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 MPI 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 MPI processes return `MPI_COMM_NULL`, in analogy to `MPI_COMM_SPLIT`. If `ndims` is zero then a zero-dimensional Cartesian topology is created. The call is erroneous if it specifies a grid that is larger than the group size or if `ndims` is negative. `MPI_CART_CREATE` will associate information representing a Cartesian topology with the specified number of dimensions, numbers of MPI processes in each coordinate direction, and periodicity with the new communicator.

8.5.2 Cartesian Convenience Function: `MPI_DIMS_CREATE`

For Cartesian topologies, the function `MPI_DIMS_CREATE` helps the user select a balanced distribution of MPI processes per coordinate direction, depending on the number of MPI processes in the group to be balanced and optional constraints that can be specified by the user.

`MPI_DIMS_CREATE`(`nnodes`, `ndims`, `dims`)

IN	<code>nnodes</code>	number of nodes in a grid (integer)
IN	<code>ndims</code>	number of Cartesian dimensions (integer)
INOUT	<code>dims</code>	integer array of size <code>ndims</code> specifying the number of nodes in each dimension

C binding

```
int MPI_Dims_create(int nnodes, int ndims, int dims[])
```

Fortran 2008 binding

```
MPI_Dims_create(nnodes, ndims, dims, ierror)
  INTEGER, INTENT(IN) :: nnodes, ndims
  INTEGER, INTENT(INOUT) :: dims(ndims)
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
  INTEGER NNODES, NDIMS, DIMS(*), IERROR
```

The entries in the array `dims` are set to describe a Cartesian grid with `ndims` dimensions and a total of `nnodes` nodes. The dimensions are set to be as close to each other as possible, using an appropriate divisibility algorithm. The caller may further constrain the operation of this routine by specifying elements of array `dims`. If `dims[i]` is set to a positive number, the routine will not modify the number of nodes in dimension `i`; only those entries where `dims[i] = 0` are modified by the call.

Negative input values of `dims[i]` are erroneous. An error will occur if `nnodes` is not a multiple of

$$\prod_{i, \text{dims}[i] \neq 0} \text{dims}[i].$$

For `dims[i]` set by the call, `dims[i]` will be ordered in nonincreasing order. Array `dims` is suitable for use as input to routine `MPI_CART_CREATE`. `MPI_DIMS_CREATE` is local. If `ndims` is zero and `nnodes` is one, `MPI_DIMS_CREATE` returns `MPI_SUCCESS`.

Example 8.1. The use of the array argument `dims` in `MPI_DIMS_CREATE`.

dims before call	function call	dims on return
(0,0)	<code>MPI_DIMS_CREATE(6, 2, dims)</code>	(3,2)
(0,0)	<code>MPI_DIMS_CREATE(7, 2, dims)</code>	(7,1)
(0,3,0)	<code>MPI_DIMS_CREATE(6, 3, dims)</code>	(2,3,1)
(0,3,0)	<code>MPI_DIMS_CREATE(7, 3, dims)</code>	erroneous call

8.5.3 Graph Constructor

`MPI_GRAPH_CREATE(comm_old, nnodes, index, edges, reorder, comm_graph)`

IN	<code>comm_old</code>	input communicator (handle)
IN	<code>nnodes</code>	number of nodes in graph (integer)
IN	<code>index</code>	array of integers describing node degrees (see below)
IN	<code>edges</code>	array of integers describing graph edges (see below)
IN	<code>reorder</code>	ranking may be reordered (<code>true</code>) or not (<code>false</code>) (logical)
OUT	<code>comm_graph</code>	new communicator with associated graph topology (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)
```

Fortran 2008 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
```

Fortran 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 MPI process in the new group is identical to its rank in the old group. Otherwise, the function may reorder the MPI processes. If the size, `nnodes`, of the graph is smaller than the size of the

group of `comm_old`, then `MPI_COMM_NULL` is returned by some MPI processes, 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 MPI processes. The call is erroneous if it specifies a graph that is larger than the group size of the input communicator.

The three parameters `nnodes`, `index` and `edges` define the graph structure. `nnodes` is the number of nodes of the graph. The nodes are numbered from 0 to `nnodes-1`. The i -th entry of array `index` stores the total number of neighbors of the first i graph nodes. The lists of neighbors of nodes 0, 1, ..., `nnodes-1` are stored in consecutive locations in array `edges`. The array `edges` is a flattened representation of the edge lists. The total number of entries in `index` is `nnodes` and the total number of entries in `edges` is equal to the number of graph edges.

The definitions of the arguments `nnodes`, `index`, and `edges` are illustrated with the following simple example.

Example 8.2. Specification of the adjacency matrix for `MPI_GRAPH_CREATE`. Assume there are four MPI processes with ranks 0, 1, 2, 3 in the input communicator with the following adjacency matrix:

MPI 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 , $i=1, \dots, nnodes-1$; 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 , $i=1, \dots, nnodes-1$; 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 MPI process is allowed to be defined multiple times in the list of neighbors of an MPI process (i.e., there may be multiple edges between two MPI processes). An MPI process is also allowed to be a neighbor to itself (i.e., a self loop in the graph). The adjacency matrix is allowed to be nonsymmetric.

Advice to users. Performance implications of using multiple edges or a nonsymmetric adjacency matrix are not defined. The definition of a node-neighbor edge does not imply a direction of the communication. (*End of advice to users.*)

Advice to implementors. The following topology information is likely to be stored with a communicator:

- 1 • Type of topology (Cartesian/graph),
- 2
- 3 • For a Cartesian topology:
 - 4 1. ndims (number of dimensions),
 - 5 2. dims (numbers of MPI processes per coordinate direction),
 - 6 3. periods (periodicity information),
 - 7
 - 8 4. own_position (own position in grid, could also be computed from rank and
 - 9 dims)
- 10 • For a graph topology:
 - 11 1. index,
 - 12 2. edges,
 - 13
 - 14 which are the vectors defining the graph structure.

15 For a graph structure the number of nodes is equal to the number of MPI processes
 16 in the group. Therefore, the number of nodes does not have to be stored explicitly.
 17 An additional zero entry at the start of array `index` simplifies access to the topology
 18 information. (*End of advice to implementors.*)

20 8.5.4 Distributed Graph Constructor

21
 22 MPI_GRAPH_CREATE requires that each MPI process passes the full (global) communica-
 23 tion graph to the call. This limits the scalability of this constructor. With the distributed
 24 graph interface, the communication graph is specified in a fully distributed fashion. Each
 25 MPI process specifies only the part of the communication graph of which it is aware. Typ-
 26 ically, this could be the set of MPI processes from which the MPI process will eventually
 27 receive or get data, or the set of MPI processes to which the MPI process will send or put
 28 data, or some combination of such edges. Two different interfaces can be used to create a
 29 distributed graph topology. MPI_DIST_GRAPH_CREATE_ADJACENT creates a distributed
 30 graph communicator with each MPI process specifying each of its incoming and outgoing
 31 (adjacent) edges in the logical communication graph and thus requires minimal communi-
 32 cation during creation. MPI_DIST_GRAPH_CREATE provides full flexibility such that any
 33 MPI process can indicate that communication will occur between any pair of MPI processes
 34 in the graph.

35 To provide better possibilities for optimization by the MPI library, the distributed
 36 graph constructors permit weighted communication edges and take an `info` argument that
 37 can further influence process reordering or other optimizations performed by the MPI library.
 38 For example, hints can be provided on how edge weights are to be interpreted, the quality
 39 of the reordering, and/or the time permitted for the MPI library to process the graph.

40
 41
 42 MPI_DIST_GRAPH_CREATE_ADJACENT(comm_old, indegree, sources, sourceweights,
 43 outdegree, destinations, destweights, info, reorder, comm_dist_graph)

44 IN comm_old input communicator (handle)

45 IN indegree size of sources and sourceweights arrays (non-negative

46 integer)

47

48

IN	sources	ranks of MPI processes for which the calling process is a destination (array of non-negative integers)	1 2
IN	sourceweights	weights of the edges into the calling MPI process (array of non-negative integers)	3 4 5
IN	outdegree	size of destinations and destweights arrays (non-negative integer)	6 7
IN	destinations	ranks of MPI processes for which the calling MPI process is a source (array of non-negative integers)	8 9
IN	destweights	weights of the edges out of the calling MPI process (array of non-negative integers)	10 11 12
IN	info	hints on optimization and interpretation of weights (handle)	13 14
IN	reorder	the ranks may be reordered (<i>true</i>) or not (<i>false</i>) (logical)	15 16
OUT	comm_dist_graph	new communicator with associated distributed graph topology (handle)	17 18 19

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)
```

Fortran 2008 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
```

Fortran 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 MPI process passes all information about its incoming and outgoing edges in the virtual distributed graph topology. The calling MPI processes must ensure that each edge of the graph is described

1 in the source and in the destination process with the same weights. If there are multiple
 2 edges for a given (source,dest) pair, then the sequence of the weights of these edges does not
 3 matter. The complete communication topology is the combination of all edges shown in the
 4 sources arrays of all MPI processes in `comm_old`, which must be identical to the combination
 5 of all edges shown in the destinations arrays. Source and destination nodes must be given
 6 as the ranks of the respective MPI processes in `comm_old`. This allows a fully distributed
 7 specification of the communication graph. Isolated MPI processes (i.e., MPI processes with
 8 no outgoing or incoming edges, that is, MPI processes that have specified `indegree` and
 9 `outdegree` as zero and thus do not occur as source or destination in the graph specification)
 10 are allowed.

11 The call creates a new communicator `comm_dist_graph` of distributed graph topology
 12 type to which topology information has been attached. The number of MPI processes in
 13 `comm_dist_graph` is identical to the number of MPI processes in `comm_old`. The call to
 14 `MPI_DIST_GRAPH_CREATE_ADJACENT` is collective.

15 Weights are specified as nonnegative integers and can be used to influence the process
 16 mapping strategy and other internal MPI optimizations. For instance, approximate count
 17 arguments of later communication calls along specific edges could be used as their edge
 18 weights. Multiplicity of edges can likewise indicate more intense communication between
 19 pairs of MPI processes. However, the exact meaning of edge weights is not specified by
 20 the MPI standard and is left to the implementation. In C or Fortran, an application can
 21 supply the special value `MPI_UNWEIGHTED` for the weight array to indicate that all edges
 22 have the same (effectively no) weight. It is erroneous to supply `MPI_UNWEIGHTED` for some
 23 but not all MPI processes of `comm_old`. If the graph is weighted but `indegree` or `outdegree`
 24 is zero, then `MPI_WEIGHTS_EMPTY` or any arbitrary array may be passed to `sourceweights`
 25 or `destweights` respectively. Note that `MPI_UNWEIGHTED` and `MPI_WEIGHTS_EMPTY` are
 26 not special weight values; rather they are special values for the total array argument. In
 27 Fortran, `MPI_UNWEIGHTED` and `MPI_WEIGHTS_EMPTY` are objects like `MPI_BOTTOM` (not
 28 usable for initialization or assignment). See Section 2.5.4.

29
 30 *Advice to users.* In the case of an empty weights array argument passed while
 31 constructing a weighted graph, one should not pass `NULL` because the value of
 32 `MPI_UNWEIGHTED` may be equal to `NULL`. The value of this argument would then
 33 be indistinguishable from `MPI_UNWEIGHTED` to the implementation. In this case
 34 `MPI_WEIGHTS_EMPTY` should be used instead. (*End of advice to users.*)

35
 36 *Advice to implementors.* It is recommended that `MPI_UNWEIGHTED` not be imple-
 37 mented as `NULL`. (*End of advice to implementors.*)

38
 39 *Rationale.* To ensure backward compatibility, `MPI_UNWEIGHTED` may still be imple-
 40 mented as `NULL`. See Annex B.4. (*End of rationale.*)

41
 42 The meaning of the `info` and `reorder` arguments is defined in the description of the
 43 following routine.

44
 45 `MPI_DIST_GRAPH_CREATE(comm_old, n, sources, degrees, destinations, weights, info,`
 46 `reorder, comm_dist_graph)`

47
 48 IN `comm_old` input communicator (handle)

IN	n	number of source nodes for which this MPI process specifies edges (non-negative integer)	1 2
IN	sources	array containing the n source nodes for which this MPI process specifies edges (array of non-negative integers)	3 4 5 6
IN	degrees	array specifying the number of destinations for each source node in the source node array (array of non-negative integers)	7 8 9
IN	destinations	destination nodes for the source nodes in the source node array (array of non-negative integers)	10 11
IN	weights	weights for source to destination edges (array of non-negative integers)	12 13 14
IN	info	hints on optimization and interpretation of weights (handle)	15 16
IN	reorder	the ranks may be reordered (true) or not (false) (logical)	17 18
OUT	comm_dist_graph	new communicator with associated distributed graph topology (handle)	19 20 21

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)

```

Fortran 2008 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

```

Fortran 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 MPI process calls the constructor with a set of directed (source,destination) communication edges as described below. Every MPI process passes an array of n source nodes in the sources array. For

1 each source node, a nonnegative number of destination nodes is specified in the `degrees`
2 array. The destination nodes are stored in the corresponding consecutive segment of the
3 `destinations` array. More precisely, if the i -th node in `sources` is s , this specifies `degrees[i]` edges
4 (s,d) with d of the j -th such edge stored in `destinations[degrees[0]+...+degrees[i-1]+j]`. The
5 weight of this edge is stored in `weights[degrees[0]+...+degrees[i-1]+j]`. Both the `sources` and
6 the `destinations` arrays may contain the same node more than once, and the order in which
7 nodes are listed as destinations or sources is not significant. Similarly, different processes
8 may specify edges with the same source and destination nodes. Source and destination
9 nodes must be given as the ranks of the respective MPI processes in `comm_old`. Different
10 MPI processes may specify different numbers of source and destination nodes, as well as
11 different source to destination edges. This allows a fully distributed specification of the
12 communication graph. Isolated MPI processes (i.e., MPI processes with no outgoing or
13 incoming edges, that is, MPI processes that do not occur as source or destination node in
14 the graph specification) are allowed.

15 The call creates a new communicator `comm_dist_graph` of distributed graph topology
16 type to which topology information has been attached. The number of MPI processes in
17 `comm_dist_graph` is identical to the number of MPI processes in `comm_old`. The call to
18 `MPI_DIST_GRAPH_CREATE` is collective.

19 If `reorder = false`, all MPI processes will have the same rank in `comm_dist_graph` as in
20 `comm_old`. If `reorder = true` then the MPI library is free to remap to other MPI processes (of
21 `comm_old`) in order to improve communication on the edges of the communication graph.
22 The weight associated with each edge is a hint to the MPI library about the amount or
23 intensity of communication on that edge, and may be used to compute a “best” reordering.

24 Weights are specified as nonnegative integers and can be used to influence the MPI
25 process remapping strategy and other internal MPI optimizations. For instance, approxi-
26 mate count arguments of later communication calls along specific edges could be used as
27 their edge weights. Multiplicity of edges can likewise indicate more intense communication
28 between pairs of MPI processes. However, the exact meaning of edge weights and mul-
29 tiplicity of edges is not specified by the MPI standard and is left to the implementation.
30 In C or Fortran, an application can supply the special value `MPI_UNWEIGHTED` for the
31 weight array to indicate that all edges have the same (effectively no) weight. It is erro-
32 neous to supply `MPI_UNWEIGHTED` for some but not all MPI processes of `comm_old`. If the
33 graph is weighted but $n = 0$, then `MPI_WEIGHTS_EMPTY` or any arbitrary array may be
34 passed to `weights`. Note that `MPI_UNWEIGHTED` and `MPI_WEIGHTS_EMPTY` are not spe-
35 cial weight values; rather they are special values for the total array argument. In Fortran,
36 `MPI_UNWEIGHTED` and `MPI_WEIGHTS_EMPTY` are objects like `MPI_BOTTOM` (not usable
37 for initialization or assignment). See Section 2.5.4.

38
39 *Advice to users.* In the case of an empty `weights` array argument passed while
40 constructing a weighted graph, one should not pass `NULL` because the value of
41 `MPI_UNWEIGHTED` may be equal to `NULL`. The value of this argument would then
42 be indistinguishable from `MPI_UNWEIGHTED` to the implementation.
43 `MPI_WEIGHTS_EMPTY` should be used instead. (*End of advice to users.*)

44 *Advice to implementors.* It is recommended that `MPI_UNWEIGHTED` not be imple-
45 mented as `NULL`. (*End of advice to implementors.*)

46
47 *Rationale.* To ensure backward compatibility, `MPI_UNWEIGHTED` may still be imple-
48 mented as `NULL`. See Annex B.4. (*End of rationale.*)

The meaning of the `weights` argument can be influenced by the `info` argument. The `info` argument can be used to guide the mapping of MPI processes to the hardware; 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 MPI 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 MPI processes can construct the full topology from the distributed specification and use this in a call to `MPI_GRAPH_CREATE` to create the topology. This may serve as a reference implementation of the functionality, and may be acceptable for small communicators. However, a scalable high-quality implementation would save the topology graph in a distributed way. (*End of advice to implementors.*)

Example 8.3. Several ways to specify the adjacency matrix for `MPI_DIST_GRAPH_CREATE` and `MPI_DIST_GRAPH_CREATE_ADJACENT`. As for Example 8.2, assume there are four MPI processes with ranks 0, 1, 2, 3 in the input communicator with the following adjacency matrix and unit edge weights:

MPI 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 MPI process specifies its outgoing edges. The arguments per MPI process would be:

MPI 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 MPI process with rank 0 in the input communicator, which could be done with the following arguments per MPI process:

MPI 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:

MPI process	indegree	sources	sourceweights	outdegree	destinations	destweights
0	2	1,3	1,1	2	1,3	1,1
1	1	0	1	1	0	1
2	1	3	1	1	3	1
3	2	0,2	1,1	2	0,2	1,1

Example 8.4. Cartesian grid plus diagonals specified with `MPI_DIST_GRAPH_CREATE`. A two-dimensional $P \times Q$ torus where all MPI processes communicate along the dimensions and along the diagonal edges 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);

```

8.5.5 Topology Inquiry Functions

If a *virtual topology* has been defined with one of the above functions, then the topology information can be looked up using inquiry functions. They all are local calls.

MPI_TOPO_TEST(comm, status)

IN	comm	communicator (handle)
OUT	status	topology type of communicator comm (state)

C binding

```
int MPI_Topo_test(MPI_Comm comm, int *status)
```

Fortran 2008 binding

```
MPI_Topo_test(comm, status, ierror)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, INTENT(OUT) :: status
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_TOPO_TEST(COMM, STATUS, IERROR)
  INTEGER COMM, STATUS, IERROR
```

The function `MPI_TOPO_TEST` returns the type of topology that is associated with a communicator.

The output value `status` is one of the following:

<code>MPI_GRAPH</code>	graph topology
<code>MPI_CART</code>	Cartesian topology
<code>MPI_DIST_GRAPH</code>	distributed graph topology
<code>MPI_UNDEFINED</code>	no topology

MPI_GRAPHDIMS_GET(comm, nnodes, nedges)

IN	comm	communicator with associated graph topology (handle)
OUT	nnodes	number of nodes in graph (same as number of MPI processes in the group of comm) (integer)
OUT	nedges	number of edges in graph (integer)

C binding

```
int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)
```

Fortran 2008 binding

```
MPI_Graphdims_get(comm, nnodes, nedges, ierror)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, INTENT(OUT) :: nnodes, nedges
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```

MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR)
    INTEGER COMM, NNODES, NEDGES, IERROR

```

The functions `MPI_GRAPHDIMS_GET` and `MPI_GRAPH_GET` retrieve the graph topology information that is associated with the communicator. 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 associated graph topology (handle)
IN	maxindex	length of vector <code>index</code> in the calling program (integer)
IN	maxedges	length of vector <code>edges</code> in the calling program (integer)
OUT	index	array of integers containing the graph structure (for details see the definition of <code>MPI_GRAPH_CREATE</code>)
OUT	edges	array of integers containing the graph structure

C binding

```

int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int index[],
    int edges[])

```

Fortran 2008 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

```

Fortran 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 associated Cartesian topology (handle)
OUT	ndims	number of dimensions of the Cartesian structure (integer)

C binding

```

int MPI_Cartdim_get(MPI_Comm comm, int *ndims)

```

Fortran 2008 binding

```

MPI_Cartdim_get(comm, ndims, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm

```

```

INTEGER, INTENT(OUT) :: ndims
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran 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 is associated with the communicator. 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 associated Cartesian topology (handle)
IN	maxdims	length of vectors <code>dims</code> , <code>periods</code> , and <code>coords</code> in the calling program (integer)
OUT	dims	number of MPI processes for each Cartesian dimension (array of integers)
OUT	periods	periodicity (<code>true/false</code>) for each Cartesian dimension (array of logicals)
OUT	coords	coordinates of calling MPI process in Cartesian structure (array of integers)

C binding

```

int MPI_Cart_get(MPI_Comm comm, int maxdims, int dims[], int periods[],
                int coords[])

```

Fortran 2008 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

```

Fortran binding

```

MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
LOGICAL PERIODS(*)

```

If `maxdims` in a call to `MPI_CART_GET` is less than the number of dimensions of the Cartesian topology associated with the communicator `comm`, the outcome is unspecified.

```

1 MPI_CART_RANK(comm, coords, rank)
2   IN      comm      communicator with associated Cartesian topology
3                        (handle)
4
5   IN      coords    integer array (of size ndims) specifying the Cartesian
6                        coordinates of an MPI process
7
8   OUT     rank      rank of specified MPI process (integer)

```

C binding

```

10 int MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank)

```

Fortran 2008 binding

```

12 MPI_Cart_rank(comm, coords, rank, ierror)
13   TYPE(MPI_Comm), INTENT(IN) :: comm
14   INTEGER, INTENT(IN) :: coords(*)
15   INTEGER, INTENT(OUT) :: rank
16   INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

18 MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
19   INTEGER COMM, COORDS(*), RANK, IERROR

```

For a communicator with an associated Cartesian topology, the function `MPI_CART_RANK` translates the logical coordinates of an MPI process to the corresponding rank in the group of the communicator. 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 $0 ≤ \text{coords}(i) < \text{dims}(i)$ automatically. Out-of-range coordinates are erroneous for nonperiodic dimensions.

If `comm` is associated with a zero-dimensional Cartesian topology, `coords` is not significant and 0 is returned in `rank`.

```

31 MPI_CART_COORDS(comm, rank, maxdims, coords)

```

```

32   IN      comm      communicator with associated Cartesian topology
33                        (handle)
34
35   IN      rank      rank of an MPI process within group of comm
36                        (integer)
37
38   IN      maxdims   length of vector coords in the calling program
39                        (integer)
40
41   OUT     coords    coordinates of the MPI process with the rank rank in
42                        Cartesian structure (array of integers)

```

C binding

```

43 int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])

```

Fortran 2008 binding

```

44 MPI_Cart_coords(comm, rank, maxdims, coords, ierror)
45   TYPE(MPI_Comm), INTENT(IN) :: comm

```



```

INTEGER, INTENT(IN) :: rank, maxdims
INTEGER, INTENT(OUT) :: coords(maxdims)
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran 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. If `maxdims` is less than the number of dimensions of the Cartesian topology associated with the communicator `comm`, the outcome is unspecified.

```

MPI_GRAPH_NEIGHBORS_COUNT(comm, rank, nneighbors)

```

IN	comm	communicator with associated graph topology (handle)
IN	rank	rank of MPI process in group of <code>comm</code> (integer)
OUT	nneighbors	number of neighbors of specified MPI process (integer)

C binding

```

int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)

```

Fortran 2008 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

```

Fortran 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 associated graph topology (handle)
IN	rank	rank of MPI process in group of <code>comm</code> (integer)
IN	maxneighbors	size of array <code>neighbors</code> (integer)
OUT	neighbors	ranks of MPI processes that are neighbors to specified MPI process (array of integers)

C binding

```

int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,
  int neighbors[])

```

Fortran 2008 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

```

Fortran 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 8.5. Inquiry of graph topology information.

Assume there are four MPI processes with ranks 0, 1, 2, 3 in the input communicator with the following adjacency matrix (note that some neighbors are listed multiple times):

MPI 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 four MPI processes will return:

Input rank	Count	Neighbors
0	3	1, 1, 3
1	2	0, 0
2	1	3
3	3	0, 2, 2

Example 8.6. Using a communicator with an associated graph topology that represents a shuffle-exchange network.

Suppose that `comm` is a communicator with a shuffle-exchange topology. The group has 2^n members. Each MPI process is labeled by a_1, \dots, a_n with $a_i \in \{0, 1\}$, and has three neighbors: $\text{exchange}(a_1, \dots, a_n) = a_1, \dots, a_{n-1}, \bar{a}_n$ ($\bar{a} = 1 - a$), $\text{shuffle}(a_1, \dots, a_n) = a_2, \dots, a_n, a_1$, and $\text{unshuffle}(a_1, \dots, a_n) = a_n, a_1, \dots, a_{n-1}$. 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 MPI 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	<code>comm</code>	communicator with associated distributed graph topology (handle)
OUT	<code>indegree</code>	number of edges into this MPI process (non-negative integer)
OUT	<code>outdegree</code>	number of edges out of this MPI process (non-negative integer)
OUT	<code>weighted</code>	false if <code>MPI_UNWEIGHTED</code> was supplied during

```

1                               creation, true otherwise (logical)
2
3 C binding
4 int MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree,
5     int *outdegree, int *weighted)
6
7 Fortran 2008 binding
8 MPI_Dist_graph_neighbors_count(comm, indegree, outdegree, weighted, ierror)
9     TYPE(MPI_Comm), INTENT(IN) :: comm
10    INTEGER, INTENT(OUT) :: indegree, outdegree
11    LOGICAL, INTENT(OUT) :: weighted
12    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14 Fortran binding
15 MPI_DIST_GRAPH_NEIGHBORS_COUNT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR)
16    INTEGER COMM, INDEGREE, OUTDEGREE, IERROR
17    LOGICAL WEIGHTED
18
19
20 MPI_DIST_GRAPH_NEIGHBORS(comm, maxindegree, sources, sourceweights,
21     maxoutdegree, destinations, destweights)
22
23 IN      comm      communicator with associated distributed graph
24                    topology (handle)
25
26 IN      maxindegree  size of sources and sourceweights arrays
27                    (non-negative integer)
28
29 OUT     sources      processes for which the calling MPI process is a
30                    destination (array of non-negative integers)
31
32 OUT     sourceweights weights of the edges into the calling MPI process
33                    (array of non-negative integers)
34
35 IN      maxoutdegree size of destinations and destweights arrays
36                    (non-negative integer)
37
38 OUT     destinations MPI processes for which the calling MPI process is a
39                    source (array of non-negative integers)
40
41 OUT     destweights  weights of the edges out of the calling MPI process
42                    (array of non-negative integers)
43
44 C binding
45 int MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[],
46     int sourceweights[], int maxoutdegree, int destinations[],
47     int destweights[])
48
49 Fortran 2008 binding
50 MPI_Dist_graph_neighbors(comm, maxindegree, sources, sourceweights,
51     maxoutdegree, destinations, destweights, ierror)
52     TYPE(MPI_Comm), INTENT(IN) :: comm
53     INTEGER, INTENT(IN) :: maxindegree, maxoutdegree

```

```

INTEGER, INTENT(OUT) :: sources(maxindegree), destinations(maxoutdegree)
INTEGER :: sourceweights(*), destweights(*)
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

MPI_DIST_GRAPH_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, SOURCEWEIGHTS,
    MAXOUTDEGREE, DESTINATIONS, DESTWEIGHTS, IERROR)
INTEGER COMM, MAXINDEGREE, SOURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE,
    DESTINATIONS(*), DESTWEIGHTS(*), IERROR

```

These calls are local. The number of edges into and out of the MPI 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 MPI processes other than the calling MPI 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 MPI process in `comm`, the order of the values in `sources` and `destinations` is identical to the input that was used by the MPI 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 MPI 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 MPI process from the distributed graph specification. (*End of advice to implementors.*)

8.5.6 Cartesian Shift Coordinates

If the MPI 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 MPI process for the receive, and the rank of a destination MPI process for the send. If the function `MPI_CART_SHIFT` is called for a communicator with an associated Cartesian topology, it provides the calling MPI 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, but not zero). The function is local.

```

MPI_CART_SHIFT(comm, direction, disp, rank_source, rank_dest)

```

IN	comm	communicator with associated Cartesian topology (handle)
IN	direction	coordinate dimension of shift (integer)

1	IN	disp	displacement (> 0: upwards shift, < 0: downwards shift) (integer)
2			
3	OUT	rank_source	rank of source MPI process (integer)
4			
5	OUT	rank_dest	rank of destination MPI process (integer)
6			

C binding

```

8 int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source,
9                   int *rank_dest)

```

Fortran 2008 binding

```

11 MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror)
12   TYPE(MPI_Comm), INTENT(IN) :: comm
13   INTEGER, INTENT(IN) :: direction, disp
14   INTEGER, INTENT(OUT) :: rank_source, rank_dest
15   INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

17 MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)
18   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 topology 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 is returned in rank_source or rank_dest, indicating that the source or the destination for the shift is out of range.

It is erroneous to call MPI_CART_SHIFT with a direction that is either negative or greater than or equal to the number of dimensions in the Cartesian communicator. This implies that it is erroneous to call MPI_CART_SHIFT with a comm that is associated with a zero-dimensional Cartesian topology.

Example 8.7. Using MPI_CART_SHIFT for a Cartesian topology.

The communicator, comm, has a two-dimensional, periodic, Cartesian topology associated with it. A two-dimensional array of REALs is stored one element per MPI process, in variable A. One wishes to skew this array, by shifting column i (vertically, i.e., along the column) by i steps.

```

37 ...
38 ! find MPI process rank
39 CALL MPI_COMM_RANK(comm, rank, ierr)
40 ! find Cartesian coordinates
41 CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
42 ! compute shift source and destination
43 CALL MPI_CART_SHIFT(comm, 0, coords(2), source, dest, ierr)
44 ! skew array
45 CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0, source, 0, comm, &
46                          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.*)

8.5.7 Partitioning of Cartesian Structures

MPI_CART_SUB(comm, remain_dims, newcomm)

IN	comm	communicator with associated Cartesian topology (handle)
IN	remain_dims	the <i>i</i> -th entry of <code>remain_dims</code> specifies whether the <i>i</i> -th dimension is kept in the subgrid (<code>true</code>) or is dropped (<code>false</code>) (array of logicals)
OUT	newcomm	new communicator with associated Cartesian topology containing the subgrid that includes the calling MPI process (handle)

C binding

```
int MPI_Cart_sub(MPI_Comm comm, const int remain_dims[], MPI_Comm *newcomm)
```

Fortran 2008 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
```

Fortran binding

```
MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)
  INTEGER COMM, NEWCOMM, IERROR
  LOGICAL REMAIN_DIMS(*)
```

MPI_CART_SUB can be used to partition the group associated with a communicator that has an associated Cartesian topology into subgroups that form lower-dimensional Cartesian subgrids, and to create for each subgroup a communicator with the associated subgrid Cartesian topology. The topologies of the new communicators describe the subgrids. The number of dimensions of the subgrids is the number of remaining dimensions, i.e., the number of `true` values in `remain_dims`. The numbers of MPI processes in each coordinate direction of the subgrids are the remaining numbers of MPI processes in each coordinate direction of the grid associated with the original communicator, i.e., the values of the original grid dimensions for which the corresponding entry in `remain_dims` is `true`. The periodicity for the remaining dimensions in the new communicator is preserved from the original communicator. 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 8.8. Creation of nonoverlapping Cartesian subcommunicators with `MPI_CART_SUB`.

1 Assume that `MPI_Cart_create(..., comm)` has defined a $(2 \times 3 \times 4)$ grid. Let `remain_dims =`
 2 `(true, false, true)`. Then a call to

```
3
4 MPI_Cart_sub(comm, remain_dims, &newcomm);
```

5
 6 will create three communicators each with eight MPI processes in a 2×4 Cartesian topology.
 7 If `remain_dims = (false, false, true)` then the call to

```
8
9 MPI_Cart_sub(comm, remain_dims, &newcomm);
```

10
 11 will create six nonoverlapping communicators, each with four MPI processes, in a one-
 12 dimensional Cartesian topology.

13 8.5.8 Low-Level Topology Functions

14
 15 The two additional functions introduced in this section can be used to implement all other
 16 topology functions. In general they will not be called by the user directly, except when
 17 creating additional *virtual topology* capabilities other than those provided by MPI. The two
 18 calls are both local.
 19

20
 21 `MPI_CART_MAP(comm, ndims, dims, periods, newrank)`

22	IN	<code>comm</code>	input communicator (handle)
23	IN	<code>ndims</code>	number of dimensions of Cartesian structure (integer)
24	IN	<code>dims</code>	integer array of size <code>ndims</code> specifying the number of 25 processes in each coordinate direction
26	IN	<code>periods</code>	logical array of size <code>ndims</code> specifying the periodicity 27 specification in each coordinate direction
28	OUT	<code>newrank</code>	reordered rank of the calling MPI process; 29 <code>MPI_UNDEFINED</code> if calling MPI process does not 30 belong to grid (integer)

31 **C binding**

```
32 int MPI_Cart_map(MPI_Comm comm, int ndims, const int dims[],  

33                 const int periods[], int *newrank)
```

34 **Fortran 2008 binding**

```
35 MPI_Cart_map(comm, ndims, dims, periods, newrank, ierror)  

36 TYPE(MPI_Comm), INTENT(IN) :: comm  

37 INTEGER, INTENT(IN) :: ndims, dims(ndims)  

38 LOGICAL, INTENT(IN) :: periods(ndims)  

39 INTEGER, INTENT(OUT) :: newrank  

40 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

41 **Fortran binding**

```
42 MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)  

43 INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR  

44 LOGICAL PERIODS(*)
```


MPI_CART_MAP computes an “optimal” placement for the calling MPI process on the physical machine. A possible implementation of this function is to always return the rank of the calling MPI 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 MPI_GRAPH_CREATE
IN	edges	integer array specifying the graph structure
OUT	newrank	reordered rank of the calling MPI process; MPI_UNDEFINED if the calling MPI process does not belong to graph (integer)

C binding

```
int MPI_Graph_map(MPI_Comm comm, int nnodes, const int index[],
                 const int edges[], int *newrank)
```

Fortran 2008 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
```

Fortran 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

1 MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank), then calling
 2 MPI_COMM_SPLIT(comm, color, key, comm_graph), with color = 0 if newrank \neq
 3 MPI_UNDEFINED, color = MPI_UNDEFINED otherwise, and key = newrank.

4 All other graph topology functions can be implemented locally, using the topology
 5 information that is cached with the communicator. (*End of advice to implementors.*)
 6

8.6 Neighborhood Collective Communication on Virtual Topologies

9
 10 *Virtual topologies* specify a communication graph, but they implement no communication
 11 function themselves. Many applications require sparse nearest neighbor communications
 12 that can be expressed as graph topologies. We now describe several collective operations
 13 that perform communication along the edges of a graph representing a *virtual topology*.
 14 All of these functions are collective; i.e., they must be called by all MPI processes in the
 15 specified communicator. See Section 6 for an overview of other dense (global) collective
 16 communication operations and the semantics of collective operations.

17 If the graph was created with MPI_DIST_GRAPH_CREATE_ADJACENT with sources
 18 and destinations containing 0, . . . , n-1, where n is the number of MPI processes in the group
 19 of comm_old (i.e., the graph is fully connected and also includes an edge from each node
 20 to itself), then the sparse neighborhood communication routine performs the same data
 21 exchange as the corresponding dense (fully-connected) collective operation. In the case of a
 22 Cartesian communicator, only nearest neighbor communication is provided, corresponding
 23 to rank_source and rank_dest in MPI_CART_SHIFT with input disp = 1.

24 *Rationale.* Neighborhood collective communications enable communication on a
 25 *virtual topology*. This high-level specification of data exchange among neighboring
 26 MPI processes enables optimizations in the MPI library because the communication
 27 pattern is known statically (the topology). Thus, the implementation can compute
 28 optimized message schedules during creation of the topology [39]. This functionality
 29 can significantly simplify the implementation of neighbor exchanges [35]. (*End of*
 30 *rationale.*)
 31

32 For a distributed graph topology, created with MPI_DIST_GRAPH_CREATE, the se-
 33 quence of neighbors in the send and receive buffers at each MPI process is defined as
 34 the sequence returned by MPI_DIST_GRAPH_NEIGHBORS for destinations and sources,
 35 respectively. For a general graph topology, created with MPI_GRAPH_CREATE, the use
 36 of neighborhood collective communication is restricted to adjacency matrices, where the
 37 number of edges between any two MPI processes is defined to be the same for both MPI
 38 processes (i.e., with a symmetric adjacency matrix). In this case, the order of neighbors
 39 in the send and receive buffers is defined as the sequence of neighbors as returned by
 40 MPI_GRAPH_NEIGHBORS. Note that graph topologies should generally be replaced by the
 41 distributed graph topologies.

42 For a Cartesian topology, created with MPI_CART_CREATE, the sequence of neighbors
 43 in the send and receive buffers at each MPI process is defined by the order of the dimen-
 44 sions, first the neighbor in the negative direction and then in the positive direction with
 45 displacement 1. The numbers of sources and destinations in the communication routines
 46 are 2*ndims with ndims defined in MPI_CART_CREATE. If a neighbor does not exist, i.e.,
 47 at the border of a Cartesian topology in the case of a nonperiodic virtual grid dimension
 48 (i.e., periods[. . .]=false), then this neighbor is defined to be MPI_PROC_NULL.

If a neighbor in any of the functions is `MPI_PROC_NULL`, then the neighborhood collective communication behaves like a point-to-point communication with `MPI_PROC_NULL` in this direction. That is, the buffer is still part of the sequence of neighbors but it is neither communicated nor updated.

8.6.1 Neighborhood Gather

In the neighborhood gather operation, each MPI process i gathers data items from each MPI process j if an edge (j, i) exists in the topology graph, and each MPI process i sends the same data items to all MPI processes j where an edge (i, j) exists. The send buffer is sent to each neighboring MPI 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	<code>sendbuf</code>	starting address of send buffer (choice)
IN	<code>sendcount</code>	number of elements sent to each neighbor (non-negative integer)
IN	<code>sendtype</code>	datatype of send buffer elements (handle)
OUT	<code>recvbuf</code>	starting address of receive buffer (choice)
IN	<code>recvcount</code>	number of elements received from each neighbor (non-negative integer)
IN	<code>recvtype</code>	datatype of receive buffer elements (handle)
IN	<code>comm</code>	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)

int MPI_Neighbor_allgather_c(const void *sendbuf, MPI_Count sendcount,
    MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
    MPI_Datatype recvtype, MPI_Comm comm)
```

Fortran 2008 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

MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
    recvtype, comm, ierror) !(_c)
```

```

1  TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
2  INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
3  TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
4  TYPE(*), DIMENSION(..) :: recvbuf
5  TYPE(MPI_Comm), INTENT(IN) :: comm
6  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

8  MPI_NEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
9  RECVTYPE, COMM, IERROR)
10  <type> SENDBUF(*), RECVBUF(*)
11  INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR

```

The `MPI_NEIGHBOR_ALLGATHER` procedure supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 8.6. If `comm` is a distributed graph communicator, the outcome is as if each MPI process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

```

13  MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
14  int *srcs=(int*)malloc(indegree*sizeof(int));
15  int *dsts=(int*)malloc(outdegree*sizeof(int));
16  MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
17  outdegree, dsts, MPI_UNWEIGHTED);
18  int k;
19  /* assume sendbuf and recvbuf are of type (char*) */
20  for(k=0; k<outdegree; ++k)
21  MPI_Isend(sendbuf, sendcount, sendtype, dsts[k],...);
22  for(k=0; k<indegree; ++k)
23  MPI_Irecv(recvbuf+k*recvcount*extent(recvtype), recvcount, recvtype,
24  srcs[k],...);
25  MPI_Waitall(...);

```

Figure 8.1 shows the neighborhood gather communication of one MPI process with outgoing neighbors $d_0 \dots d_3$ and incoming neighbors $s_0 \dots s_5$. The MPI 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.

All arguments are significant on all MPI processes and the argument `comm` must have identical values on all MPI processes.

The type signature associated with `sendcount`, `sendtype`, at an MPI process must be equal to the type signature associated with `recvcount`, `recvtype` at all other MPI processes. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating MPI 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.

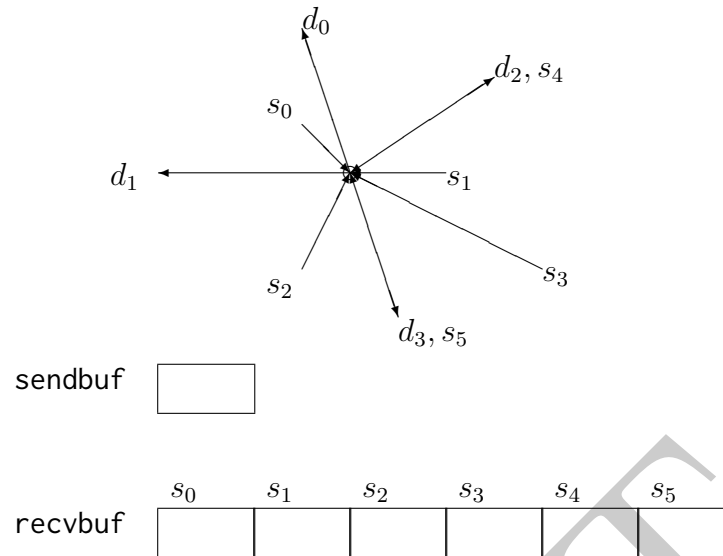


Figure 8.1: Neighborhood gather communication example

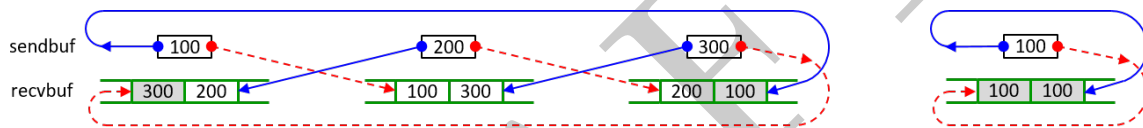


Figure 8.2: Cartesian neighborhood allgather example for 3 and 1 processes in a dimension

Example 8.9. Buffer usage of `MPI_NEIGHBOR_ALLGATHER` in the case of a Cartesian virtual topology.

On a Cartesian virtual topology, the buffer usage in a given direction d with $\text{dims}[d]=3$ and 1, respectively during creation of the communicator is described in Figure 8.2.

The figure may apply to any (or multiple) directions in the Cartesian topology. The grey buffers are required in all cases but are only accessed if during creation of the communicator, `periods[d]` was defined as nonzero (in C) or `.TRUE.` (in Fortran).

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, rcvtype, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements sent to each neighbor (non-negative integer)
IN	sendtype	datatype of send buffer elements (handle)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcounts	nonnegative integer array (of length indegree) containing the number of elements that are received from each neighbor

```

1      IN      displ      integer array (of length indegree). Entry i specifies
2                          the displacement (relative to recvbuf) at which to
3                          place the incoming data from neighbor i
4
5      IN      recvtype    datatype of receive buffer elements (handle)
6
7      IN      comm        communicator with topology structure (handle)

```

C binding

```

9  int MPI_Neighbor_allgatherv(const void *sendbuf, int sendcount,
10                             MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
11                             const int displs[], MPI_Datatype recvtype, MPI_Comm comm)

```

```

12 int MPI_Neighbor_allgatherv_c(const void *sendbuf, MPI_Count sendcount,
13                               MPI_Datatype sendtype, void *recvbuf,
14                               const MPI_Count recvcounts[], const MPI_Aint displs[],
15                               MPI_Datatype recvtype, MPI_Comm comm)

```

Fortran 2008 binding

```

18 MPI_Neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
19                          displs, recvtype, comm, ierror)

```

```

20  TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
21  INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
22  TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
23  TYPE(*), DIMENSION(..) :: recvbuf
24  TYPE(MPI_Comm), INTENT(IN) :: comm
25  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

27 MPI_Neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
28                          displs, recvtype, comm, ierror) !(_c)

```

```

29  TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
30  INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcounts(*)
31  TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
32  TYPE(*), DIMENSION(..) :: recvbuf
33  INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
34  TYPE(MPI_Comm), INTENT(IN) :: comm
35  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

37 MPI_NEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
38                          DISPLS, RECVTYPE, COMM, IERROR)

```

```

39  <type> SENDBUF(*), RECVBUF(*)
40  INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
41  IERROR

```

The `MPI_NEIGHBOR_ALLGATHERV` procedure supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 8.6. If `comm` is a distributed graph communicator, the outcome is as if each MPI process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

```

47 MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
48 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;

/* assume sendbuf and recvbuf are of type (char*) */
for(k=0; k<outdegree; ++k)
    MPI_Isend(sendbuf, sendcount, sendtype, dsts[k],...);

for(k=0; k<indegree; ++k)
    MPI_Irecv(recvbuf+displs[k]*extent(recvtype), recvcnts[k], recvtype,
              srcs[k],...);

MPI_Waitall(...);

```

The type signature associated with `sendcount`, `sendtype`, at MPI process j must be equal to the type signature associated with `recvcnts[l]`, `recvtype` at any other MPI 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 MPI 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 MPI processes and the argument `comm` must have identical values on all MPI processes.

8.6.2 Neighborhood Alltoall

In the neighborhood alltoall operation, each MPI process i receives data items from each MPI process j if an edge (j, i) exists in the topology graph or Cartesian topology. Similarly, each MPI process i sends data items to all MPI 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 MPI process and the l -th block in the receive buffer is received from the l -th neighbor.

```

MPI_NEIGHBOR_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcnt, recvtype,
                       comm)

```

IN	<code>sendbuf</code>	starting address of send buffer (choice)
IN	<code>sendcount</code>	number of elements sent to each neighbor (non-negative integer)
IN	<code>sendtype</code>	datatype of send buffer elements (handle)
OUT	<code>recvbuf</code>	starting address of receive buffer (choice)
IN	<code>recvcnt</code>	number of elements received from each neighbor (non-negative integer)
IN	<code>recvtype</code>	datatype of receive buffer elements (handle)
IN	<code>comm</code>	communicator with topology structure (handle)

C binding

```

1 int MPI_Neighbor_alltoall(const void *sendbuf, int sendcount,
2     MPI_Datatype sendtype, void *recvbuf, int recvcount,
3     MPI_Datatype recvtype, MPI_Comm comm)
4
5 int MPI_Neighbor_alltoall_c(const void *sendbuf, MPI_Count sendcount,
6     MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
7     MPI_Datatype recvtype, MPI_Comm comm)
8

```

Fortran 2008 binding

```

9 MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
10     recvtype, comm, ierror)
11     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
12     INTEGER, INTENT(IN) :: sendcount, recvcount
13     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
14     TYPE(*), DIMENSION(..) :: recvbuf
15     TYPE(MPI_Comm), INTENT(IN) :: comm
16     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18 MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
19     recvtype, comm, ierror) !(_c)
20     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
21     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
22     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
23     TYPE(*), DIMENSION(..) :: recvbuf
24     TYPE(MPI_Comm), INTENT(IN) :: comm
25     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26

```

Fortran binding

```

27 MPI_NEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
28     RECVTYPE, COMM, IERROR)
29     <type> SENDBUF(*), RECVBUF(*)
30     INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
31

```

The `MPI_NEIGHBOR_ALLTOALL` procedure supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 8.6. If `comm` is a distributed graph communicator, the outcome is as if each MPI process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

```

37 MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
38 int *srcs=(int*)malloc(indegree*sizeof(int));
39 int *dsts=(int*)malloc(outdegree*sizeof(int));
40 MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
41     outdegree, dsts, MPI_UNWEIGHTED);
42 int k;
43 /* assume sendbuf and recvbuf are of type (char*) */
44 for(k=0; k<outdegree; ++k)
45     MPI_Isend(sendbuf+k*sendcount*extent(sendtype), sendcount, sendtype,
46     dsts[k],...);
47
48 for(k=0; k<indegree; ++k)

```



```
MPI_Irecv(recvbuf+k*recvcount*extent(recvtype), recvcount, recvtype,
          srcs[k],...);
```

```
MPI_Waitall(...);
```

The type signature associated with `sendcount`, `sendtype`, at an MPI process must be equal to the type signature associated with `recvcount`, `recvtype` at any other MPI process. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating MPI 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 MPI processes and the argument `comm` must have identical values on all MPI processes.

Example 8.10. Buffer usage of `MPI_NEIGHBOR_ALLTOALL` in the case of a Cartesian virtual topology.

For a halo communication on a Cartesian grid, the buffer usage in a given direction `d` with `dims[d]=3` and 1, respectively during creation of the communicator is described in Figure 8.3. The figure may apply to any (or multiple) directions in the Cartesian topology. The grey buffers are required in all cases but are only accessed if during creation of the communicator, `periods[d]` was defined as nonzero (in C) or `.TRUE.` (in Fortran).

If `sendbuf` and `recvbuf` are declared as `(char *)` and contain a sequence of buffers each described by `sendcount,sendtype` and `recvbuf,recvtype`, then after `MPI_NEIGHBOR_ALLTOALL` on a Cartesian communicator returned, the content of the `recvbuf` is as if the following code is executed:

```
MPI_Cartdim_get(comm, &ndims);
MPI_Type_get_extent(sendtype, &send_lb, &send_extent);
MPI_Type_get_extent(recvtype, &recv_lb, &recv_extent);
for( /*direction*/ d=0; d < ndims; d++) {
    MPI_Cart_shift(comm, /*direction*/ d, /*disp*/ 1, &rank_source, &rank_dest);
    MPI_Sendrecv(sendbuf+(d*2+0)*sendcount*send_extent,
                sendcount, sendtype, rank_source, /*sendtag*/ d*2,
                recvbuf+(d*2+1)*recvcount*recv_extent,
                recvcount, recvtype, rank_dest, /*recvtag*/ d*2,
                comm,&status); /*communication in direction of displacement -1*/
    MPI_Sendrecv(sendbuf+(d*2+1)*sendcount*send_extent,
                sendcount, sendtype, rank_dest, /*sendtag*/ d*2+1,
                recvbuf+(d*2+0)*recvcount*recv_extent,
                recvcount, recvtype, rank_source, /*recvtag*/ d*2+1,
                comm,&status); /*communication in direction of displacement +1*/
}
```

The first call to `MPI_Sendrecv` implements the solid arrows’ communication pattern in each diagram of Figure 8.3, whereas the second call is for the dashed arrows’ pattern.

Advice to implementors. For a Cartesian topology, if the grid in a direction `d` is periodic and `dims[d]` is equal to 1 or 2, then `rank_source` and `rank_dest` are identical, but still all `ndims` send and `ndims` receive operations use different buffers. If in this case, the two send and receive operations per direction or of all directions are internally parallelized, then the several send and receive operations for the same sender-receiver MPI process pair shall be initiated in the same sequence on sender and receiver side or they shall be distinguished by different tags. The code above shows a valid sequence

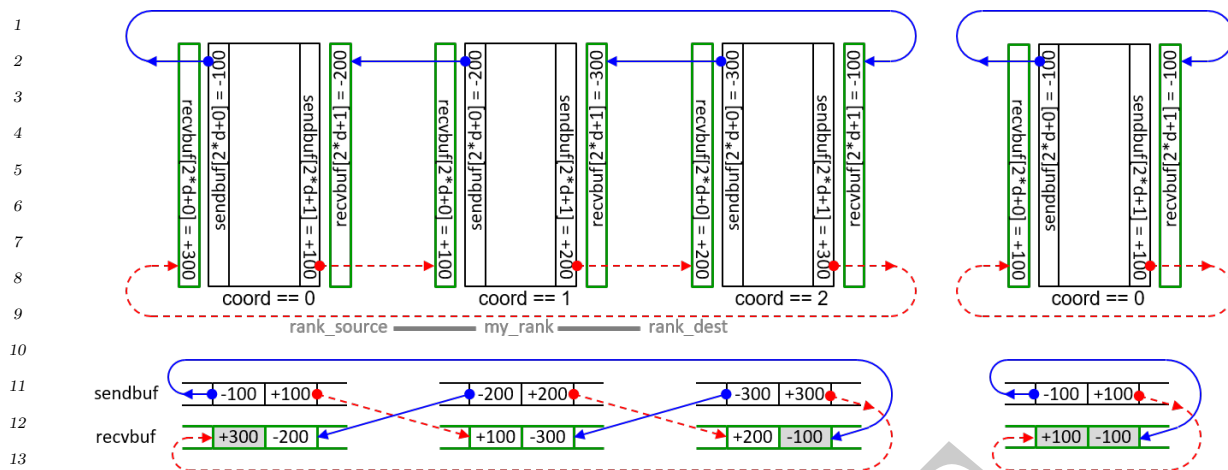


Figure 8.3: Cartesian neighborhood alltoall example for 3 and 1 MPI processes in a dimension

of operations and tags. (*End of advice to implementors.*)

The vector variant of `MPI_NEIGHBOR_ALLTOALL` allows sending/receiving different numbers of elements to and from each neighbor.

`MPI_NEIGHBOR_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, rcvbuf, rcvcounts, rdispls, rcvtype, comm)`

IN	<code>sendbuf</code>	starting address of send buffer (choice)
IN	<code>sendcounts</code>	nonnegative integer array (of length outdegree) specifying the number of elements to send to each neighbor
IN	<code>sdispls</code>	integer array (of length outdegree). Entry <code>j</code> specifies the displacement (relative to <code>sendbuf</code>) from which to send the outgoing data to neighbor <code>j</code>
IN	<code>sendtype</code>	datatype of send buffer elements (handle)
OUT	<code>rcvbuf</code>	starting address of receive buffer (choice)
IN	<code>rcvcounts</code>	nonnegative integer array (of length indegree) specifying the number of elements that are received from each neighbor
IN	<code>rdispls</code>	integer array (of length indegree). Entry <code>i</code> specifies the displacement (relative to <code>rcvbuf</code>) at which to place the incoming data from neighbor <code>i</code>
IN	<code>rcvtype</code>	datatype of receive buffer elements (handle)
IN	<code>comm</code>	communicator with topology structure (handle)

C binding

```
int MPI_Neighbor_alltoallv(const void *sendbuf, const int sendcounts[],
                          const int sdispls[], MPI_Datatype sendtype, void *rcvbuf,
```

```

        const int recvcnts[], const int rdispls[],
        MPI_Datatype recvtype, MPI_Comm comm)
int MPI_Neighbor_alltoallv_c(const void *sendbuf, const MPI_Count sendcounts[],
        const MPI_Aint sdispls[], MPI_Datatype sendtype, void *recvbuf,
        const MPI_Count recvcnts[], const MPI_Aint rdispls[],
        MPI_Datatype recvtype, MPI_Comm comm)

```

Fortran 2008 binding

```

MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
        recvcnts, rdispls, recvtype, comm, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcnts(*), 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(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
        recvcnts, rdispls, recvtype, comm, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*), recvcnts(*)
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..) :: recvbuf
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran 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

```

The `MPI_NEIGHBOR_ALLTOALLV` procedure supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 8.6. If `comm` is a distributed graph communicator, the outcome is as if each MPI 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;
/* 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],...);

```

```

1  for(k=0; k<indegree; ++k)
2      MPI_Irecv(recvbuf+rdispls[k]*extent(recvtype), recvcnts[k],
3              recvtype, srcs[k],...);
4
5  MPI_Waitall(...);

```

The type signature associated with `sendcounts[k]`, `sendtype` with `dsts[k]=j` at MPI process i must be equal to the type signature associated with `recvcnts[l]`, `recvtype` with `srcs[l]=i` at MPI 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 MPI 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 MPI processes and the argument `comm` must have identical values on all MPI processes.

`MPI_NEIGHBOR_ALLTOALLW` allows one to send and receive with different datatypes to and from each neighbor.

```

20 MPI_NEIGHBOR_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcnts,
21                        rdispls, recvtypes, comm)

```

23	IN	<code>sendbuf</code>	starting address of send buffer (choice)
24	IN	<code>sendcounts</code>	nonnegative integer array (of length <code>outdegree</code>) specifying the number of elements to send to each neighbor
25			
26			
27	IN	<code>sdispls</code>	integer array (of length <code>outdegree</code>). Entry j specifies the displacement in bytes (relative to <code>sendbuf</code>) from which to take the outgoing data destined for neighbor j (array of integers)
28			
29			
30			
31			
32	IN	<code>sendtypes</code>	array of datatypes (of length <code>outdegree</code>). Entry j specifies the type of data to send to neighbor j (array of handles)
33			
34			
35	OUT	<code>recvbuf</code>	starting address of receive buffer (choice)
36	IN	<code>recvcnts</code>	nonnegative integer array (of length <code>indegree</code>) specifying the number of elements that are received from each neighbor
37			
38			
39			
40	IN	<code>rdispls</code>	integer array (of length <code>indegree</code>). Entry i specifies the displacement in bytes (relative to <code>recvbuf</code>) at which to place the incoming data from neighbor i (array of integers)
41			
42			
43			
44	IN	<code>recvtypes</code>	array of datatypes (of length <code>indegree</code>). Entry i specifies the type of data received from neighbor i (array of handles)
45			
46			
47			
48	IN	<code>comm</code>	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)
int MPI_Neighbor_alltoallw_c(const void *sendbuf, const MPI_Count sendcounts[],
                              const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],
                              void *recvbuf, const MPI_Count recvcounts[],
                              const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],
                              MPI_Comm comm)

```

Fortran 2008 binding

```

MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                       recvcounts, rdispls, recvtypes, comm, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
INTEGER, INTENT(IN) :: sendcounts(*), recvcounts(*)
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
TYPE(*), DIMENSION(..) :: recvbuf
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                       recvcounts, rdispls, recvtypes, comm, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*), recvcounts(*)
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
TYPE(*), DIMENSION(..) :: recvbuf
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

MPI_NEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
                       RECVCOUNTS, RDISPLS, RECVTYPES, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)

```

The `MPI_NEIGHBOR_ALLTOALLW` procedure supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 8.6. If `comm` is a distributed graph communicator, the outcome is as if each MPI 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);

```

```

1  int k;
2
3  /* assume sendbuf and recvbuf are of type (char*) */
4  for(k=0; k<outdegree; ++k)
5      MPI_Isend(sendbuf+sdispls[k], sendcounts[k], sendtypes[k],
6              dsts[k],...);
7
8  for(k=0; k<indegree; ++k)
9      MPI_Irecv(recvbuf+rdispls[k], recvcounst[k], recvtypes[k],
10             srcs[k],...);
11 MPI_Waitall(...);

```

The type signature associated with `sendcounts[k]`, `sendtypes[k]` with `dsts[k]=j` at MPI process i must be equal to the type signature associated with `recvcounst[l]`, `recvtypes[l]` with `srcs[l]=i` at MPI 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 MPI 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 MPI processes and the argument `comm` must have identical values on all MPI processes.

8.7 Nonblocking Neighborhood Communication on Process Topologies

Nonblocking variants of the neighborhood collective operations allow relaxed synchronization and overlapping of computation and communication. The semantics are similar to nonblocking collective operations as described in Section 6.12.

8.7.1 Nonblocking Neighborhood Gather

`MPI_INEIGHBOR_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcounst, recvtype, comm, request)`

IN	<code>sendbuf</code>	starting address of send buffer (choice)
IN	<code>sendcount</code>	number of elements sent to each neighbor (non-negative integer)
IN	<code>sendtype</code>	datatype of send buffer elements (handle)
OUT	<code>recvbuf</code>	starting address of receive buffer (choice)
IN	<code>recvcounst</code>	number of elements received from each neighbor (non-negative integer)
IN	<code>recvtype</code>	datatype of receive buffer elements (handle)
IN	<code>comm</code>	communicator with topology structure (handle)
OUT	<code>request</code>	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)
int MPI_Ineighbor_allgather_c(const void *sendbuf, MPI_Count sendcount,
        MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
        MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)

```

Fortran 2008 binding

```

MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
        recvtype, comm, request, ierror)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
INTEGER, INTENT(IN) :: sendcount, recvcount
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
        recvtype, comm, request, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

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_ALLGATHER starts a nonblocking variant of
MPI_NEIGHBOR_ALLGATHER.

```

```

MPI_INEIGHBOR_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcoun, displ,
        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	datatype of send buffer elements (handle)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcoun	nonnegative integer array (of length indegree) containing the number of elements that are received from each neighbor

1	IN	displs	integer array (of length indegree). Entry <i>i</i> specifies the displacement (relative to <code>recvbuf</code>) at which to place the incoming data from neighbor <i>i</i>
2			
3			
4	IN	recvtype	datatype of receive buffer elements (handle)
5			
6	IN	comm	communicator with topology structure (handle)
7	OUT	request	communication request (handle)
8			

C binding

```

10 int MPI_Ineighbor_allgatherv(const void *sendbuf, int sendcount,
11     MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
12     const int displs[], MPI_Datatype recvtype, MPI_Comm comm,
13     MPI_Request *request)
14 
```

```

15 int MPI_Ineighbor_allgatherv_c(const void *sendbuf, MPI_Count sendcount,
16     MPI_Datatype sendtype, void *recvbuf,
17     const MPI_Count recvcounts[], const MPI_Aint displs[],
18     MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
19 
```

Fortran 2008 binding

```

20 MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
21     displs, recvtype, comm, request, ierror)
22     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
23     INTEGER, INTENT(IN) :: sendcount
24     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
25     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
26     INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
27     TYPE(MPI_Comm), INTENT(IN) :: comm
28     TYPE(MPI_Request), INTENT(OUT) :: request
29     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30 
```

```

31 MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
32     displs, recvtype, comm, request, ierror) !(_c)
33     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
34     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
35     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
36     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
37     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
38     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
39     TYPE(MPI_Comm), INTENT(IN) :: comm
40     TYPE(MPI_Request), INTENT(OUT) :: request
41     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42 
```

Fortran binding

```

43 MPI_INEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
44     DISPLS, RECVTYPE, COMM, REQUEST, IERROR)
45     <type> SENDBUF(*), RECVBUF(*)
46     INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
47     REQUEST, IERROR
48 
```


MPI_INEIGHBOR_ALLGATHERV starts a nonblocking variant of MPI_NEIGHBOR_ALLGATHERV.

8.7.2 Nonblocking Neighborhood Alltoall

MPI_INEIGHBOR_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)

IN	sendbuf	starting address of send buffer (choice)	10
IN	sendcount	number of elements sent to each neighbor (non-negative integer)	11
IN	sendtype	datatype of send buffer elements (handle)	12
OUT	recvbuf	starting address of receive buffer (choice)	13
IN	recvcount	number of elements received from each neighbor (non-negative integer)	14
IN	recvtype	datatype of receive buffer elements (handle)	15
IN	comm	communicator with topology structure (handle)	16
OUT	request	communication request (handle)	17

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)
```

```
int MPI_Ineighbor_alltoall_c(const void *sendbuf, MPI_Count sendcount,
    MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
    MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
```

Fortran 2008 binding

```
MPI_Ineighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
    recvtype, comm, request, ierror)
```

```
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
```

```
INTEGER, INTENT(IN) :: sendcount, recvcount
```

```
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
```

```
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
```

```
TYPE(MPI_Comm), INTENT(IN) :: comm
```

```
TYPE(MPI_Request), INTENT(OUT) :: request
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_Ineighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
    recvtype, comm, request, ierror) !(_c)
```

```
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
```

```
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
```

```
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
```

```
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
```

```
TYPE(MPI_Comm), INTENT(IN) :: comm
```

```

1     TYPE(MPI_Request), INTENT(OUT) :: request
2     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

4 MPI_INEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
5     RECVTYPE, COMM, REQUEST, IERROR)
6     <type> SENDBUF(*), RECVBUF(*)
7     INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
8

```

```

9     MPI_INEIGHBOR_ALLTOALL starts a nonblocking variant of
10    MPI_NEIGHBOR_ALLTOALL.

```

```

12 MPI_INEIGHBOR_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcoun
13    ts, rdispls, recvtype, comm, request)

```

15	IN	sendbuf	starting address of send buffer (choice)
16	IN	sendcounts	nonnegative integer array (of length outdegree)
17			specifying the number of elements to send to each
18			neighbor
19	IN	sdispls	integer array (of length outdegree). Entry j specifies
20			the displacement (relative to sendbuf) from which
21			send the outgoing data to neighbor j
22	IN	sendtype	datatype of send buffer elements (handle)
23	OUT	recvbuf	starting address of receive buffer (choice)
24	IN	recvcoun	nonnegative integer array (of length indegree)
25			specifying the number of elements that are received
26			from each neighbor
27	IN	rdispls	integer array (of length indegree). Entry i specifies
28			the displacement (relative to recvbuf) at which to
29			place the incoming data from neighbor i
30	IN	recvtype	datatype of receive buffer elements (handle)
31	IN	comm	communicator with topology structure (handle)
32	OUT	request	communication request (handle)

C binding

```

37
38 int MPI_Ineighbor_alltoallv(const void *sendbuf, const int sendcounts[],
39     const int sdispls[], MPI_Datatype sendtype, void *recvbuf,
40     const int recvcoun
41     ts, const int rdispls[],
42     MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
43
44 int MPI_Ineighbor_alltoallv_c(const void *sendbuf,
45     const MPI_Count sendcounts[], const MPI_Aint sdispls[],
46     MPI_Datatype sendtype, void *recvbuf,
47     const MPI_Count recvcoun
48     ts, const MPI_Aint rdispls[],
49     MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)

```

Fortran 2008 binding

```

MPI_Inighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
    recvcounts, rdispls, recvtype, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
        recvcounts(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

MPI_Inighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
    recvcounts, rdispls, recvtype, comm, request, ierror) !(_c)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*),
        recvcounts(*)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
        rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran 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

```

MPI_INEIGHBOR_ALLTOALLV 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	nonnegative 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)

1	IN	sendtypes	array of datatypes (of length outdegree). Entry j
2			specifies the type of data to send to neighbor j (array
3			of handles)
4	OUT	recvbuf	starting address of receive buffer (choice)
5	IN	recvcounts	nonnegative integer array (of length indegree)
6			specifying the number of elements that are received
7			from each neighbor
8	IN	rdispls	integer array (of length indegree). Entry i specifies
9			the displacement in bytes (relative to recvbuf) at
10			which to place the incoming data from neighbor i
11			(array of integers)
12	IN	recvtypes	array of datatypes (of length indegree). Entry i
13			specifies the type of data received from neighbor i
14			(array of handles)
15	IN	comm	communicator with topology structure (handle)
16	OUT	request	communication request (handle)
17			
18			
19			

C binding

```

21 int MPI_Ineighbor_alltoallw(const void *sendbuf, const int sendcounts[],
22     const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],
23     void *recvbuf, const int recvcounts[], const MPI_Aint rdispls[],
24     const MPI_Datatype recvtypes[], MPI_Comm comm,
25     MPI_Request *request)
26
27 int MPI_Ineighbor_alltoallw_c(const void *sendbuf,
28     const MPI_Count sendcounts[], const MPI_Aint sdispls[],
29     const MPI_Datatype sendtypes[], void *recvbuf,
30     const MPI_Count recvcounts[], const MPI_Aint rdispls[],
31     const MPI_Datatype recvtypes[], MPI_Comm comm,
32     MPI_Request *request)

```

Fortran 2008 binding

```

34 MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
35     recvcounts, rdispls, recvtypes, comm, request, ierror)
36     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
37     INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
38     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
39     rdispls(*)
40     TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*), recvtypes(*)
41     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
42     TYPE(MPI_Comm), INTENT(IN) :: comm
43     TYPE(MPI_Request), INTENT(OUT) :: request
44     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46 MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
47     recvcounts, rdispls, recvtypes, comm, request, ierror) !(_c)
48     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf

```

```

INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*),
    recvcounts(*)
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
    rdispls(*)
TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*), recvtypes(*)
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

MPI_INEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
    RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
    REQUEST, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)

```

MPI_INEIGHBOR_ALLTOALLW starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALLW.

8.8 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 6.13.

8.8.1 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	datatype 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	datatype of receive buffer elements (handle)
IN	comm	communicator with topology structure (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)

C binding

```

1 int MPI_Neighbor_allgather_init(const void *sendbuf, int sendcount,
2     MPI_Datatype sendtype, void *recvbuf, int recvcount,
3     MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
4     MPI_Request *request)
5
6

```

```

7 int MPI_Neighbor_allgather_init_c(const void *sendbuf, MPI_Count sendcount,
8     MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
9     MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
10    MPI_Request *request)
11

```

Fortran 2008 binding

```

12 MPI_Neighbor_allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
13     recvtype, comm, info, request, ierror)
14     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
15     INTEGER, INTENT(IN) :: sendcount, recvcount
16     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
17     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
18     TYPE(MPI_Comm), INTENT(IN) :: comm
19     TYPE(MPI_Info), INTENT(IN) :: info
20     TYPE(MPI_Request), INTENT(OUT) :: request
21     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22

```

```

23 MPI_Neighbor_allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
24     recvtype, comm, info, request, ierror) !(_c)
25     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
26     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
27     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
28     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
29     TYPE(MPI_Comm), INTENT(IN) :: comm
30     TYPE(MPI_Info), INTENT(IN) :: info
31     TYPE(MPI_Request), INTENT(OUT) :: request
32     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33

```

Fortran binding

```

34 MPI_NEIGHBOR_ALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, REVCOUNT,
35     RECVTYPE, COMM, INFO, REQUEST, IERROR)
36     <type> SENDBUF(*), RECVBUF(*)
37     INTEGER SENDCOUNT, SENDTYPE, REVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
38     IERROR
39

```

40 Creates a persistent collective communication request for the neighborhood allgather
41 operation.

```

42
43 MPI_NEIGHBOR_ALLGATHERV_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
44     displ, recvtype, comm, info, request)
45

```

46	IN	sendbuf	starting address of send buffer (choice)
47	IN	sendcount	number of elements sent to each neighbor (non-negative integer)
48			

IN	sendtype	datatype of send buffer elements (handle)	1
OUT	recvbuf	starting address of receive buffer (choice)	2
IN	recvcnts	nonnegative integer array (of length indegree) containing the number of elements that are received from each neighbor	3 4 5 6
IN	displs	integer array (of length indegree). Entry <i>i</i> specifies the displacement (relative to <i>recvbuf</i>) at which to place the incoming data from neighbor <i>i</i>	7 8 9
IN	recvtype	datatype of receive buffer elements (handle)	10
IN	comm	communicator with topology structure (handle)	11
IN	info	info argument (handle)	12
OUT	request	communication request (handle)	13 14 15
C binding			16
<code>int MPI_Neighbor_allgatherv_init(const void *sendbuf, int sendcount,</code>			17
<code> MPI_Datatype sendtype, void *recvbuf, const int recvcnts[],</code>			18
<code> const int displs[], MPI_Datatype recvtype, MPI_Comm comm,</code>			19
<code> MPI_Info info, MPI_Request *request)</code>			20 21
<code>int MPI_Neighbor_allgatherv_init_c(const void *sendbuf, MPI_Count sendcount,</code>			22
<code> MPI_Datatype sendtype, void *recvbuf,</code>			23
<code> const MPI_Count recvcnts[], const MPI_Aint displs[],</code>			24
<code> MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,</code>			25
<code> MPI_Request *request)</code>			26 27
Fortran 2008 binding			28
<code>MPI_Neighbor_allgatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcnts,</code>			29
<code> displs, recvtype, comm, info, request, ierror)</code>			30
<code> TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf</code>			31
<code> INTEGER, INTENT(IN) :: sendcount, displs(*)</code>			32
<code> TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype</code>			33
<code> TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf</code>			34
<code> INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcnts(*)</code>			35
<code> TYPE(MPI_Comm), INTENT(IN) :: comm</code>			36
<code> TYPE(MPI_Info), INTENT(IN) :: info</code>			37
<code> TYPE(MPI_Request), INTENT(OUT) :: request</code>			38
<code> INTEGER, OPTIONAL, INTENT(OUT) :: ierror</code>			39
<code>MPI_Neighbor_allgatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcnts,</code>			40
<code> displs, recvtype, comm, info, request, ierror) !(_c)</code>			41
<code> TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf</code>			42
<code> INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount</code>			43
<code> TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype</code>			44
<code> TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf</code>			45
<code> INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcnts(*)</code>			46
<code> INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)</code>			47
<code> TYPE(MPI_Comm), INTENT(IN) :: comm</code>			48

```

1     TYPE(MPI_Info), INTENT(IN) :: info
2     TYPE(MPI_Request), INTENT(OUT) :: request
3     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4

```

Fortran binding

```

5 MPI_NEIGHBOR_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
6     DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)
7     <type> SENDBUF(*), RECVBUF(*)
8     INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
9     INFO, REQUEST, IERROR
10

```

11 Creates a persistent collective communication request for the neighborhood allgatherv
12 operation.

8.8.2 Persistent Neighborhood Alltoall

```

17 MPI_NEIGHBOR_ALLTOALL_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount,
18     recvtype, comm, info, request)
19
20 IN     sendbuf     starting address of send buffer (choice)
21 IN     sendcount   number of elements sent to each neighbor
22         (non-negative integer)
23 IN     sendtype    datatype of send buffer elements (handle)
24 OUT    recvbuf     starting address of receive buffer (choice)
25 IN     recvcount   number of elements received from each neighbor
26         (non-negative integer)
27 IN     recvtype    datatype of receive buffer elements (handle)
28 IN     comm        communicator with topology structure (handle)
29 IN     info        info argument (handle)
30 OUT    request     communication request (handle)
31
32
33
34

```

C binding

```

35 int MPI_Neighbor_alltoall_init(const void *sendbuf, int sendcount,
36     MPI_Datatype sendtype, void *recvbuf, int recvcount,
37     MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
38     MPI_Request *request)
39
40 int MPI_Neighbor_alltoall_init_c(const void *sendbuf, MPI_Count sendcount,
41     MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
42     MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
43     MPI_Request *request)
44

```

Fortran 2008 binding

```

45 MPI_Neighbor_alltoall_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
46     recvtype, comm, info, request, ierror)
47     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
48

```



```

INTEGER, INTENT(IN) :: sendcount, recvcount           1
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype  2
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf     3
TYPE(MPI_Comm), INTENT(IN) :: comm                  4
TYPE(MPI_Info), INTENT(IN) :: info                 5
TYPE(MPI_Request), INTENT(OUT) :: request           6
INTEGER, OPTIONAL, INTENT(OUT) :: ierror           7
                                                    8
MPI_Neighbor_alltoall_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
    recvtype, comm, info, request, ierror) !(_c)     9
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 10
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount 11
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 12
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf     13
TYPE(MPI_Comm), INTENT(IN) :: comm                  14
TYPE(MPI_Info), INTENT(IN) :: info                 15
TYPE(MPI_Request), INTENT(OUT) :: request           16
INTEGER, OPTIONAL, INTENT(OUT) :: ierror           17
                                                    18
Fortran binding                                  19
MPI_NEIGHBOR_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
    RECVTYPE, COMM, INFO, REQUEST, IERROR)          20
<type> SENDBUF(*), RECVBUF(*)                      21
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
    IERROR                                          22
                                                    23
Creates a persistent collective communication request for the neighborhood alltoall
operation.                                         24
                                                    25
MPI_NEIGHBOR_ALLTOALLV_INIT(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
    recvcounts, rdispls, recvtype, comm, info, request) 26
                                                    27
IN      sendbuf      starting address of send buffer (choice) 28
IN      sendcounts   nonnegative integer array (of length outdegree)
                    specifying the number of elements to send to each
                    neighbor 29
IN      sdispls      integer array (of length outdegree). Entry j specifies
                    the displacement (relative to sendbuf) from which
                    send the outgoing data to neighbor j 30
IN      sendtype     datatype of send buffer elements (handle) 31
OUT     recvbuf      starting address of receive buffer (choice) 32
IN      recvcounts   nonnegative integer array (of length indegree)
                    specifying the number of elements that are received
                    from each neighbor 33
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 34

```

```

1      IN      recvtype      datatype of receive buffer elements (handle)
2      IN      comm         communicator with topology structure (handle)
3
4      IN      info         info argument (handle)
5      OUT     request      communication request (handle)
6

```

C binding

```

8      int MPI_Neighbor_alltoallv_init(const void *sendbuf, const int sendcounts[],
9                                     const int sdispls[], MPI_Datatype sendtype, void *recvbuf,
10                                    const int recvcnts[], const int rdispls[],
11                                    MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
12                                    MPI_Request *request)
13

```

```

14     int MPI_Neighbor_alltoallv_init_c(const void *sendbuf,
15                                       const MPI_Count sendcounts[], const MPI_Aint sdispls[],
16                                       MPI_Datatype sendtype, void *recvbuf,
17                                       const MPI_Count recvcnts[], const MPI_Aint rdispls[],
18                                       MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
19                                       MPI_Request *request)
20

```

Fortran 2008 binding

```

21     MPI_Neighbor_alltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
22                                 recvcnts, rdispls, recvtype, comm, info, request, ierror)
23     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
24     INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
25                                 recvcnts(*), rdispls(*)
26     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
27     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
28     TYPE(MPI_Comm), INTENT(IN) :: comm
29     TYPE(MPI_Info), INTENT(IN) :: info
30     TYPE(MPI_Request), INTENT(OUT) :: request
31     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33     MPI_Neighbor_alltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
34                                 recvcnts, rdispls, recvtype, comm, info, request, ierror) !(_c)
35     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
36     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*),
37                                 recvcnts(*)
38     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
39                                 rdispls(*)
40     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
41     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
42     TYPE(MPI_Comm), INTENT(IN) :: comm
43     TYPE(MPI_Info), INTENT(IN) :: info
44     TYPE(MPI_Request), INTENT(OUT) :: request
45     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46

```

Fortran binding

```

47     MPI_NEIGHBOR_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
48

```

```

    RECVCOUNTS, RDISPLS, RECVMODE, 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,
    recvcnts, rdispls, recvtypes, comm, info, request)

```

IN	sendbuf	starting address of send buffer (choice)	12
IN	sendcounts	nonnegative integer array (of length outdegree) specifying the number of elements to send to each neighbor	13 14 15
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)	16 17 18 19 20
IN	sendtypes	array of datatypes (of length outdegree). Entry j specifies the type of data to send to neighbor j (array of handles)	21 22 23
OUT	recvbuf	starting address of receive buffer (choice)	24
IN	recvcnts	nonnegative integer array (of length indegree) specifying the number of elements that are received from each neighbor	25 26 27 28
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)	29 30 31 32
IN	recvtypes	array of datatypes (of length indegree). Entry i specifies the type of data received from neighbor i (array of handles)	33 34 35
IN	comm	communicator with topology structure (handle)	36
IN	info	info argument (handle)	37 38
OUT	request	communication request (handle)	39 40

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 recvcnts[], const MPI_Aint rdispls[],
    const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info,
    MPI_Request *request)

```

```

1 int MPI_Neighbor_alltoallw_init_c(const void *sendbuf,
2     const MPI_Count sendcounts[], const MPI_Aint sdispls[],
3     const MPI_Datatype sendtypes[], void *recvbuf,
4     const MPI_Count recvcnts[], const MPI_Aint rdispls[],
5     const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info,
6     MPI_Request *request)

```

Fortran 2008 binding

```

9 MPI_Neighbor_alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
10     recvcnts, rdispls, recvtypes, comm, info, request, ierror)
11     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
12     INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcnts(*)
13     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
14     rdispls(*)
15     TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*), recvtypes(*)
16     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
17     TYPE(MPI_Comm), INTENT(IN) :: comm
18     TYPE(MPI_Info), INTENT(IN) :: info
19     TYPE(MPI_Request), INTENT(OUT) :: request
20     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

21 MPI_Neighbor_alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
22     recvcnts, rdispls, recvtypes, comm, info, request, ierror)
23     !(_c)
24     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
25     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*),
26     recvcnts(*)
27     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
28     rdispls(*)
29     TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*), recvtypes(*)
30     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
31     TYPE(MPI_Comm), INTENT(IN) :: comm
32     TYPE(MPI_Info), INTENT(IN) :: info
33     TYPE(MPI_Request), INTENT(OUT) :: request
34     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

36 MPI_NEIGHBOR_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
37     RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST, IERROR)
38     <type> SENDBUF(*), RECVBUF(*)
39     INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
40     INFO, REQUEST, IERROR
41     INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)

```

Creates a persistent collective communication request for the neighborhood alltoallw operation.

```

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 MPI 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 MPI 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 8.4: Set-up of MPI process structure for two-dimensional parallel Poisson solver

8.9 An Application Example

Example 8.11. Neighborhood collective communication in a Cartesian virtual topology. The example in Figures 8.4–8.7 shows how the grid definition and inquiry functions can be used in an application program. A partial differential equation, for instance the Poisson equation, is to be solved on a rectangular domain. First, the MPI processes organize themselves in a two-dimensional structure. Each MPI 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 MPI process computes new values for the solution grid function at the points $u(1:100,1:100)$ owned by the MPI process. Then the values at inter-process boundaries have to be exchanged with neighboring MPI 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 $(\text{own_coord}(1)-1, \text{own_coord}(2))$.

```

1  SUBROUTINE exchange(u, comm_cart, neigh_rank, num_neigh)
2  USE MPI
3  REAL u(0:101,0:101)
4  INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
5  REAL sndbuf(100,num_neigh), rcvbuf(100,num_neigh)
6  INTEGER ierr
7  sndbuf(1:100,1) = u( 1,1:100)
8  sndbuf(1:100,2) = u(100,1:100)
9  sndbuf(1:100,3) = u(1:100, 1)
10 sndbuf(1:100,4) = u(1:100,100)
11 CALL MPI_NEIGHBOR_ALLTOALL(sndbuf, 100, MPI_REAL, rcvbuf, 100, MPI_REAL, &
12                               comm_cart, ierr)
13 ! instead of
14 ! CALL MPI_IRECV(rcvbuf(1,1),100,MPI_REAL, neigh_rank(1),..., rq(1), ierr)
15 ! CALL MPI_ISEND(sndbuf(1,2),100,MPI_REAL, neigh_rank(2),..., rq(2), ierr)
16 ! Always pairing a receive from rank_source with a send to rank_dest
17 ! of the same direction in MPI_CART_SHIFT!
18 ! CALL MPI_IRECV(rcvbuf(1,2),100,MPI_REAL, neigh_rank(2),..., rq(3), ierr)
19 ! CALL MPI_ISEND(sndbuf(1,1),100,MPI_REAL, neigh_rank(1),..., rq(4), ierr)
20 ! CALL MPI_IRECV(rcvbuf(1,3),100,MPI_REAL, neigh_rank(3),..., rq(5), ierr)
21 ! CALL MPI_ISEND(sndbuf(1,4),100,MPI_REAL, neigh_rank(4),..., rq(6), ierr)
22 ! CALL MPI_IRECV(rcvbuf(1,4),100,MPI_REAL, neigh_rank(4),..., rq(7), ierr)
23 ! CALL MPI_ISEND(sndbuf(1,3),100,MPI_REAL, neigh_rank(3),..., rq(8), ierr)
24 ! Of course, one can first start all four IRECV and then all four ISEND,
25 ! Or vice versa, but both in the sequence shown above. Otherwise, the
26 ! matching would be wrong for 2 or only 1 MPI processes in a direction.
27 ! CALL MPI_WAITALL(2*num_neigh, rq, statuses, ierr)
28 u( 0,1:100) = rcvbuf(1:100,1)
29 u(101,1:100) = rcvbuf(1:100,2)
30 u(1:100, 0) = rcvbuf(1:100,3)
31 u(1:100,101) = rcvbuf(1:100,4)
32 END

```

Figure 8.5: Communication routine with local data copying and sparse neighborhood all-to-all

```

SUBROUTINE exchange(u, comm_cart, neigh_rank, num_neigh)
USE MPI
IMPLICIT NONE
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 8.6: Communication routine with sparse neighborhood all-to-all-w and without local data copying

```

1  INTEGER ndims, num_neigh
2  LOGICAL reorder
3  PARAMETER (ndims=2, num_neigh=4, reorder=.true.)
4  INTEGER comm, comm_size, comm_cart, dims(ndims), it, ierr
5  LOGICAL periods(ndims)
6  REAL u(0:101,0:101), f(0:101,0:101)
7  DATA dims / ndims * 0 /
8  INTEGER sndcounts(num_neigh), sndtypes(num_neigh)
9  INTEGER rcvcounts(num_neigh), rcvtypes(num_neigh)
10 INTEGER(KIND=MPI_ADDRESS_KIND) lb, sizeofreal
11 INTEGER(KIND=MPI_ADDRESS_KIND) sdispls(num_neigh), rdispls(num_neigh)
12 INTEGER type_vec, request, info, status(MPI_STATUS_SIZE)
13 comm = MPI_COMM_WORLD
14 CALL MPI_COMM_SIZE(comm, comm_size, ierr)
15 ! Set MPI process grid size and periodicity
16 CALL MPI_DIMS_CREATE(comm_size, ndims, dims, ierr)
17 periods(1) = .TRUE.
18 periods(2) = .TRUE.
19 ! Create a grid structure in WORLD group
20 CALL MPI_CART_CREATE(comm, ndims, dims, periods, reorder, &
21                       comm_cart, ierr)
22 ! Create datatypes for the neighborhood communication
23 !
24 ! Insert code from example in Figure 7.4 to create and initialize
25 ! sndcounts, sdispls, sndtypes, rcvcounts, rdispls, and rcvtypes
26 !
27 ! Initialize the neighborhood all-to-all-w operation
28 info = MPI_INFO_NULL
29 CALL MPI_NEIGHBOR_ALLTOALLW_INIT(u, sndcounts, sdispls, sndtypes, &
30                                  u, rcvcounts, rdispls, rcvtypes, &
31                                  comm_cart, info, request, ierr)
32 ! Initialize the grid functions and start the iteration
33 CALL init(u, f)
34 DO it=1,100
35 ! Start data exchange with neighbor processes
36 CALL MPI_START(request, ierr)
37 ! Compute inner cells
38 CALL relax_inner (u, f)
39 ! Check on completion of neighbor exchange
40 CALL MPI_WAIT(request, status, ierr)
41 ! Compute edge cells
42 CALL relax_edges(u, f)
43 END DO
44 CALL output(u)
45 CALL MPI_REQUEST_FREE(request, ierr)
46 CALL MPI_TYPE_FREE(type_vec, ierr)

```

Figure 8.7: Two-dimensional parallel Poisson solver with persistent sparse neighborhood all-to-all-w and without local data copying

Chapter 9

MPI Environmental Management

This chapter discusses routines for getting and, where appropriate, setting various parameters that relate to the MPI implementation and the execution environment (such as error handling). The procedures for entering and leaving the MPI execution environment are also described here.

9.1 Implementation Information

9.1.1 Version Inquiries

In order to cope with changes to the MPI Standard, there are both compile-time and runtime ways to determine which version of the standard is in use in the environment one is using.

The “version” will be represented by two separate integers, for the version and subversion: In C,

```
#define MPI_VERSION 4
#define MPI_SUBVERSION 0
```

in Fortran,

```
INTEGER :: MPI_VERSION, MPI_SUBVERSION
PARAMETER (MPI_VERSION = 4)
PARAMETER (MPI_SUBVERSION = 0)
```

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)
```

Fortran 2008 binding

```
MPI_Get_version(version, subversion, ierror)
  INTEGER, INTENT(OUT) :: version, subversion
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)
```

1 INTEGER VERSION, SUBVERSION, IERROR

2 MPI_GET_VERSION can be called at any time in an MPI program. This function must
3 always be thread-safe, as defined in Section 11.6. Valid (MPI_VERSION, MPI_SUBVERSION)
4 pairs in this and previous versions of the MPI standard are (4,0), (3,1), (3,0), (2,2), (2,1),
5 (2,0), and (1,2).
6

7
8 MPI_GET_LIBRARY_VERSION(version, resultlen)

9 OUT version version number (string)
10 OUT resultlen Length (in printable characters) of the result
11 returned in version (integer)
12

13 **C binding**

14 int MPI_Get_library_version(char *version, int *resultlen)

15 **Fortran 2008 binding**

16 MPI_Get_library_version(version, resultlen, ierror)
17 CHARACTER(LEN=MPI_MAX_LIBRARY_VERSION_STRING), INTENT(OUT) :: version
18 INTEGER, INTENT(OUT) :: resultlen
19 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21

22 **Fortran binding**

23 MPI_GET_LIBRARY_VERSION(VERSION, RESULTLEN, IERROR)
24 CHARACTER*(*) VERSION
25 INTEGER RESULTLEN, IERROR
26

27 This routine returns a string representing the version of the MPI library. The version
28 argument is a character string for maximum flexibility.

29 *Advice to implementors.* An implementation of MPI should return a different string
30 for every change to its source code or build that could be visible to the user. (*End of*
31 *advice to implementors.*)
32

33 The argument version must represent storage that is
34 MPI_MAX_LIBRARY_VERSION_STRING characters long. MPI_GET_LIBRARY_VERSION may
35 write up to this many characters into version.

36 The number of characters actually written is returned in the output argument, resultlen.
37 In C, a null character is additionally stored at version[resultlen]. The value of resultlen cannot
38 be larger than MPI_MAX_LIBRARY_VERSION_STRING - 1. In Fortran, version is padded on
39 the right with blank characters. The value of resultlen cannot be larger than
40 MPI_MAX_LIBRARY_VERSION_STRING.

41 MPI_GET_LIBRARY_VERSION can be called at any time in an MPI program. This
42 function must always be thread-safe, as defined in Section 11.6.
43

44 9.1.2 Environmental Inquiries

45
46 When using the World Model (Section 11.2), a set of attributes that describe the execution
47 environment is attached to the communicator MPI_COMM_WORLD when MPI is initialized.
48 The values of these attributes can be inquired by using the function

MPI_COMM_GET_ATTR described in Section 7.7 and in Section 19.3.7. It is erroneous to 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.

When using the Sessions Model (Section 11.3), only the MPI_TAG_UB attribute is available.

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.

In the Sessions Model, the attribute MPI_TAG_UB is attached to all communicators created by MPI_COMM_CREATE_FROM_GROUP and MPI_INTERCOMM_CREATE_FROM_GROUPS, with the same value on all MPI processes in the communicator. In the World Model, the attribute MPI_TAG_UB has the same value on all processes of MPI_COMM_WORLD.

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 require 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

1 (e.g., OPEN, REWIND, WRITE). For C, this means that all of the ISO C I/O operations are
 2 supported (e.g., fopen, fprintf, lseek).

3 If every process can provide language-standard I/O, then the value MPI_ANY_SOURCE
 4 will be returned. Otherwise, if the calling process can provide language-standard I/O,
 5 then its rank will be returned. Otherwise, if some process can provide language-standard
 6 I/O then the rank of one such process will be returned. The same value need not be
 7 returned by all processes. If no process can provide language-standard I/O, then the value
 8 MPI_PROC_NULL will be returned.

9
 10 *Advice to users.* Note that input is not collective, and this attribute does *not* indicate
 11 which process can or does provide input. (*End of advice to users.*)

12 Clock Synchronization

13
 14 The value returned for MPI_WTIME_IS_GLOBAL is 1 if clocks at all processes in
 15 MPI_COMM_WORLD are synchronized, 0 otherwise. A collection of clocks is considered
 16 synchronized if explicit effort has been taken to synchronize them. The expectation is that
 17 the variation in time, as measured by calls to MPI_WTIME, will be less than one half the
 18 round-trip time for an MPI message of length zero. If time is measured at a process just
 19 before a send and at another process just after a matching receive, the second time should
 20 be always higher than the first one.

21
 22 The attribute MPI_WTIME_IS_GLOBAL need not be present when the clocks are not
 23 synchronized (however, the attribute key MPI_WTIME_IS_GLOBAL is always valid). This
 24 attribute may be associated with communicators other than MPI_COMM_WORLD.

25 The attribute MPI_WTIME_IS_GLOBAL has the same value on all processes of
 26 MPI_COMM_WORLD.

27 Inquire Processor Name

28
 29
 30
 31 MPI_GET_PROCESSOR_NAME(name, resultlen)

32	OUT	name	A unique specifier for the actual (as opposed to
33			virtual) node.
34			
35	OUT	resultlen	Length (in printable characters) of the result
36			returned in name

37 C binding

38
 39 int MPI_Get_processor_name(char *name, int *resultlen)

40 Fortran 2008 binding

41 MPI_Get_processor_name(name, resultlen, ierror)
 42 CHARACTER(LEN=MPI_MAX_PROCESSOR_NAME), INTENT(OUT) :: name
 43 INTEGER, INTENT(OUT) :: resultlen
 44 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

45 Fortran binding

46
 47 MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERROR)
 48 CHARACTER*(*) NAME

INTEGER RESULTLEN, IERROR

This routine returns the name of the processor on which it was called at the moment of the call. The name is a character string for maximum flexibility. From this value it must be possible to identify a specific piece of hardware; possible values include “processor 9 in rack 4 of mpp.cs.org” and “231” (where 231 is the actual processor number in the running homogeneous system). The argument `name` must represent storage that is at least `MPI_MAX_PROCESSOR_NAME` characters long. `MPI_GET_PROCESSOR_NAME` may write up to this many characters into `name`.

The number of characters actually written is returned in the output argument, `resultlen`. In C, a null character is additionally stored at `name[resultlen]`. The value of `resultlen` cannot be larger than `MPI_MAX_PROCESSOR_NAME-1`. In Fortran, `name` is padded on the right with blank characters. The value of `resultlen` cannot be larger than `MPI_MAX_PROCESSOR_NAME`.

Rationale. This function allows MPI implementations that do process migration to return the current processor. Note that nothing in MPI *requires* or defines process migration; this definition of `MPI_GET_PROCESSOR_NAME` simply allows such an implementation. (*End of rationale.*)

Advice to users. The user must provide at least `MPI_MAX_PROCESSOR_NAME` space to write the processor name—processor names can be this long. The user should examine the output argument, `resultlen`, to determine the actual length of the name. (*End of advice to users.*)

9.2 Memory Allocation

In some systems, message-passing and remote-memory-access (RMA) operations run faster when accessing specially allocated memory (e.g., memory that is shared by the other processes in the communicating group on an SMP). MPI provides a mechanism for allocating and freeing such special memory. The use of such memory for message-passing or RMA is not mandatory, and this memory can be used without restrictions as any other dynamically allocated memory. However, implementations may restrict the use of some RMA functionality as defined in Section 12.5.3.

`MPI_ALLOC_MEM(size, info, baseptr)`

IN	size	size of memory segment in bytes (non-negative integer)
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)`

Fortran 2008 binding

`MPI_Alloc_mem(size, info, baseptr, ierror)`

USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR

INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size

```

1     TYPE(MPI_Info), INTENT(IN) :: info
2     TYPE(C_PTR), INTENT(OUT) :: baseptr
3     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4

```

Fortran binding

```

5 MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
6     INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
7     INTEGER INFO, IERROR
8

```

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:

```

14 INTERFACE MPI_ALLOC_MEM
15     SUBROUTINE MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
16         IMPORT :: MPI_ADDRESS_KIND
17         INTEGER :: INFO, IERROR
18         INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR
19     END SUBROUTINE
20     SUBROUTINE MPI_ALLOC_MEM_CPTR(SIZE, INFO, BASEPTR, IERROR)
21         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
22         IMPORT :: MPI_ADDRESS_KIND
23         INTEGER :: INFO, IERROR
24         INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
25         TYPE(C_PTR) :: BASEPTR
26     END SUBROUTINE
27 END INTERFACE
28

```

The base procedure name of this overloaded function is `MPI_ALLOC_MEM_CPTR`. The implied specific procedure names are described in Section 19.1.5.

By default, the allocated memory shall be aligned to at least the alignment required for load/store accesses of any datatype corresponding to a predefined MPI datatype. The `info` argument may be used to specify a desired alternative minimum alignment in bytes for the allocated memory by setting the value of the key `"mpi_minimum_memory_alignment"` to an integral number equal to a power of two. An implementation may ignore values smaller than the default required alignment. The `info` argument can also 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. The corresponding `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.

```

43 MPI_FREE_MEM(base)
44

```

```

45     IN         base                initial address of memory segment allocated by
46                                     MPI_ALLOC_MEM (choice)
47
48

```

C binding

```
int MPI_Free_mem(void *base)
```

Fortran 2008 binding

```
MPI_Free_mem(base, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: base
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran 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 9.1. 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 9.2. Example of use of `MPI_ALLOC_MEM`, in Fortran with nonstandard **Cray-pointers**. We assume 4-byte REALs, and assume that these pointers are address-sized.

```

1  REAL A
2  POINTER (P, A(100,100)) ! no memory is allocated
3
4  INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
5  SIZE = 4*100*100
6
7  CALL MPI_ALLOC_MEM(SIZE, MPI_INFO_NULL, P, IERR)
8  ! memory is allocated
9  ...
10 A(3,5) = 2.71
11 ...
12 CALL MPI_FREE_MEM(A, IERR) ! memory is freed

```

This code is not Fortran 77 or Fortran 90 code. Some compilers may not support this code or need a special option, e.g., the GNU gFortran compiler needs `-fcray-pointer`.

Advice to implementors. Some compilers map Cray-pointers to address-sized integers, some to `TYPE(C_PTR)` pointers (e.g., Cray Fortran, version 7.3.3). From the user's viewpoint, this mapping is irrelevant because Examples 9.2 should work correctly with an MPI-3.0 (or later) library if Cray-pointers are available. (*End of advice to implementors.*)

Example 9.3. Same example, in C.

```

24 float (*f)[100][100];
25 /* no memory is allocated */
26 MPI_Alloc_mem(sizeof(float)*100*100, MPI_INFO_NULL, &f);
27 /* memory allocated */
28 ...
29 (*f)[5][3] = 2.71;
30 ...
31 MPI_Free_mem(f);

```

9.3 Error Handling

An MPI implementation may be unable or choose not to handle some failures that occur during MPI calls. These can include failures that generate exceptions or traps, such as floating point errors or access violations. The set of failures that are handled by MPI is implementation-dependent. Each such failure causes an error to be raised.

The above text takes precedence over any text on error handling within this document. Specifically, text that states that errors *will* be handled should be read as *may* be handled. More background information about how MPI treats errors can be found in Section 2.8.

A user can associate error handlers to four types of objects: communicators, windows, files, and sessions. The specified error handling routine will be used for any error that occurs during a call to MPI for the respective object. MPI calls that are not related to any MPI objects are considered to be attached to the communicator `MPI_COMM_SELF` when using the World Model (see Section 11.2). When `MPI_COMM_SELF` is not initialized (i.e., before `MPI_INIT` / `MPI_INIT_THREAD`, after `MPI_FINALIZE`, or when using the Sessions Model exclusively) the error raises the initial error handler (set during the launch operation,

see 11.8.4). The attachment of error handlers to objects is purely local: different processes may attach different error handlers to corresponding objects.

Several predefined error handlers are available in MPI:

MPI_ERRORS_ARE_FATAL: The handler, when called, causes the program to abort all connected MPI processes. This is similar to calling MPI_ABORT using a communicator containing all connected processes with an implementation-specific value as the `errorcode` argument.

MPI_ERRORS_ABORT: The handler, when called, is invoked on a communicator in a manner similar to calling MPI_ABORT on that communicator. If the error handler is invoked on an window or file, it is similar to calling MPI_ABORT using a communicator containing the group of MPI processes associated with the window or file, respectively. If the error handler is invoked on a session, the operation aborts only the local MPI process. In all cases, the value that would be provided as the `errorcode` argument to MPI_ABORT is implementation-specific.

MPI_ERRORS_RETURN: The handler has no effect other than returning the error code to the user.

Advice to implementors. The implementation-specific error information resulting from MPI_ERRORS_ARE_FATAL and MPI_ERRORS_ABORT provided to the invoking environment should be meaningful to the end-user, for example a predefined error class. (*End of advice to implementors.*)

Implementations may provide additional predefined error handlers and programmers can code their own error handlers.

Unless otherwise requested, the error handler MPI_ERRORS_ARE_FATAL is set as the default initial error handler and associated with predefined communicators. Thus, if the user chooses not to control error handling, every error that MPI handles is treated as fatal. Since (almost) all MPI calls return an error code, a user may choose to handle errors in its main code, by testing the return code of MPI calls and executing a suitable recovery code when the call was not successful. In this case, the error handler MPI_ERRORS_RETURN will be used. Usually it is more convenient and more efficient not to test for errors after each MPI call, and have such error handled by a nontrivial MPI error handler. Note that unlike predefined communicators, windows and files do not inherit from the initial error handler, as defined in Sections 12.6 and 14.7 respectively.

When an error is raised, MPI will provide the user information about that error using an error code. Some errors might prevent MPI from completing further API calls successfully and those functions will continue to report errors until the cause of the error is corrected or the user terminates the application. The user can make the determination of whether or not to attempt to continue when handling such an error.

Advice to users. For example, users may be unable to correct errors corresponding to some error classes, such as MPI_ERR_INTERN. Such errors may cause subsequent MPI calls to complete in error. (*End of advice to users.*)

Advice to implementors. A high-quality implementation will, to the greatest possible extent, circumscribe the impact of an error, so that normal processing can continue after an error handler was invoked. The implementation documentation will provide

1 information on the possible effect of each class of errors and available recovery actions.
 2 (*End of advice to implementors.*)
 3

4 An MPI error handler is an opaque object, which is accessed by a handle. MPI calls
 5 are provided to create new error handlers, to associate error handlers with objects, and to
 6 test which error handler is associated with an object. C has distinct typedefs for user de-
 7 fined error handling callback functions that accept communicator, file, window, and session
 8 arguments. In Fortran there are four user routines.

9 An error handler object is created by a call to `MPI_XXX_CREATE_ERRHANDLER`,
 10 where XXX is, respectively, `COMM`, `WIN`, `FILE`, or `SESSION`.

11 An error handler is attached to a communicator, window, file, or session by a call to
 12 `MPI_XXX_SET_ERRHANDLER`. The error handler must be either a predefined error handler,
 13 or an error handler that was created by a call to `MPI_XXX_CREATE_ERRHANDLER`, with
 14 matching XXX. An error handler can also be attached to a session using the `errorhandler`
 15 argument to `MPI_SESSION_INIT`. The predefined error handlers `MPI_ERRORS_RETURN` and
 16 `MPI_ERRORS_ARE_FATAL` can be attached to communicators, windows, files, or sessions.

17 The error handler currently associated with a communicator, window, file, or session
 18 can be retrieved by a call to `MPI_XXX_GET_ERRHANDLER`.

19 The MPI function `MPI_ERRHANDLER_FREE` can be used to free an error handler that
 20 was created by a call to `MPI_XXX_CREATE_ERRHANDLER`.

21 `MPI_XXX_GET_ERRHANDLER` behave as if a new error handler object is created. That
 22 is, once the error handler is no longer needed, `MPI_ERRHANDLER_FREE` should be called
 23 with the error handler returned from `MPI_XXX_GET_ERRHANDLER` to mark the error
 24 handler for deallocation. This provides behavior similar to that of `MPI_COMM_GROUP`
 25 and `MPI_GROUP_FREE`.

26
 27 *Advice to implementors.* High-quality implementations should raise an error when
 28 an error handler that was created by a call to `MPI_XXX_CREATE_ERRHANDLER` is
 29 attached to an object of the wrong type with a call to `MPI_YYY_SET_ERRHANDLER`.
 30 To do so, it is necessary to maintain, with each error handler, information on the
 31 typedef of the associated user function. (*End of advice to implementors.*)

32 The syntax for these calls is given below.
 33

34 9.3.1 Error Handlers for Communicators

35
 36
 37 `MPI_COMM_CREATE_ERRHANDLER(comm_errhandler_fn, errhandler)`

38	IN	<code>comm_errhandler_fn</code>	user defined error handling procedure (function)
39			
40	OUT	<code>errhandler</code>	MPI error handler (handle)
41			

42 C binding

```
43 int MPI_Comm_create_errhandler(
44     MPI_Comm_errhandler_function *comm_errhandler_fn,
45     MPI_Errhandler *errhandler)
46
```

47 Fortran 2008 binding

```
48 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

```

Fortran 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 *comm, int *error_code,
    . . . );

```

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)
```

Fortran 2008 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
```

Fortran 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)
```

Fortran 2008 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
```

Fortran 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.

9.3.2 Error Handlers for Windows

```
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)
```

Fortran 2008 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
```

Fortran 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 *win, int *error_code, . . .);
```

The first argument is the window in use, the second is the error code to be returned. The remaining arguments are “varargs” arguments whose number and meaning is implementation-dependent. An implementation should clearly document these arguments. 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 object (handle)
IN	errhandler	new error handler for window (handle)

C binding

```
int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)
```

Fortran 2008 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
```

Fortran binding

```
MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
  INTEGER WIN, ERRHANDLER, IERROR
```

1 Attaches a new error handler to a window. The error handler must be either a pre-
 2 defined error handler, or an error handler created by a call to
 3 MPI_WIN_CREATE_ERRHANDLER.

4
 5
 6 MPI_WIN_GET_ERRHANDLER(win, errhandler)

7	IN	win	window object (handle)
8			
9	OUT	errhandler	error handler currently associated with window (handle)
10			

11 C binding

12 int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)

13 Fortran 2008 binding

14 MPI_Win_get_errhandler(win, errhandler, ierror)
 15 TYPE(MPI_Win), INTENT(IN) :: win
 16 TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
 17 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

18 Fortran binding

19 MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
 20 INTEGER WIN, ERRHANDLER, IERROR

21 Retrieves the error handler currently associated with a window.

22 9.3.3 Error Handlers for Files

23
 24
 25
 26
 27
 28 MPI_FILE_CREATE_ERRHANDLER(file_errhandler_fn, errhandler)

29	IN	file_errhandler_fn	user defined error handling procedure (function)
30			
31	OUT	errhandler	MPI error handler (handle)
32			

33 C binding

34 int MPI_File_create_errhandler(
 35 MPI_File_errhandler_function *file_errhandler_fn,
 36 MPI_Errhandler *errhandler)

37 Fortran 2008 binding

38 MPI_File_create_errhandler(file_errhandler_fn, errhandler, ierror)
 39 PROCEDURE(MPI_File_errhandler_function) :: file_errhandler_fn
 40 TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
 41 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

42 Fortran binding

43 MPI_FILE_CREATE_ERRHANDLER(FILE_ERRHANDLER_FN, ERRHANDLER, IERROR)
 44 EXTERNAL FILE_ERRHANDLER_FN
 45 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 *file, int *error_code,
    . . . );
```

The first argument is the file in use, the second is the error code to be returned. The remaining arguments are “varargs” arguments whose number and meaning is implementation-dependent. An implementation should clearly document these arguments.

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)
```

Fortran 2008 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
```

Fortran 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)
```

Fortran 2008 binding

```

1 MPI_File_get_errhandler(file, errhandler, ierror)
2     TYPE(MPI_File), INTENT(IN) :: file
3     TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
4     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

7 MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
8     INTEGER FILE, ERRHANDLER, IERROR

```

Retrieves the error handler currently associated with a file.

9.3.4 Error Handlers for Sessions

```

15 MPI_SESSION_CREATE_ERRHANDLER(session_errhandler_fn, errhandler)

```

17	IN	session_errhandler_fn	user defined error handling procedure (function)
18	OUT	errhandler	MPI error handler (handle)

C binding

```

21 int MPI_Session_create_errhandler(
22     MPI_Session_errhandler_function *session_errhandler_fn,
23     MPI_Errhandler *errhandler)
24

```

Fortran 2008 binding

```

26 MPI_Session_create_errhandler(session_errhandler_fn, errhandler, ierror)
27     PROCEDURE(MPI_Session_errhandler_function) :: session_errhandler_fn
28     TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
29     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

31 MPI_SESSION_CREATE_ERRHANDLER(SESSION_ERRHANDLER_FN, ERRHANDLER, IERROR)
32     EXTERNAL SESSION_ERRHANDLER_FN
33     INTEGER ERRHANDLER, IERROR

```

Creates an error handler that can be attached to a session object. In C, the `session_errhandler_fn` argument should be a function of type `MPI_Session_errhandler_function`, which is defined as

```

38 typedef void MPI_Session_errhandler_function(MPI_Session *session,
39     int *error_code, . . . );

```

The first argument is the session in use, the second is the error code to be returned. The remaining arguments are “varargs” arguments whose number and meaning is implementation-dependent. An implementation should clearly document these arguments. With the Fortran `mpi_f08` module, the `session_errhandler_fn` argument should be of the form:

```

44 ABSTRACT INTERFACE
45     SUBROUTINE MPI_Session_errhandler_function(session, error_code)
46     TYPE(MPI_Session) :: session
47     INTEGER :: error_code
48

```


With the Fortran `mpi` module and `mpif.h`, the `SESSION_ERRHANDLER_FN` argument should be of the form:

```
SUBROUTINE SESSION_ERRHANDLER_FUNCTION(SESSION, ERROR_CODE)
  INTEGER SESSION, ERROR_CODE
```

`MPI_SESSION_SET_ERRHANDLER(session, errhandler)`

INOUT	session	session (handle)
IN	errhandler	new error handler for session (handle)

C binding

```
int MPI_Session_set_errhandler(MPI_Session session, MPI_Errhandler errhandler)
```

Fortran 2008 binding

```
MPI_Session_set_errhandler(session, errhandler, ierror)
  TYPE(MPI_Session), INTENT(IN) :: session
  TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_SESSION_SET_ERRHANDLER(SESSION, ERRHANDLER, IERROR)
  INTEGER SESSION, ERRHANDLER, IERROR
```

Attaches a new error handler to a session. The error handler must be either a pre-defined error handler, or an error handler created by a call to `MPI_SESSION_CREATE_ERRHANDLER`.

`MPI_SESSION_GET_ERRHANDLER(session, errhandler)`

IN	session	session (handle)
OUT	errhandler	error handler currently associated with session (handle)

C binding

```
int MPI_Session_get_errhandler(MPI_Session session, MPI_Errhandler *errhandler)
```

Fortran 2008 binding

```
MPI_Session_get_errhandler(session, errhandler, ierror)
  TYPE(MPI_Session), INTENT(IN) :: session
  TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_SESSION_GET_ERRHANDLER(SESSION, ERRHANDLER, IERROR)
  INTEGER SESSION, ERRHANDLER, IERROR
```

Retrieves the error handler currently associated with a session.

9.3.5 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)
```

Fortran 2008 binding

```
MPI_Errhandler_free(errhandler, ierror)
  TYPE(MPI_Errhandler), INTENT(INOUT) :: errhandler
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran 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
OUT	string	Text that corresponds to the errorcode
OUT	resultlen	Length (in printable characters) of the result returned in string

C binding

```
int MPI_Error_string(int errorcode, char *string, int *resultlen)
```

Fortran 2008 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
```

Fortran 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`.

This function must always be thread-safe, as defined in Section 11.6. It is one of the few routines that may be called before MPI is initialized or after MPI is finalized.

Rationale. The form of this function was chosen to make the Fortran and C bindings similar. A version that returns a pointer to a string has two difficulties. First, the return string must be statically allocated and different for each error message (allowing the pointers returned by successive calls to `MPI_ERROR_STRING` to point to the correct message). Second, in Fortran, a function declared as returning `CHARACTER*(*)` can not be referenced in, for example, a `PRINT` statement. (*End of rationale.*)

9.4 Error Codes and Classes

The error codes returned by MPI are left entirely to the implementation (with the exception of `MPI_SUCCESS`). This is done to allow an implementation to provide as much information as possible in the error code (for use with `MPI_ERROR_STRING`).

All MPI function calls shall return `MPI_SUCCESS` if and only if the specification of that function has been fulfilled at the point of return. For multiple completion functions, if the function returns `MPI_ERR_IN_STATUS`, the error code in each status object shall be set to `MPI_SUCCESS` if and only if the specification of the operation represented by the corresponding `MPI_Request` has been fulfilled at the point of return.

When an operation raises an error, it may not satisfy its specification (for example, a synchronizing operation may not have synchronized) and the content of the output buffers, targeted memory, or output parameters is undefined. However, a valid error code shall always be set when an operation raises an error, whether in the return value, error field in the status object, or element in an array of error codes.

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 9.1 and Table 9.2.

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,

$$0 = \text{MPI_SUCCESS} < \text{MPI_ERR_...} \leq \text{MPI_ERR_LASTCODE}.$$

Rationale. The difference between `MPI_ERR_UNKNOWN` and `MPI_ERR_OTHER` is that `MPI_ERROR_STRING` can return useful information about `MPI_ERR_OTHER`.

Note that `MPI_SUCCESS = 0` is necessary to be consistent with C practice; the separation of error classes and error codes allows us to define the error classes this way. Having a known `LASTCODE` is often a nice sanity check as well. (*End of rationale.*)

`MPI_ERROR_CLASS(errorcode, errorclass)`

IN	errorcode	Error code returned by an MPI routine
OUT	errorclass	Error class associated with errorcode

C binding

`int MPI_Error_class(int errorcode, int *errorclass)`

Table 9.1: Error classes (Part 1)

MPI_SUCCESS	No error
MPI_ERR_ACCESS	Permission denied
MPI_ERR_AMODE	Error related to the amode passed to MPI_FILE_OPEN
MPI_ERR_ARG	Invalid argument of some other kind
MPI_ERR_ASSERT	Invalid assertion argument
MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)
MPI_ERR_BASE	Invalid base passed to MPI_FREE_MEM
MPI_ERR_BUFFER	Invalid buffer pointer argument
MPI_ERR_COMM	Invalid communicator argument
MPI_ERR_CONVERSION	An error occurred in a user supplied data conversion function
MPI_ERR_COUNT	Invalid count argument
MPI_ERR_DIMS	Invalid dimension argument
MPI_ERR_DISP	Invalid displacement argument
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_ERRHANDLER	Invalid error handler argument
MPI_ERR_FILE	Invalid file handle argument
MPI_ERR_FILE_EXISTS	File exists
MPI_ERR_FILE_IN_USE	File operation could not be completed, as the file is currently open by some process
MPI_ERR_GROUP	Invalid group argument
MPI_ERR_INFO	Invalid info argument
MPI_ERR_INFO_KEY	Key longer than MPI_MAX_INFO_KEY
MPI_ERR_INFO_NOKEY	Invalid key passed to MPI_INFO_DELETE
MPI_ERR_INFO_VALUE	Value longer than MPI_MAX_INFO_VAL
MPI_ERR_IN_STATUS	Error code is in status
MPI_ERR_INTERN	Internal MPI (implementation) error
MPI_ERR_IO	Other I/O error
MPI_ERR_KEYVAL	Invalid keyval argument
MPI_ERR_LOCKTYPE	Invalid locktype argument
MPI_ERR_NAME	Invalid service name passed to MPI_LOOKUP_NAME
MPI_ERR_NO_MEM	MPI_ALLOC_MEM failed because memory is exhausted
MPI_ERR_NO_SPACE	Not enough space
MPI_ERR_NO_SUCH_FILE	File does not exist
MPI_ERR_NOT_SAME	Collective argument not identical on all processes, or collective routines called in a different order by different processes

Table 9.2: Error classes (Part 2)		1
MPI_ERR_OP	Invalid operation argument	3
MPI_ERR_OTHER	Known error not in this list	4
MPI_ERR_PENDING	Pending request	5
MPI_ERR_PORT	Invalid port name passed to MPI_COMM_CONNECT	6 7
MPI_ERR_PROC_ABORTED	Operation failed because a peer process has aborted	8 9
MPI_ERR_QUOTA	Quota exceeded	10
MPI_ERR_RANK	Invalid rank argument	11
MPI_ERR_READ_ONLY	Read-only file or file system	12
MPI_ERR_REQUEST	Invalid request argument	13
MPI_ERR_RMA_ATTACH	Memory cannot be attached (e.g., because of resource exhaustion)	14 15
MPI_ERR_RMA_CONFLICT	Conflicting accesses to window	16
MPI_ERR_RMA_FLAVOR	Passed window has the wrong flavor for the called function	17 18
MPI_ERR_RMA_RANGE	Target memory is not part of the win- dow (in the case of a window created with MPI_WIN_CREATE_DYNAMIC, tar- get memory is not attached)	19 20 21 22
MPI_ERR_RMA_SHARED	Memory cannot be shared (e.g., some pro- cess in the group of the specified commu- nicator cannot expose shared memory)	23 24 25
MPI_ERR_RMA_SYNC	Wrong synchronization of RMA calls	26
MPI_ERR_ROOT	Invalid root argument	27
MPI_ERR_SERVICE	Invalid service name passed to MPI_UNPUBLISH_NAME	28 29
MPI_ERR_SESSION	Invalid session argument	30
MPI_ERR_SIZE	Invalid size argument	31
MPI_ERR_SPAWN	Error in spawning processes	32
MPI_ERR_TAG	Invalid tag argument	33
MPI_ERR_TOPOLOGY	Invalid topology argument	34
MPI_ERR_TRUNCATE	Message truncated on receive	35
MPI_ERR_TYPE	Invalid datatype argument	36
MPI_ERR_UNKNOWN	Unknown error	37
MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to MPI_FILE_SET_VIEW	38 39
MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on a file that supports sequential access only	40 41
MPI_ERR_VALUE_TOO_LARGE	Value is too large to store	42
MPI_ERR_WIN	Invalid window argument	43
MPI_ERR_LASTCODE	Last error code	44

45
46
47
48

Fortran 2008 binding

```

1 MPI_Error_class(errorcode, errorclass, ierror)
2     INTEGER, INTENT(IN) :: errorcode
3     INTEGER, INTENT(OUT) :: errorclass
4     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
6

```

Fortran binding

```

7 MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
8     INTEGER ERRORCODE, ERRORCLASS, IERROR
9
10

```

11 The function `MPI_ERROR_CLASS` maps each standard error code (error class) onto itself.

12 This function must always be thread-safe, as defined in Section 11.6. It is one of the few routines that may be called before MPI is initialized or after MPI is finalized.

15 9.5 Error Classes, Error Codes, and Error Handlers

17 Users may want to write a layered library on top of an existing MPI implementation, and this library may have its own set of error codes and classes. An example of such a library is an I/O library based on MPI, see Chapter 14. For this purpose, functions are needed to:

- 21 1. add a new error class to the ones an MPI implementation already knows.
- 22 2. associate error codes with this error class, so that `MPI_ERROR_CLASS` works.
- 23 3. associate strings with these error codes, so that `MPI_ERROR_STRING` works.
- 24 4. invoke the error handler associated with a communicator, window, or object.

25 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.

```

31
32 MPI_ADD_ERROR_CLASS(errorclass)
33

```

```

34     OUT     errorclass          value for the new error class (integer)
35

```

C binding

```

36 int MPI_Add_error_class(int *errorclass)
37
38

```

Fortran 2008 binding

```

39 MPI_Add_error_class(errorclass, ierror)
40     INTEGER, INTENT(OUT) :: errorclass
41     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42

```

Fortran binding

```

43 MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR)
44     INTEGER ERRORCLASS, IERROR
45

```

46 Creates a new error class and returns the value for it.

Rationale. To avoid conflicts with existing error codes and classes, the value is set by the implementation and not by the user. (*End of rationale.*)

Advice to users. Since a call to `MPI_ADD_ERROR_CLASS` is local, the same `errorclass` may not be returned on all processes that make this call. Thus, it is not safe to assume that registering a new error on a set of processes at the same time will yield the same `errorclass` on all of the processes. Getting the “same” error on multiple processes may not cause the same value of error code to be generated. (*End of advice to users.*)

The value of `MPI_ERR_LASTCODE` is a constant value and is not affected by new user-defined error codes and classes. Instead, when using the World Model (Section 11.2), a predefined attribute key `MPI_LASTUSEDPCODE` 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_LASTUSEDPCODE` will not change unless the user calls a function to explicitly add an error class/code. In a multithreaded 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_LASTUSEDPCODE` is valid. (*End of advice to users.*)

`MPI_ADD_ERROR_CODE(errorclass, errorcode)`

IN	<code>errorclass</code>	error class (integer)
OUT	<code>errorcode</code>	new error code to be associated with <code>errorclass</code> (integer)

C binding

```
int MPI_Add_error_code(int errorclass, int *errorcode)
```

Fortran 2008 binding

```
MPI_Add_error_code(errorclass, errorcode, ierror)
  INTEGER, INTENT(IN) :: errorclass
  INTEGER, INTENT(OUT) :: errorcode
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran 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.*)

```

1 MPI_ADD_ERROR_STRING(errorcode, string)
2     IN          errorcode          error code or class (integer)
3
4     IN          string             text corresponding to errorcode (string)
5

```

C binding

```

6 int MPI_Add_error_string(int errorcode, const char *string)
7
8

```

Fortran 2008 binding

```

9 MPI_Add_error_string(errorcode, string, ierror)
10     INTEGER, INTENT(IN) :: errorcode
11     CHARACTER(LEN=*), INTENT(IN) :: string
12     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13

```

Fortran binding

```

14 MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR)
15     INTEGER ERRORCODE, IERROR
16     CHARACTER*(*) STRING
17
18

```

19 Associates an error string with an error code or class. The string must be no more
20 than MPI_MAX_ERROR_STRING characters long. The length of the string is as defined in the
21 calling language. The length of the string does not include the null terminator in C. Trailing
22 blanks will be stripped in Fortran. Calling MPI_ADD_ERROR_STRING for an errorcode that
23 already has a string will replace the old string with the new string. It is erroneous to call
24 MPI_ADD_ERROR_STRING for an error code or class with a value \leq MPI_ERR_LASTCODE.

25 If MPI_ERROR_STRING is called when no string has been set, it will return a empty
26 string (all spaces in Fortran, "" in C).

27 Section 9.3 describes the methods for creating and associating error handlers with
28 communicators, files, windows, and sessions.

```

29
30 MPI_COMM_CALL_ERRHANDLER(comm, errorcode)
31     IN          comm              communicator with error handler (handle)
32
33     IN          errorcode        error code (integer)
34

```

C binding

```

35 int MPI_Comm_call_errhandler(MPI_Comm comm, int errorcode)
36
37

```

Fortran 2008 binding

```

38 MPI_Comm_call_errhandler(comm, errorcode, ierror)
39     TYPE(MPI_Comm), INTENT(IN) :: comm
40     INTEGER, INTENT(IN) :: errorcode
41     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42

```

Fortran binding

```

43 MPI_COMM_CALL_ERRHANDLER(COMM, ERRORCODE, IERROR)
44     INTEGER COMM, ERRORCODE, IERROR
45
46

```

47 This function invokes the error handler assigned to the communicator with the error
48 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)`

Fortran 2008 binding

`MPI_Win_call_errhandler(win, errorcode, ierror)`
 TYPE(MPI_Win), INTENT(IN) :: win
 INTEGER, INTENT(IN) :: errorcode
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran 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. In contrast to communicators, the error handler `MPI_ERRORS_ARE_FATAL` is associated with a window when it is created. (*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)`

Fortran 2008 binding

`MPI_File_call_errhandler(fh, errorcode, ierror)`
 TYPE(MPI_File), INTENT(IN) :: fh
 INTEGER, INTENT(IN) :: errorcode
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran 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. The default error handler for files is `MPI_ERRORS_RETURN`. (*End of advice to users.*)

```
MPI_SESSION_CALL_ERRHANDLER(session, errorcode)
```

```
IN      session          session with error handler (handle)
```

```
IN      errorcode       error code (integer)
```

C binding

```
int MPI_Session_call_errhandler(MPI_Session session, int errorcode)
```

Fortran 2008 binding

```
MPI_Session_call_errhandler(session, errorcode, ierror)
```

```
TYPE(MPI_Session), INTENT(IN) :: session
```

```
INTEGER, INTENT(IN) :: errorcode
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_SESSION_CALL_ERRHANDLER(SESSION, ERRORCODE, IERROR)
```

```
INTEGER SESSION, ERRORCODE, IERROR
```

This function invokes the error handler assigned to the session with the error code supplied. This function returns `MPI_SUCCESS` in C and the same value in `IERROR` if the error handler was successfully called (assuming the process is not aborted and the error handler returns).

Advice to users. Users are warned that handlers should not be called recursively with `MPI_COMM_CALL_ERRHANDLER`, `MPI_FILE_CALL_ERRHANDLER`, `MPI_WIN_CALL_ERRHANDLER`, or `MPI_SESSION_CALL_ERRHANDLER`. Doing this can create a situation where an infinite recursion is created. This can occur if `MPI_COMM_CALL_ERRHANDLER`, `MPI_FILE_CALL_ERRHANDLER`, `MPI_WIN_CALL_ERRHANDLER`, or `MPI_SESSION_CALL_ERRHANDLER` is called inside an error handler.

Error codes and classes are associated with a process. As a result, they may be used in any error handler. Error handlers should be prepared to deal with any error code they are given. Furthermore, it is good practice to only call an error handler with the appropriate error codes. For example, file errors would normally be sent to the file error handler. (*End of advice to users.*)

9.6 Timers and Synchronization

MPI defines a timer. A timer is specified even though it is not “message-passing,” because timing parallel programs is important in “performance debugging” and because existing

timers (both in POSIX 1003.1-1988 and 1003.4D 14.1 and in Fortran 90) are either inconvenient or do not provide adequate access to high resolution timers. See also Section 2.6.4.

MPI_WTIME()

C binding

double MPI_Wtime(void)

Fortran 2008 binding

DOUBLE PRECISION MPI_Wtime()

Fortran binding

DOUBLE PRECISION MPI_WTIME()

MPI_WTIME returns a floating-point number of seconds, representing elapsed 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”), and it allows high-resolution. One would use it like this:

Example 9.4.

```
{
    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 9.1.2).

MPI_WTICK()

C binding

double MPI_Wtick(void)

Fortran 2008 binding

DOUBLE PRECISION MPI_Wtick()

Fortran 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}) .

DRAFT

Chapter 10

The Info Object

Many of the routines in MPI take an argument `info`. `info` is an opaque object with a handle of type `MPI_Info` in C and Fortran with the `mpi_f08` module, and `INTEGER` in Fortran with the `mpi` module or the include file `mpif.h`. It stores an unordered set of (key,value) pairs (both key and value are strings). A key can have only one value. MPI reserves several keys and requires that if an implementation uses a reserved key, it must provide the specified functionality. An implementation is not required to support these keys and may support any others not reserved by MPI.

Some info hints allow the MPI library to restrict its support for certain operations in order to improve performance or resource utilization. If an application provides such an info hint, it must be compatible with any changes in the behavior of the MPI library that are allowed by the info hint.

An implementation must support info objects as caches for arbitrary (key,value) pairs, regardless of whether it recognizes the key. Each function that takes hints in the form of an `MPI_Info` must be prepared to ignore any key it does not recognize. This description of info objects does not attempt to define how a particular function should react if it recognizes a key but not the associated value. `MPI_INFO_GET_NKEYS`, `MPI_INFO_GET_NTHKEY`, and `MPI_INFO_GET_STRING` must retain all (key,value) pairs so that layered functionality can also use the Info object.

Keys have an implementation-defined maximum length of `MPI_MAX_INFO_KEY`, which is at least 32 and at most 255. Values have an implementation-defined maximum length of `MPI_MAX_INFO_VAL`. In Fortran, leading and trailing spaces are stripped from both. Returned values will never be larger than these maximum lengths. Both key and value are case sensitive.

Rationale. Keys have a maximum length because the set of known keys will always be finite and known to the implementation and because there is no reason for keys to be complex. The small maximum size allows applications to declare keys of size `MPI_MAX_INFO_KEY`. The limitation on value sizes is so that an implementation is not forced to deal with arbitrarily long strings. (*End of rationale.*)

Advice to users. `MPI_MAX_INFO_VAL` might be very large, so it might not be wise to declare a string of that size. (*End of advice to users.*)

When `info` is used as an argument to any MPI routine, it is interpreted before that routine returns, so that it may be read, modified, or freed immediately after return. Changes to an info object after return from a routine do not affect that interpretation.

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

1 implementation must support the following representations. Valid values for a boolean must
 2 include the strings “true” and “false” (all lowercase). For integers, valid values must include
 3 string representations of decimal values of integers that are within the range of a standard
 4 integer type in the program. (However it is possible that not every integer is a valid value
 5 for a given key.) On positive numbers, + signs are optional. No space may appear between
 6 a + or – sign and the leading digit of a number. For comma separated lists, the string
 7 must contain valid elements separated by commas. Leading and trailing spaces are stripped
 8 automatically from the types of info values described above and for each element of a comma
 9 separated list. These rules apply to all info values of these types. Implementations are free
 10 to specify a different interpretation for values of other info keys.
 11

12 **MPI_INFO_CREATE**(info)

13 OUT info info object created (handle)

14
 15
 16 **C binding**

17 int MPI_Info_create(MPI_Info *info)

18
 19 **Fortran 2008 binding**

20 MPI_Info_create(info, ierror)
 21 TYPE(MPI_Info), INTENT(OUT) :: info
 22 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

23 **Fortran binding**

24 MPI_INFO_CREATE(INFO, IERROR)
 25 INTEGER INFO, IERROR

26
 27 MPI_INFO_CREATE creates a new info object. The newly created object contains no
 28 key/value pairs.
 29

30 **MPI_INFO_SET**(info, key, value)

31
 32 INOUT info info object (handle)
 33 IN key key (string)
 34 IN value value (string)

35
 36
 37 **C binding**

38 int MPI_Info_set(MPI_Info info, const char *key, const char *value)

39 **Fortran 2008 binding**

40 MPI_Info_set(info, key, value, ierror)
 41 TYPE(MPI_Info), INTENT(IN) :: info
 42 CHARACTER(LEN=*), INTENT(IN) :: key, value
 43 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

44 **Fortran binding**

45 MPI_INFO_SET(INFO, KEY, VALUE, IERROR)
 46 INTEGER INFO, IERROR
 47 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)
```

Fortran 2008 binding

```
MPI_Info_delete(info, key, ierror)
  TYPE(MPI_Info), INTENT(IN) :: info
  CHARACTER(LEN=*), INTENT(IN) :: key
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran 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_STRING(info, key, buflen, value, flag)

IN	info	info object (handle)
IN	key	key (string)
INOUT	buflen	length of buffer (integer)
OUT	value	value (string)
OUT	flag	true if key defined, false if not (logical)

C binding

```
int MPI_Info_get_string(MPI_Info info, const char *key, int *buflen,
  char *value, int *flag)
```

Fortran 2008 binding

```
MPI_Info_get_string(info, key, buflen, value, flag, ierror)
  TYPE(MPI_Info), INTENT(IN) :: info
  CHARACTER(LEN=*), INTENT(IN) :: key
  INTEGER, INTENT(INOUT) :: buflen
  CHARACTER(LEN=*), INTENT(OUT) :: value
  LOGICAL, INTENT(OUT) :: flag
```

1 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

2 **Fortran binding**

3 MPI_INFO_GET_STRING(INFO, KEY, BUFLen, VALUE, FLAG, IERROR)

4 INTEGER INFO, BUFLen, IERROR

5 CHARACTER*(*) KEY, VALUE

6 LOGICAL FLAG

7

8 This function retrieves the value associated with key from info, if any. If such a key
9 exists in info, it sets flag to true and returns the value in value, otherwise it sets flag to
10 false and leaves value unchanged. buflen on input is the size of the provided buffer, value,
11 for the output of buflen it is the size of the buffer needed to store the value string. If the
12 buflen passed into the function is less than the actual size needed to store the value string
13 (including null terminator in C), the value is truncated. On return, the value of buflen will
14 be set to the required buffer size to hold the value string. If buflen is set to 0, value is
15 not changed. In C, buflen includes the required space for the null terminator. In C, this
16 function returns a null terminated string in all cases where the buflen input value is greater
17 than 0.

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

19

20 *Advice to users.* The MPI_INFO_GET_STRING function can be used to obtain the
21 size of the required buffer for a value string by setting the buflen to 0. The returned
22 buflen can then be used to allocate memory before calling MPI_INFO_GET_STRING
23 again to obtain the value string. (*End of advice to users.*)

24

25

26 MPI_INFO_GET_NKEYS(info, nkeys)

27

28 IN info info object (handle)

29 OUT nkeys number of defined keys (integer)

30

31 **C binding**

32 int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)

33

34 **Fortran 2008 binding**

35 MPI_Info_get_nkeys(info, nkeys, ierror)

36 TYPE(MPI_Info), INTENT(IN) :: info

37 INTEGER, INTENT(OUT) :: nkeys

38 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

39

40 **Fortran binding**

41 MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)

42 INTEGER INFO, NKEYS, IERROR

43 MPI_INFO_GET_NKEYS returns the number of currently defined keys in info.

44

45

46

47

48

MPI_INFO_GET_NTHKEY(info, n, key)				1
IN	info	info object (handle)		2
IN	n	key number (integer)		3
OUT	key	key (string)		4
				5
				6

C binding

```
int MPI_Info_get_nthkey(MPI_Info info, int n, char *key)
```

Fortran 2008 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
```

Fortran binding

```
MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERROR)
  INTEGER INFO, N, IERROR
  CHARACTER*(*) KEY
```

This function returns the *n*th defined key in *info*. Keys are numbered $0 \dots N - 1$ 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)				26
IN	info	info object (handle)		27
OUT	newinfo	info object created (handle)		28
				29
				30

C binding

```
int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo)
```

Fortran 2008 binding

```
MPI_Info_dup(info, newinfo, ierror)
  TYPE(MPI_Info), INTENT(IN) :: info
  TYPE(MPI_Info), INTENT(OUT) :: newinfo
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran 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.

```

1 MPI_INFO_FREE(info)
2     INOUT   info                      info object (handle)
3

```

C binding

```

5 int MPI_Info_free(MPI_Info *info)
6

```

Fortran 2008 binding

```

8 MPI_Info_free(info, ierror)
9     TYPE(MPI_Info), INTENT(INOUT) :: info
10    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11

```

Fortran binding

```

12 MPI_INFO_FREE(INFO, IERROR)
13     INTEGER INFO, IERROR
14

```

15 This function frees info and sets it to MPI_INFO_NULL.

```

18 MPI_INFO_CREATE_ENV(info)
19

```

```

20     OUT     info                      info object (handle)
21

```

C binding

```

22 int MPI_Info_create_env(int argc, char *argv[], MPI_Info *info)
23

```

Fortran 2008 binding

```

24 MPI_Info_create_env(info, ierror)
25     TYPE(MPI_Info), INTENT(OUT) :: info
26     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27

```

Fortran binding

```

28 MPI_INFO_CREATE_ENV(INFO, IERROR)
29     INTEGER INFO, IERROR
30

```

31 This routine produces an output object info with the same construction as MPI_INFO_ENV as created during MPI_INIT or MPI_INIT_THREAD when the same arguments are used. This construction is described in Section 11.2.1; however, this function can be called when not using the World Model, e.g., when using the Sessions Model. This object is not a direct copy or alias of the MPI_INFO_ENV object and could contain different values based on the input arguments and other sources. Multiple calls to this procedure that are given the same input arguments will produce info objects consistent with the definition of MPI_INFO_ENV. The version for ISO C accepts the argc and argv that are provided by the arguments to main or 0 for argc and NULL for argv. The user is responsible for freeing the info object via MPI_INFO_FREE. This procedure is local.

32 This procedure must always be thread-safe, as defined in Section 11.6. It is one of the few routines that may be called before MPI is initialized or after MPI is finalized.

Advice to users.

33 In some circumstances (e.g., when passing 0 to argc and NULL to argv in C or in Fortran where such arguments do not exist), the info object may not be populated or may be populated incompletely because this procedure is local and the implementation may

not be able to determine the correct values. Note that this could result in different values in the resulting info object at different MPI processes.

(End of advice to users.)

DRAFT

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

DRAFT

Chapter 11

Process Initialization, Creation, and Management

11.1 Introduction

MPI is primarily concerned with communication rather than process or resource management. However, it is necessary to address these issues to some degree in order to define a useful framework for communication. This chapter presents a set of MPI interfaces that allows for several approaches to MPI initialization and process management while placing minimal restrictions on the execution environment.

One goal of MPI is to achieve *source code portability*. By this we mean that a program written using MPI and complying with the relevant language standards is portable as written, and must not require any source code changes when moved from one system to another. This explicitly does *not* say anything about how an MPI program is started or launched from the command line, nor what the user must do to set up the environment in which an MPI program will run. However, an implementation may require some setup or initialization procedure to be performed before the complete set of MPI routines may be called.

To this end, MPI presents two models for **MPI process initialization**. In the World Model, an initial set of processes is created that are related by their membership in a common MPI_COMM_WORLD (see Section 11.2) communicator. In the Sessions Model (Section 11.3), an initial set of processes is also created, but the application must explicitly manage the creation of MPI groups, and hence MPI communicators. MPI_COMM_WORLD is only valid for use as a communicator in the World Model, i.e., after a successful call to MPI_INIT or MPI_INIT_THREAD and before a call to MPI_FINALIZE. An application can employ both of these Process Models concurrently. In multi-component MPI applications, for example, a component such as a library can make use of the Sessions Model to instantiate MPI resources without impacting the rest of the application.

The Dynamic Process Model (see Section 11.7), provides for the creation and management of additional processes after an MPI application has been started. A major impetus for the Dynamic Process Model comes from the PVM [25] research effort. This work has provided a wealth of experience with process management and resource control that illustrates their benefits and potential pitfalls.

In developing the Dynamic Process Model, the MPI Forum decided not to address resource control because it was not able to design a portable interface that would be appropriate for the broad spectrum of existing and potential resource and process controllers. MPI assumes that resource control is provided externally.

Process management functionality is included in MPI to enable its use in classes of message-passing applications requiring process control. These include task farms, serial

1 applications with parallel modules, and problems that require a run-time assessment of the
 2 number and type of processes that should be started.

3 The following goals are central to the design of MPI process management:

- 4 • The MPI process model must apply to the vast majority of current parallel environ-
 5 ments.
- 6 • MPI must not take over operating system responsibilities. It should instead provide a
 7 clean interface between an application and system software.
- 8 • MPI must guarantee communication determinism in the presence of dynamic pro-
 9 cesses, i.e., dynamic process management must not introduce unavoidable race condi-
 10 tions.
- 11 • MPI must not contain features that compromise performance.

12
 13 The Dynamic Process Model addresses these issues in two ways. First, MPI remains
 14 primarily a communication library. It does not manage the parallel environment in which
 15 a parallel program executes, though it provides a minimal interface between an application
 16 and external resource and process managers.

17
 18 Second, MPI maintains a consistent concept of a communicator, regardless of how its
 19 members came into existence. A communicator is never changed once created, and it is
 20 always created using deterministic collective operations.
 21

22 11.2 The World Model

23 11.2.1 Starting MPI Processes

24
 25 When using the World Model, MPI is initialized by calling either `MPI_INIT` or
 26 `MPI_INIT_THREAD`.
 27

28
 29
 30 `MPI_INIT()`

31 **C binding**

32 `int MPI_Init(int *argc, char ***argv)`

33 **Fortran 2008 binding**

34 `MPI_Init(ierr)`

35 `INTEGER, OPTIONAL, INTENT(OUT) :: ierr`

36 **Fortran binding**

37 `MPI_INIT(IERROR)`

38 `INTEGER IERROR`

39
 40
 41 In the World Model, an MPI program must contain exactly one call to an MPI ini-
 42 tialization routine: `MPI_INIT` or `MPI_INIT_THREAD`. `MPI_COMM_WORLD` and
 43 `MPI_COMM_SELF` are not valid for use as communicators prior to invocation of `MPI_INIT` or
 44 `MPI_INIT_THREAD`. Subsequent calls to either of these initialization routines are erroneous.
 45 A subset of MPI functions may be invoked before MPI initialization routines are called. See
 46 Section 11.4. `MPI_INIT` accepts the `argc` and `argv` that are provided by the arguments to
 47 `main` or `NULL`:
 48

Example 11.1. Initializing MPI using MPI_INIT

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);

    /* parse arguments */
    /* main program */

    MPI_Finalize();    /* see below */
    return 0;
}

```

The Fortran version takes only IERROR.

Conforming implementations of MPI are required to allow applications to pass NULL for both the `argc` and `argv` arguments of `main` in C.

Failures may disrupt the execution of the program before or during MPI initialization. A high-quality implementation shall not deadlock during MPI initialization, even in the presence of failures. Except for functions with the `MPI_T_` prefix, failures in MPI operations prior to or during MPI initialization are reported by invoking the initial error handler. Users can use the "mpi_initial_errhandler" info key during the launch of MPI processes (e.g., `MPI_COMM_SPAWN` / `MPI_COMM_SPAWN_MULTIPLE`, or `mpixec`) to set a nonfatal initial error handler before MPI initialization. When the initial error handler is set to `MPI_ERRORS_ABORT`, raising an error before or during initialization aborts the local MPI process (i.e., it is similar to calling `MPI_ABORT` on `MPI_COMM_SELF`). An implementation may not always be capable of determining, before MPI initialization, what constitutes the local MPI process, or the set of connected processes. In this case, errors before initialization may cause a different set of MPI processes to abort than specified. During MPI initialization, the initial error handler is associated with `MPI_COMM_WORLD`, `MPI_COMM_SELF`, and the communicator returned by `MPI_COMM_GET_PARENT` (if any).

Advice to implementors. Some failures may leave MPI in an undefined state, or raise an error before the error handling capabilities are fully operational, in which cases the implementation may be incapable of providing the desired error handling behavior. Of note, in some implementations, the notion of an MPI process is not clearly established in the early stages of MPI initialization (for example, when the implementation considers threads that called `MPI_INIT` as independent MPI processes); in this case, before MPI is initialized, the `MPI_ERRORS_ABORT` error handler may abort what would have become multiple MPI processes.

When a failure occurs during MPI initialization, the implementation may decide to return `MPI_SUCCESS` from the MPI initialization function instead of raising an error. It is recommended that an implementation masks an initialization error only when it expects that later MPI calls will result in well-specified behavior (i.e., barring additional failures, either the outcome of any call will be correct, or the call will raise an appropriate error). For example, it may be difficult for an implementation to avoid unspecified behavior when the group of `MPI_COMM_WORLD` does not contain the same set of MPI processes at all members of the communicator, or if the communicator returned from `MPI_COMM_GET_PARENT` was not initialized correctly. (*End of advice to implementors.*)

After MPI is initialized, the application can access information about the execution 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.

"mpi_initial_errhandler": Name of the initial errhandler.

"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".

Advice to users. If one of the "argv" arguments contains a space, there is no way to tell from the value of the "argv" info key whether a space is part of the argument or is separating different arguments. (*End of advice to users.*)

The info object `MPI_INFO_ENV` need not contain a (key,value) pair for each of these predefined keys; the set of (key,value) pairs provided is implementation-dependent. Implementations may provide additional, implementation specific, (key,value) pairs.

In cases where the MPI processes were started with `MPI_COMM_SPAWN_MULTIPLE` or, equivalently, with a startup mechanism that supports multiple process specifications, then the values stored in the info object `MPI_INFO_ENV` at a process are those values that affect the local MPI process.

Example 11.2. If MPI is started with a call to

```
mpiexec -n 5 -arch x86_64 ocean : -n 10 -arch power9 atmos
```

Then the first 5 processes will have in their `MPI_INFO_ENV` object the pairs (command, ocean), (maxprocs, 5), and (arch, x86_64). The next 10 processes will have in `MPI_INFO_ENV` (command, atmos), (maxprocs, 10), and (arch, power9)

Advice to users. The values passed in `MPI_INFO_ENV` are the values of the arguments passed to the mechanism that started the MPI execution—not the actual value provided. Thus, the value associated with "maxprocs" is the number of MPI processes requested; it can be larger than the actual number of processes obtained, if the soft option was used. (*End of advice to users.*)

Advice to implementors. High-quality implementations will provide a (key,value) pair for each parameter that can be passed to the command that starts an MPI program. (*End of advice to implementors.*)

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

MPI_INIT_THREAD(required, provided)

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)
```

Fortran 2008 binding

```
MPI_Init_thread(required, provided, ierror)
  INTEGER, INTENT(IN) :: required
  INTEGER, INTENT(OUT) :: provided
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)
  INTEGER REQUIRED, PROVIDED, IERROR
```

This call initializes MPI in the same way that a call to MPI_INIT would. In addition, it initializes the thread environment. The argument `required` is used to specify the desired level of thread support. The possible values are listed in increasing order of thread support.

MPI_THREAD_SINGLE: Only one thread will execute.

MPI_THREAD_FUNNELED: The process may be multithreaded, but the application must ensure that only the main thread makes MPI calls (for the definition of main thread, see `MPI_IS_THREAD_MAIN` on page 467).

MPI_THREAD_SERIALIZED: The process may be multithreaded, and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are “serialized”).

MPI_THREAD_MULTIPLE: Multiple threads may call MPI, with no restrictions.

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

1 that provided > required (thus providing a stronger level of support than required by the
2 user). Finally, if the user requirement cannot be satisfied, then the call will return in
3 provided the highest supported level.

4 A **thread compliant** MPI implementation will be able to return provided
5 = MPI_THREAD_MULTIPLE. Such an implementation may always return provided
6 = MPI_THREAD_MULTIPLE, irrespective of the value of required.

7 An MPI library that is not thread compliant must always return
8 provided = MPI_THREAD_SINGLE, even if MPI_INIT_THREAD is called on a multithreaded
9 process. The library should also return correct values for the MPI calls that can be executed
10 before initialization, even if multiple threads have been spawned.

11
12 *Rationale.* Such code is erroneous, but if the MPI initialization is performed by a
13 library, the error cannot be detected until MPI_INIT_THREAD is called. The require-
14 ments in the previous paragraph ensure that the error can be properly detected. (*End*
15 *of rationale.*)

16
17 A call to MPI_INIT has the same effect as a call to MPI_INIT_THREAD with a required
18 = MPI_THREAD_SINGLE.

19 Vendors may provide (implementation dependent) means to specify the level(s) of
20 thread support available when the MPI program is started, e.g., with arguments to mpiexec.
21 This will affect the outcome of calls to MPI_INIT and MPI_INIT_THREAD. Suppose, for ex-
22 ample, that an MPI program has been started so that only MPI_THREAD_MULTIPLE is avail-
23 able. Then MPI_INIT_THREAD will return provided = MPI_THREAD_MULTIPLE, irrespective
24 of the value of required; a call to MPI_INIT will also initialize the MPI thread support level
25 to MPI_THREAD_MULTIPLE. Suppose, instead, that an MPI program has been started so
26 that all four levels of thread support are available. Then, a call to MPI_INIT_THREAD will
27 return provided = required; alternatively, a call to MPI_INIT will initialize the MPI thread
28 support level to MPI_THREAD_SINGLE.

29
30 *Rationale.* Various optimizations are possible when MPI code is executed single-
31 threaded, or is executed on multiple threads, but not concurrently: mutual exclusion
32 code may be omitted. Furthermore, if only one thread executes, then the MPI library
33 can use library functions that are not thread safe, without risking conflicts with user
34 threads. Also, the model of one communication thread, multiple computation threads
35 fits many applications well, e.g., if the process code is a sequential Fortran/C program
36 with MPI calls that has been parallelized by a compiler for execution on an SMP node,
37 in a cluster of SMPs, then the process computation is multithreaded, but MPI calls
38 will likely execute on a single thread.

39 The design accommodates a static specification of the thread support level, for en-
40 vironments that require static binding of libraries, and for compatibility for current
41 multithreaded MPI codes. (*End of rationale.*)

42
43 *Advice to implementors.* If provided is not MPI_THREAD_SINGLE then the MPI library
44 should not invoke C or Fortran library calls that are not thread safe, e.g., in an
45 environment where malloc is not thread safe, then malloc should not be used by the
46 MPI library.

47 Some implementors may want to use different MPI libraries for different levels of thread
48 support. They can do so using dynamic linking and selecting which library will be

linked when `MPI_INIT_THREAD` is invoked. If this is not possible, then optimizations for lower levels of thread support will occur only when the level of thread support required is specified at link time.

Note that `required` need not be the same value on all processes of `MPI_COMM_WORLD`.
(*End of advice to implementors.*)

As with `MPI_INIT`, discussed in Section 11.2.1, the version for ISO C accepts the `argc` and `argv` that are provided by the arguments to `main` or `NULL` for both arguments.

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

`MPI_QUERY_THREAD(provided)`

OUT `provided` `provided` level of thread support (integer)

C binding

```
int MPI_Query_thread(int *provided)
```

Fortran 2008 binding

```
MPI_Query_thread(provided, ierror)
  INTEGER, INTENT(OUT) :: provided
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_QUERY_THREAD(PROVIDED, IERROR)
  INTEGER PROVIDED, IERROR
```

The call returns in `provided` the current level of thread support, which will be the value returned in `provided` by `MPI_INIT_THREAD`, if MPI was initialized by a call to `MPI_INIT_THREAD()`. This function is only applicable when using the World Model to initialize MPI. In the case of applications using both the World Model and the Sessions Model, this function only returns the thread support level returned in `provided` by `MPI_INIT_THREAD`.

`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)
```

Fortran 2008 binding

```
MPI_Is_thread_main(flag, ierror)
  LOGICAL, INTENT(OUT) :: flag
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran 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`). This function is only applicable when using the World Model to initialize MPI. In the case of applications using both the World Model and the Sessions Model, the behavior of this procedure is the same as if the application were only using the World Model.

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 `MPI_COMM_WORLD` and `MPI_COMM_SELF` cannot be used until the World Model is active, i.e., until `MPI_INIT_THREAD` is invoked by one thread (which, thereby, becomes the main thread). In particular, it is possible to enter the MPI execution with a multithreaded process.

In the World Model, 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. (*End of advice to users.*)

11.2.2 Finalizing MPI

`MPI_FINALIZE()`

C binding

```
int MPI_Finalize(void)
```

Fortran 2008 binding

```
MPI_Finalize(ierr)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierr
```

Fortran binding

```
MPI_FINALIZE(IERROR)
    INTEGER IERROR
```

This routine cleans up all MPI state associated with the World Model. If an MPI program terminates normally (i.e., not due to a call to `MPI_ABORT` or an unrecoverable error) then each process must call `MPI_FINALIZE` before it exits.

Before an MPI process invokes `MPI_FINALIZE`, the process must perform all MPI calls needed to complete its involvement in MPI communications associated with the World

Model. It must locally complete all MPI operations that it initiated and must execute matching calls needed to complete MPI communications initiated by other processes. For example, if the process executed a nonblocking send, it must eventually call `MPI_WAIT`, `MPI_TEST`, `MPI_REQUEST_FREE`, or any derived function; if the process is the target of a send, then it must post the matching receive; if it is part of a group executing a collective operation, then it must have completed its participation in the operation.

The call to `MPI_FINALIZE` does not clean up MPI state associated with objects created using `MPI_SESSION_INIT` and other Sessions Model methods, nor objects created using the communicator returned by `MPI_COMM_GET_PARENT`. See Sections 11.3 and 11.8.

The call to `MPI_FINALIZE` does not free objects created by MPI calls; these objects are freed using `MPI_XXX_FREE` calls.

`MPI_FINALIZE` is collective over all connected processes. If no processes were spawned, accepted or connected then this means over `MPI_COMM_WORLD`; otherwise it is collective over the union of all processes that have been and continue to be connected, as explained in Section 11.10.4.

The following examples illustrate these rules.

Example 11.3. The following code is correct

Process 0	Process 1
<code>MPI_Init();</code>	<code>MPI_Init();</code>
<code>MPI_Send(dest=1);</code>	<code>MPI_Recv(src=0);</code>
<code>MPI_Finalize();</code>	<code>MPI_Finalize();</code>

Example 11.4. Without a matching receive, the program is erroneous

Process 0	Process 1
<code>MPI_Init();</code>	<code>MPI_Init();</code>
<code>MPI_Send(dest=1);</code>	
<code>MPI_Finalize();</code>	<code>MPI_Finalize();</code>

Example 11.5. This program is correct: Process 0 calls `MPI_Finalize` after it has executed the MPI calls that complete the send operation. Likewise, process 1 executes the MPI call that completes the matching receive operation before it calls `MPI_Finalize`.

Process 0	Process 1
<code>MPI_Init();</code>	<code>MPI_Init();</code>
<code>MPI_Isend(dest=1);</code>	<code>MPI_Recv(src=0);</code>
<code>MPI_Request_free();</code>	<code>MPI_Finalize();</code>
<code>MPI_Finalize();</code>	<code>exit();</code>
<code>exit();</code>	

Example 11.6. 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
<code>MPI_Init();</code>	<code>MPI_Init();</code>
<code>buffer = malloc(1000000);</code>	<code>MPI_Recv(src=0);</code>
<code>MPI_Buffer_attach();</code>	<code>MPI_Finalize();</code>

```

1      MPI_Send(dest=1);           exit();
2      MPI_Finalize();
3      free(buffer);
4      exit();

```

Example 11.7. 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
<code>MPI_Issend(dest=1);</code>	<code>MPI_Finalize();</code>
<code>MPI_Cancel();</code>	
<code>MPI_Wait();</code>	
<code>MPI_Finalize();</code>	

Advice to implementors. Even though a process has executed all MPI calls needed to complete the communications it is involved with, such communication may not yet be completed from the viewpoint of the underlying MPI system. For example, a blocking send may have returned, even though the data is still buffered at the sender in an MPI buffer; an MPI process may receive a cancel request for a message it has completed receiving. The MPI implementation must ensure that a process has completed any involvement in MPI communication before `MPI_FINALIZE` returns. Thus, if a process exits after the call to `MPI_FINALIZE`, this will not cause an ongoing communication to fail. The MPI implementation should also complete freeing all objects marked for deletion by MPI calls that freed them. (*End of advice to implementors.*)

Failures may disrupt MPI operations during and after MPI finalization. A high quality implementation shall not deadlock in MPI finalization, even in the presence of failures. The normal rules for MPI error handling continue to apply. After `MPI_COMM_SELF` has been “freed” (see Section 11.2.4), errors that are not associated with a communicator, window, or file raise the initial error handler (set during the launch operation, see 11.8.4).

Although it is not required that all processes return from `MPI_FINALIZE`, it is required that, when it has not failed or aborted, at least the MPI process that was assigned rank 0 in `MPI_COMM_WORLD` returns, so that users can know that the MPI portion of the computation is over. In addition, in a POSIX environment, users may desire to supply an exit code for each process that returns from `MPI_FINALIZE`.

Note that a failure may terminate the MPI process that was assigned rank 0 in `MPI_COMM_WORLD`, in which case it is possible that no MPI process returns from `MPI_FINALIZE`.

Advice to users. Applications that handle errors are encouraged to implement all rank-specific code before the call to `MPI_FINALIZE`. In Example 11.8, the process with rank 0 in `MPI_COMM_WORLD` may have been terminated before, during, or after the call to `MPI_FINALIZE`, possibly leading to the code after `MPI_FINALIZE` never being executed. (*End of advice to users.*)

Example 11.8. The following illustrates the use of requiring that at least one process return and that it be known that process 0 is one of the processes that return. One wants code like the following to work no matter how many processes return.

```

...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
...
MPI_Finalize();
if (myrank == 0) {
    resultfile = fopen("outfile", "w");
    dump_results(resultfile);
    fclose(resultfile);
}
exit(0);

```

11.2.3 Determining Whether MPI Has Been Initialized When Using the World Model

One of the goals of MPI is to allow for layered libraries. A library using the World Model needs to know if MPI has been initialized using either of `MPI_INIT` or `MPI_INIT_THREAD`. In MPI the function `MPI_INITIALIZED` is provided to tell if MPI had been initialized using the World Model. In the World Model, once MPI has been finalized it cannot be restarted. A library needs to be able to determine this to act accordingly. To achieve this, the function `MPI_FINALIZED` is needed.

`MPI_INITIALIZED(flag)`

OUT flag Flag is true if `MPI_INIT` has been called and false otherwise (logical)

C binding

`int MPI_Initialized(int *flag)`

Fortran 2008 binding

`MPI_Initialized(flag, ierror)`
 LOGICAL, INTENT(OUT) :: flag
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

`MPI_INITIALIZED(FLAG, IERROR)`
 LOGICAL FLAG
 INTEGER IERROR

This routine may be used to determine whether `MPI_INIT` or `MPI_INIT_THREAD` has been called. `MPI_INITIALIZED` returns true if the calling process has called either of these MPI procedures. Whether `MPI_FINALIZE` has been called does not affect the behavior of `MPI_INITIALIZED`. This function must always be thread-safe, as defined in Section 11.6. This function returns false for applications using the Sessions Model exclusively.

```

1 MPI_FINALIZED(flag)
2     OUT     flag                true if MPI was finalized (logical)
3

```

C binding

```

5 int MPI_Finalized(int *flag)
6

```

Fortran 2008 binding

```

8 MPI_Finalized(flag, ierror)
9     LOGICAL, INTENT(OUT) :: flag
10    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11

```

Fortran binding

```

12 MPI_FINALIZED(FLAG, IERROR)
13     LOGICAL FLAG
14     INTEGER IERROR
15

```

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 11.6.

11.2.4 Allowing User Functions at MPI Finalization

In the context of the World Model, there are times in which it would be convenient to have actions happen when an MPI process finalizes MPI. For example, a routine may do initializations that are useful until the MPI job (or that part of the job that is being terminated in the case of dynamically created processes) finalizes MPI. This can be accomplished in MPI by attaching an attribute to MPI_COMM_SELF with a callback function. When MPI_FINALIZE is called, it will first execute the equivalent of an MPI_COMM_FREE on MPI_COMM_SELF. This will cause the delete callback function to be executed on all keys associated with MPI_COMM_SELF, in the reverse order that they were set on MPI_COMM_SELF. If no key has been attached to MPI_COMM_SELF, then no callback is invoked. The “freeing” of MPI_COMM_SELF occurs before any other parts of MPI are affected. Thus, for example, calling MPI_FINALIZED will return false in any of these callback functions. Once done with MPI_COMM_SELF, the order and rest of the actions taken by MPI_FINALIZE is not specified.

Advice to implementors. Since attributes can be added from any supported language, the MPI implementation needs to remember the creating language so the correct callback is made. Implementations that use the attribute delete callback on MPI_COMM_SELF internally should register their internal callbacks before returning from MPI_INIT / MPI_INIT_THREAD, so that libraries or applications will not have portions of the MPI implementation shut down before the application-level callbacks are made. (*End of advice to implementors.*)

11.3 The Sessions Model

There are a number of limitations with the World Model described in the preceding section. Among these are the following: MPI cannot be initialized from different application components without *a priori* knowledge or coordination; MPI cannot be initialized more than once; and MPI cannot be reinitialized after MPI_FINALIZE has been called. This section

describes an alternative approach to MPI initialization—the Sessions Model. With this approach, an MPI application, or components of the application, can instantiate MPI resources for the specific communication needs of this component. `MPI_COMM_WORLD` is not valid for use as a communicator. `MPI_INFO_ENV` is not valid for use as an info object when only using the Sessions Model. As described in Section 11.2.1, MPI must be initialized using the World Model to use this info object. Note that an application may employ both the Sessions Model and World Model concurrently (see Section 11.1).

In the Sessions Model, MPI resources can be allocated and freed multiple times in an MPI process.

As shown in Figure 11.1, when using the Sessions Model, an MPI process instantiates an **MPI Session handle**, which can be used to query the runtime system about characteristics of the job within which the process is running, as well as other system resources. Using this information, the MPI process can then create an MPI Group based on application requirements and available resources, which in turn can be used to create an MPI Communicator, Window, or File. By judicious creation of communicators, an application only needs to allocate MPI resources based on its communication requirements. Although there are existing MPI interfaces for creating communicators that can, in principle, allow for resource optimizations within an MPI implementation, this can only be done following initialization of MPI.

For multithreaded applications, the Sessions Model provides fine-grain control of the thread support level for MPI objects. It is possible to specify different thread support levels when creating different *MPI Session handles*. Thus different components of an application can use different thread support levels.

The Sessions Model introduces a concept of isolation. MPI objects derived from different *MPI Session handles* shall not be intermixed with each other in a single MPI procedure call. MPI objects derived from the Sessions Model shall not be intermixed in a single MPI procedure call with MPI objects derived from the World Model. MPI objects derived from the Sessions Model shall not be intermixed in a single MPI procedure call with MPI objects derived from the communicator obtained from a call to `MPI_COMM_GET_PARENT` or `MPI_COMM_JOIN`.

This restriction does not apply to generalized requests (Section 13.2) as such requests are not associated directly with communicators or other MPI objects. Note however, the Sessions Model does not otherwise change the semantics or behavior of MPI objects.

11.3.1 Session Creation and Destruction Methods

`MPI_SESSION_INIT`(info, errhandler, session)

IN	info	info object to specify thread support level and MPI implementation specific resources (handle)
IN	errhandler	error handler to invoke in the event that an error is encountered during this function call (handle)
OUT	session	new session (handle)

C binding

`int MPI_Session_init(MPI_Info info, MPI_Errhandler errhandler,`

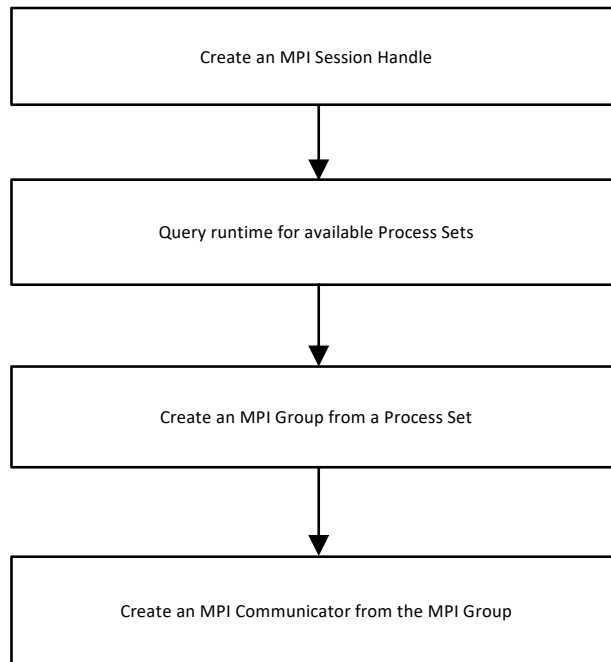


Figure 11.1: Steps to creating an MPI Communicator from an MPI Session handle.

```
MPI_Session *session)
```

Fortran 2008 binding

```
MPI_Session_init(info, errhandler, session, ierror)
  TYPE(MPI_Info), INTENT(IN) :: info
  TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
  TYPE(MPI_Session), INTENT(OUT) :: session
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_SESSION_INIT(INFO, ERRHANDLER, SESSION, IERROR)
  INTEGER INFO, ERRHANDLER, SESSION, IERROR
```

The info argument is used to request MPI functionality requirements and possible MPI implementation specific capabilities. The following info key is predefined:

"thread_level" used to request the thread support level required for MPI objects derived from the Session. Allowed values are "MPI_THREAD_SINGLE", "MPI_THREAD_FUNNELED", "MPI_THREAD_SERIALIZED", and "MPI_THREAD_MULTIPLE". Note that the thread support value is specified by a string rather than the integer values supplied to MPI_INIT_THREAD. The thread support level actually provided by the MPI implementation can be determined via a subsequent call to MPI_SESSION_GET_INFO to

return the info object associated with the Session. The default thread support level is MPI implementation dependent.

The errhandler argument specifies an error handler to invoke in the event that the Session instantiation call encounters an error. The error handler shall be either a pre-defined error handler (see 9.3) or one created using MPI_SESSION_CREATE_ERRHANDLER. Session instantiation is intended to be a lightweight operation. An MPI process may instantiate multiple Sessions. MPI_SESSION_INIT is always thread safe; multiple threads within an application may invoke it concurrently.

Advice to users. Requesting “MPI_THREAD_SINGLE” thread support level is generally not recommended, because this will conflict with other components of an application requesting higher levels of thread support. (*End of advice to users.*)

Advice to implementors. Owing to the restrictions of the MPI_THREAD_SINGLE thread support level, implementors are discouraged from making this the default thread support level for Sessions. (*End of advice to implementors.*)

MPI_SESSION_FINALIZE(session)

INOUT session session to be finalized (handle)

C binding

```
int MPI_Session_finalize(MPI_Session *session)
```

Fortran 2008 binding

```
MPI_Session_finalize(session, ierror)
  TYPE(MPI_Session), INTENT(INOUT) :: session
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_SESSION_FINALIZE(SESSION, IERROR)
  INTEGER SESSION, IERROR
```

This routine cleans up all MPI state associated with the supplied session. Every instantiated Session must be finalized using MPI_SESSION_FINALIZE. The handle session is set to MPI_SESSION_NULL by the call.

Before an MPI process invokes MPI_SESSION_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 it must execute matching calls needed to complete MPI communications initiated by other processes.

The call to MPI_SESSION_FINALIZE does not free objects created by MPI calls; these objects are freed using MPI_XXX_FREE calls.

MPI_SESSION_FINALIZE may be synchronizing on any or all of the groups associated with communicators, windows, or files derived from the session and not disconnected, freed, or closed, respectively, before the call to the MPI_SESSION_FINALIZE procedure. MPI_SESSION_FINALIZE behaves as if all such synchronizations occur concurrently. As MPI_COMM_FREE may mark a communicator for freeing later, MPI_SESSION_FINALIZE may be synchronizing on the group associated with a communicator that is only freed (with MPI_COMM_FREE) rather than disconnected (with MPI_COMM_DISCONNECT).

Rationale. This rule is similar to the rule that `MPI_FINALIZE` is collective (see 11.2.2), but does not require that `MPI_SESSION_FINALIZE` be collective over all connected MPI processes. It also allows for cases where some MPI processes may have derived a set of communicators using a different number of session handles. See Example 11.9. (*End of rationale.*)

Advice to implementors. This rule also allows for the completion of communications the MPI process is involved with that may not yet be completed from the viewpoint of the underlying MPI system. See the advice to implementors at the end of Section 11.2.2. (*End of advice to implementors.*)

Advice to implementors. An MPI implementation should be able to implement the semantics of `MPI_SESSION_FINALIZE` as a *local* procedure, provided an application frees all MPI windows, closes all MPI files, and uses `MPI_COMM_DISCONNECT` to free all MPI communicators associated with a session prior to invoking `MPI_SESSION_FINALIZE` on the corresponding session handle. (*End of advice to implementors.*)

Example 11.9. Three MPI processes are connected with 2 communicators (indicated by the = symbols), derived from one session handle in process X but from two separate session handles in both process Y and Z.

process-X	process-Y	process-Z	Remarks
			sesX, sesYA, ses YB, sesZA and sesZB are session handles.
(sesX)=====	(sesYA)=====	(sesZA)	communicator_1 and communicator_2 are derived from them.
(sesX)=====	(sesYB)=====	(sesZB)	
SF(sesX)	SF(sesYA)	SF(sesZA)	SF = MPI_SESSION_FINALIZE
	SF(sesYB)	SF(sesZB)	

Process X has only to finalize its one session handle, whereas the other two MPI processes have to call `MPI_SESSION_FINALIZE` twice in the same sequence with respect to the communicators derived from the session handles. Specifically, both process Y and process Z shall call `MPI_SESSION_FINALIZE` for the session from which `communicator_1` was derived before calling the `MPI_SESSION_FINALIZE` for the session from which `communicator_2` was derived, or vice versa (i.e., both shall finalize the session for `communicator_2` first then finalize the session for `communicator_1`). The call `SF(ses)` in process X may not return until both `SF(ses*A)` and `SF(ses*B)` are called in processes Y and Z.

11.3.2 Processes Sets

Process sets are the mechanism for MPI applications to query the runtime. Process sets are identified by process set names. Process set names have a *Uniform Resource Identifier* (URI) format. Two process set names are mandated: `"mpi://WORLD"` and `"mpi://SELF"`. Additional process set names may be defined, for example, `"mpix://UNIVERSE"` and `"hwloc://L3Cache"` may be defined by the MPI implementation. The `"mpi://"` namespace is reserved for exclusive use by the MPI standard. Figure 11.2 depicts process sets that the

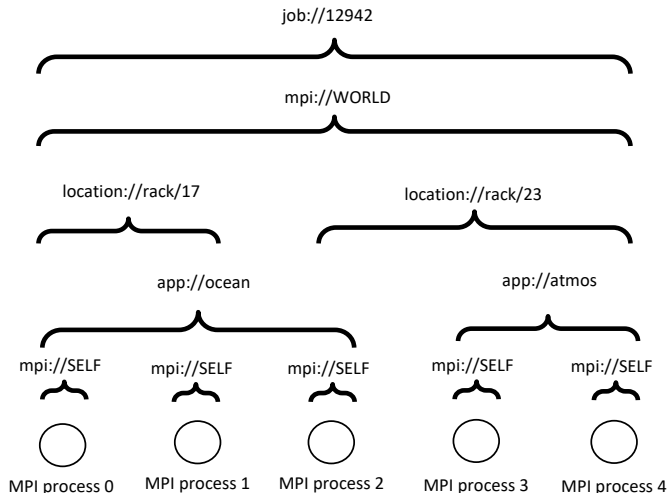


Figure 11.2: Examples of process sets. Illustrated are the two mandated process sets—`"mpi://WORLD"` and `"mpi://SELF"`—along with several optional ones that a runtime could define. In this example, `MPI_SESSION_GET_NUM_PSETS` would return five at each MPI process.

runtime could associate with an instance of an MPI job. In this example, the two mandated process sets are defined, in addition to optional, implementation specific ones.

Mechanisms for defining process sets and how system resources are assigned to these sets is considered to be implementation dependent.

A process set caches key/value tuples that are accessible to the application via an `MPI_Info` object. The `"mpi_size"` key is mandatory for all process sets.

11.3.3 Runtime Query Functions

`MPI_SESSION_GET_NUM_PSETS(session, info, npset_names)`

IN	session	session (handle)
IN	info	info object (handle)
OUT	npset_names	number of available process sets (non-negative integer)

C binding

```
int MPI_Session_get_num_psets(MPI_Session session, MPI_Info info,
                             int *npset_names)
```

Fortran 2008 binding

```
MPI_Session_get_num_psets(session, info, npset_names, ierror)
```

```

1     TYPE(MPI_Session), INTENT(IN) :: session
2     TYPE(MPI_Info), INTENT(IN) :: info
3     INTEGER, INTENT(OUT) :: npset_names
4     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

6     MPI_SESSION_GET_NUM_PSETS(SESSION, INFO, NPSET_NAMES, IERROR)
7     INTEGER SESSION, INFO, NPSET_NAMES, IERROR

```

This function is used to query the runtime for the number of available process sets in which the calling MPI process is a member. An MPI implementation is allowed to increase the number of available process sets during the execution of an MPI application when new process sets become available. However, MPI implementations are not allowed to change the index of a particular process set name, or to change the name of the process set at a particular index, or to delete a process set name once it has been added. When a process set becomes invalid, for example, when some processes become unreachable due to failures in the communication system, subsequent usage of the process set name should raise an error. For example, creating an MPI_Group from such a process set might succeed because it is a local operation, but creating an MPI_Comm from that group and attempting collective communication should raise an error.

Advice to implementors. It is anticipated that an MPI implementation may be relying on an external runtime system to provide process sets. Such runtime systems may have the ability to dynamically create process sets during the course of application execution. Requiring the number of process sets returned by MPI_SESSION_GET_NUM_PSETS to be constant over the course of application execution would prevent an application from taking advantage of such capabilities. (*End of advice to implementors.*)

```

30     MPI_SESSION_GET_NTH_PSET(session, info, n, pset_len, pset_name)

```

32	IN	session	session (handle)
33	IN	info	info object (handle)
34	IN	n	index of the desired process set name (integer)
35	INOUT	pset_len	length of the pset_name argument (integer)
36	OUT	pset_name	name of the nth process set (string)

C binding

```

40     int MPI_Session_get_nth_pset(MPI_Session session, MPI_Info info, int n,
41                               int *pset_len, char *pset_name)

```

Fortran 2008 binding

```

43     MPI_Session_get_nth_pset(session, info, n, pset_len, pset_name, ierror)
44     TYPE(MPI_Session), INTENT(IN) :: session
45     TYPE(MPI_Info), INTENT(IN) :: info
46     INTEGER, INTENT(IN) :: n
47     INTEGER, INTENT(INOUT) :: pset_len

```

```

CHARACTER(LEN=*), INTENT(OUT) :: pset_name
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

MPI_SESSION_GET_NTH_PSET(SESSION, INFO, N, PSET_LEN, PSET_NAME, IERROR)
  INTEGER SESSION, INFO, N, PSET_LEN, IERROR
  CHARACTER*(*) PSET_NAME

```

This function returns the name of the *n*th process set in the supplied *pset_name* buffer. *pset_len* is the size of the buffer needed to store the *n*th process set name. If the *pset_len* passed into the function is less than the actual buffer size needed for the process set name, then the string value returned in *pset_name* is truncated. If *pset_len* is set to 0, *pset_name* is not changed. On return, the value of *pset_len* will be set to the required buffer size to hold the process set name. In C, *pset_len* includes the required space for the null terminator. In C, this function returns a null terminated string in all cases where the *pset_len* input value is greater than 0.

If two MPI processes get the same process set name, then the intersection of the two process sets shall either be the empty set or identical to the union of the two process sets.

After a successful call to `MPI_SESSION_GET_NTH_PSET`, subsequent calls to routines that query information about the same process set name and same session handle must return the same information. An MPI implementation is not allowed to alter any of the returned process set names.

Process set names have an implementation-defined maximum length of `MPI_MAX_PSET_NAME_LEN` characters. `MPI_MAX_PSET_NAME_LEN` shall have a value of at least 63.

Advice to users. `MPI_MAX_PSET_NAME_LEN` might be very large, so it might not be wise to declare a string of that size. Users are encouraged to use `MPI_SESSION_GET_NTH_PSET` both for obtaining the length of a *pset_name* and the process set name. (*End of advice to users.*)

```

MPI_SESSION_GET_INFO(session, info_used)

```

```

IN      session          session (handle)
OUT     info_used       see explanation below (handle)

```

C binding

```

int MPI_Session_get_info(MPI_Session session, MPI_Info *info_used)

```

Fortran 2008 binding

```

MPI_Session_get_info(session, info_used, ierror)
  TYPE(MPI_Session), INTENT(IN) :: session
  TYPE(MPI_Info), INTENT(OUT) :: info_used
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

MPI_SESSION_GET_INFO(SESSION, INFO_USED, IERROR)
  INTEGER SESSION, INFO_USED, IERROR

```

1 MPI_SESSION_GET_INFO returns a new info object containing the hints of the MPI
 2 Session associated with session. The current setting of all hints related to this MPI Session
 3 is returned in info_used. An MPI implementation is required to return all hints that are
 4 supported by the implementation and have default values specified; any user-supplied hints
 5 that were not ignored by the implementation; and any additional hints that were set by
 6 the implementation. If no such hints exist, a handle to a newly created info object is
 7 returned that contains no key/value pair. The user is responsible for freeing info_used via
 8 MPI_INFO_FREE.

9
 10
 11 MPI_SESSION_GET_PSET_INFO(session, pset_name, info)

12	IN	session	session (handle)
13	IN	pset_name	name of process set (string)
14	OUT	info	info object containing information about the given
15			process set (handle)
16			

17 C binding

18 int MPI_Session_get_pset_info(MPI_Session session, const char *pset_name,
 19 MPI_Info *info)

20 Fortran 2008 binding

21 MPI_Session_get_pset_info(session, pset_name, info, ierror)
 22 TYPE(MPI_Session), INTENT(IN) :: session
 23 CHARACTER(LEN=*), INTENT(IN) :: pset_name
 24 TYPE(MPI_Info), INTENT(OUT) :: info
 25 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

26 Fortran binding

27 MPI_SESSION_GET_PSET_INFO(SESSION, PSET_NAME, INFO, IERROR)
 28 INTEGER SESSION, INFO, IERROR
 29 CHARACTER*(*) PSET_NAME

30
 31
 32 This function is used to query properties of a specific process set. The returned info
 33 object can be queried with existing MPI info object query functions. One key/value pair
 34 must be defined, "mpi_size". The value of the "mpi_size" key specifies the number of MPI
 35 processes in the process set. The user is responsible for freeing the returned MPI_Info object.

36 11.3.4 Sessions Model Examples

37
 38
 39 This section presents several examples of how to use MPI Sessions to create MPI Groups
 40 and MPI Communicators.

41 **Example 11.10.** Simple example illustrating creation of an MPI communicator using the
 42 Sessions Model.

```
43 #include <stdio.h>
44 #include <stdlib.h>
45 #include <string.h>
46 #include "mpi.h"
```



```
static MPI_Session lib_shandle = MPI_SESSION_NULL;
static MPI_Comm lib_comm = MPI_COMM_NULL;

int library_foo_init(void)
{
    int rc, flag, valuelen;
    int ret = 0;
    const char pset_name[] = "mpi://WORLD";
    const char mt_key[] = "thread_level";
    const char mt_value[] = "MPI_THREAD_MULTIPLE";
    char out_value[100]; /* large enough */
    MPI_Group wgroup = MPI_GROUP_NULL;
    MPI_Info sinfo = MPI_INFO_NULL;
    MPI_Info tinfo = MPI_INFO_NULL;

    MPI_Info_create(&sinfo);
    MPI_Info_set(sinfo, mt_key, mt_value);
    rc = MPI_Session_init(sinfo, MPI_ERRORS_RETURN,
                        &lib_shandle);
    if (rc != MPI_SUCCESS) {
        ret = -1;
        goto fn_exit;
    }

    /*
     * check we got thread support level foo library needs
     */
    rc = MPI_Session_get_info(lib_shandle, &tinfo);
    if (rc != MPI_SUCCESS) {
        ret = -1;
        goto fn_exit;
    }

    valuelen = sizeof(out_value);
    MPI_Info_get_string(tinfo, mt_key, &valuelen,
                      out_value, &flag);
    if (0 == flag) {
        printf("Could not find key %s\n", mt_key);
        ret = -1;
        goto fn_exit;
    }

    if (strcmp(out_value, mt_value)) {
        printf("Did not get thread multiple support, got %s\n",
              out_value);
        ret = -1;
        goto fn_exit;
    }

    /*
     * create a group from the WORLD process set
     */
    rc = MPI_Group_from_session_pset(lib_shandle,
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

```

```

        pset_name,
        &wgroup);

    if (rc != MPI_SUCCESS) {
        ret = -1;
        goto fn_exit;
    }

    /*
     * get a communicator
     */
    rc = MPI_Comm_create_from_group(wgroup,
                                    "org.mpi-forum.mpi-v4_0.example-ex11_8",
                                    MPI_INFO_NULL,
                                    MPI_ERRORS_RETURN,
                                    &lib_comm);

    if (rc != MPI_SUCCESS) {
        ret = -1;
        goto fn_exit;
    }

    /*
     * free group, library doesn't need it.
     */

fn_exit:
    MPI_Group_free(&wgroup);

    if (sinfo != MPI_INFO_NULL) {
        MPI_Info_free(&sinfo);
    }

    if (tinfo != MPI_INFO_NULL) {
        MPI_Info_free(&tinfo);
    }

    if (ret != 0) {
        MPI_Session_finalize(&lib_shandle);
    }

    return ret;
}

```

Example 11.10 shows how the pre-defined "mpi://WORLD" process set can be used to first create a local MPI group and then subsequently to create an MPI communicator from this group.

Example 11.11. This example illustrates the use of Process Set query functions to select a Process Set to use for MPI Group creation.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mpi.h"

```

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

int main(int argc, char *argv[])
{
    int i, n_psets, psetlen, rc, ret;
    int valuelen;
    int flag = 0;
    char *pset_name = NULL;
    char *info_val = NULL;
    MPI_Session shandle = MPI_SESSION_NULL;
    MPI_Info sinfo = MPI_INFO_NULL;
    MPI_Group pgroup = MPI_GROUP_NULL;

    if (argc < 2) {
        fprintf(stderr, "A process set name fragment is required\n");
        return EXIT_FAILURE;
    }

    rc = MPI_Session_init(MPI_INFO_NULL, MPI_ERRORS_RETURN, &shandle);
    if (rc != MPI_SUCCESS) {
        fprintf(stderr, "Could not initialize session, bailing out\n");
        return EXIT_FAILURE;
    }

    MPI_Session_get_num_psets(shandle, MPI_INFO_NULL, &n_psets);

    for (i=0, pset_name=NULL; i<n_psets; i++) {
        psetlen = 0;
        MPI_Session_get_nth_pset(shandle, MPI_INFO_NULL, i,
                                &psetlen, NULL);
        pset_name = (char *)malloc(sizeof(char) * psetlen);
        MPI_Session_get_nth_pset(shandle, MPI_INFO_NULL, i,
                                &psetlen, pset_name);
        if (strstr(pset_name, argv[1]) != NULL) break;

        free(pset_name);
        pset_name = NULL;
    }

    /*
     * get instance of an info object for this Session
     */

    MPI_Session_get_pset_info(shandle, pset_name, &sinfo);
    valuelen = 0;
    MPI_Info_get_string(sinfo, "mpi_size", &valuelen, NULL, &flag);
    if (flag) {
        info_val = (char *)malloc(valuelen);
        MPI_Info_get_string(sinfo, "mpi_size", &valuelen, info_val, &flag);
        free(info_val);
    }

    /*
     * create a group from the process set
     */

    rc = MPI_Group_from_session_pset(shandle, pset_name,
                                     &pgroup);

```

```

1  ret = (rc == MPI_SUCCESS) ? 0 : EXIT_FAILURE;
2
3  free(pset_name);
4  MPI_Group_free(&pgroup);
5  MPI_Info_free(&sinfo);
6  MPI_Session_finalize(&shandle);
7
8  fprintf(stderr, "Test completed ret = %d\n", ret);
9  return ret;
10 }

```

Example 11.11 illustrates several aspects of the Sessions Model. First, the default error handler can be specified when instantiating a Session instance. Second, there must be at least two process sets associated with a Session. Third, the example illustrates use of the Sessions info object and the one required key: "mpi_size".

Example 11.12. A Fortran 2008 example illustrating how to obtain information about available process sets, create an MPI Group from a process set, and subsequently create an MPI Communicator.

```

19 PROGRAM MAIN
20   USE mpi_f08
21   IMPLICIT NONE
22   INTEGER :: pset_len, ierror, n_psets
23   CHARACTER(LEN=:), ALLOCATABLE :: pset_name
24   TYPE(MPI_Session) :: shandle
25   TYPE(MPI_Group) :: pgroup
26   TYPE(MPI_Comm) :: pcomm
27
28   CALL MPI_Session_init(MPI_INFO_NULL, MPI_ERRORS_RETURN, &
29     shandle, ierror)
30   IF (ierror .NE. MPI_SUCCESS) THEN
31     WRITE(*,*) "MPI_Session_init failed"
32     ERROR STOP
33   END IF
34
35   CALL MPI_Session_get_num_psets(shandle, MPI_INFO_NULL, n_psets)
36   IF (n_psets .LT. 2) THEN
37     WRITE(*,*) "MPI_Session_get_num_psets didn't return at least 2 psets"
38     ERROR STOP
39   END IF
40
41   !
42   ! Just get the second pset's length and name
43   ! Note that index values are zero-based, even in Fortran
44   !
45
46   pset_len = 0
47   CALL MPI_Session_get_nth_pset(shandle, MPI_INFO_NULL, 1, &
48     pset_len, pset_name)
49   ALLOCATE(CHARACTER(LEN=pset_len)::pset_name)
50   CALL MPI_Session_get_nth_pset(shandle, MPI_INFO_NULL, 1, &
51     pset_len, pset_name)
52
53   !
54   ! create a group from the pset

```

```

!
CALL MPI_Group_from_session_pset(shandle, pset_name, pgroup)
!
! free the buffer used for the pset name
!
DEALLOCATE(pset_name)
!
! create a MPI communicator from the group
!
CALL MPI_Comm_create_from_group(pgroup, "session_example", &
                                MPI_INFO_NULL, &
                                MPI_ERRORS_RETURN, &
                                pcomm)

CALL MPI_Barrier(pcomm, ierror)
IF (ierror .NE. MPI_SUCCESS) THEN
    WRITE(*,*) "Barrier call on communicator failed"
    ERROR STOP
END IF

CALL MPI_Comm_free(pcomm)
CALL MPI_Group_free(pgroup)
CALL MPI_Session_finalize(shandle, ierror)

END PROGRAM MAIN

```

Note in this example that the call to `MPI_SESSION_FINALIZE` may block in order to ensure that the calling MPI process has completed its involvement in the preceding `MPI_BARRIER` operation. If `MPI_COMM_DISCONNECT` had been used instead of `MPI_COMM_FREE`, the example would have blocked in `MPI_COMM_DISCONNECT` rather than `MPI_SESSION_FINALIZE`.

11.4 Common Elements of Both Process Models

11.4.1 MPI Functionality that is Always Available

Some MPI functions may be invoked at any time, including prior to calling `MPI_INIT` or `MPI_SESSION_INIT`, and following MPI finalization, independent of whether the World Model, Sessions Model, or both are used. These functions can be called concurrently by multiple threads within an MPI Process. Table 11.1 lists the applicable MPI functions.

In addition to the functions listed in Table 11.1, any function with the prefix `MPI_T_` (within the constraints for functions with this prefix listed in Section 15.3.4) may also be called prior to MPI initialization and after MPI finalization.

11.4.2 Aborting MPI Processes

`MPI_ABORT(comm, errorcode)`

IN comm communicator of MPI processes to abort (handle)

Table 11.1: List of MPI Functions that can be called at any time within an MPI program, including prior to MPI initialization and following MPI finalization

MPI_INITIALIZED
MPI_FINALIZED
MPI_GET_VERSION
MPI_GET_LIBRARY_VERSION
MPI_INFO_CREATE
MPI_INFO_CREATE_ENV
MPI_INFO_SET
MPI_INFO_DELETE
MPI_INFO_GET_STRING
MPI_INFO_GET_NKEYS
MPI_INFO_GET_NTHKEY
MPI_INFO_DUP
MPI_INFO_FREE
MPI_INFO_F2C
MPI_INFO_C2F
MPI_SESSION_CREATE_ERRHANDLER
MPI_SESSION_CALL_ERRHANDLER
MPI_ERRHANDLER_FREE
MPI_ERRHANDLER_F2C
MPI_ERRHANDLER_C2F
MPI_ERROR_STRING
MPI_ERROR_CLASS

IN errorcode error code to return to invoking environment
(integer)

C binding

```
int MPI_Abort(MPI_Comm comm, int errorcode)
```

Fortran 2008 binding

```
MPI_Abort(comm, errorcode, ierror)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, INTENT(IN) :: errorcode
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_ABORT(COMM, ERRORCODE, IERROR)
  INTEGER COMM, ERRORCODE, IERROR
```

This routine makes a “best attempt” to abort all MPI processes in the group of comm. This function does not require that the invoking environment take any action with the error code. However, a Unix or POSIX environment should handle this as a return errorcode from the main program.

It may not be possible for an MPI implementation to abort only the processes 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. When using the World Model, and if no processes were spawned, accepted, or connected then this has the effect of aborting all the processes associated with `MPI_COMM_WORLD`. In the case of the Sessions Model, if an MPI process has instantiated multiple sessions, the union of the process sets in these sessions are considered connected processes. Thus invoking `MPI_ABORT` on a communicator derived from one of these sessions will result in all MPI processes in this union being aborted.

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 nonaborted 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 an appropriate error code when attempting communication with an aborted process (e.g., an error of class `MPI_ERR_PROC_ABORTED`). A high quality implementation should support equivalent behavior for communicators derived from sessions. (*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.*)

11.5 Portable MPI Process Startup

A number of implementations of MPI provide a startup command for MPI programs that is of the form

```
mpirun <mpirun arguments> <program> <program arguments>
```

Separating the command to start the program from the program itself provides flexibility, particularly for network and heterogeneous implementations. For example, the startup script need not run on one of the machines that will be executing the MPI program itself.

Having a standard startup mechanism also extends the portability of MPI programs one step further, to the command lines and scripts that manage them. For example, a validation suite script that runs hundreds of programs can be a portable script if it is written using such a standard startup mechanism. In order that the “standard” command not be confused with existing practice, which is not standard and not portable among implementations, instead of `mpirun MPI` specifies `mpiexec`.

While a standardized startup mechanism improves the usability of MPI, the range of environments is so diverse (e.g., there may not even be a command line interface) that MPI cannot mandate such a mechanism. Instead, MPI specifies an `mpiexec` startup command

and recommends but does not require it, as advice to implementors. However, if an implementation does provide a command called `mpiexec`, it must be of the form described below.

It is suggested that

```
mpiexec -n <numprocs> <program>
```

be at least one way to start `<program>` with an initial set of `<numprocs>` processes, which will be accessible as the process set named "mpi://WORLD" in the Sessions Model and/or used to form the group associated with the built-in communicator, `MPI_COMM_WORLD` in the World Model. Other arguments to `mpiexec` may be implementation-dependent.

Advice to implementors. Implementors, if they do provide a special startup command for MPI programs, are advised to give it the following form. The syntax is chosen in order that `mpiexec` be able to be viewed as a command-line version of `MPI_COMM_SPAWN` (See Section 11.8.4).

Analogous to `MPI_COMM_SPAWN`, we have

```
mpiexec -n          <maxprocs>
        -soft      <          >
        -host      <          >
        -arch      <          >
        -wdir      <          >
        -path      <          >
        -file      <          >
        -initial-errhandler <  >
        ...
        <command line>
```

for the case where a single command line for the application program and its arguments will suffice. See Section 11.8.4 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 11.13. Start 16 instances of `myprog` on the current or default machine:

```
mpiexec -n 16 myprog
```

Example 11.14. Start 10 instances of `myprog` on the machine called `ferrari`:

```
mpiexec -n 10 -host ferrari myprog
```

Example 11.15. Start 3 instances of the same program `myprog` with different command-line arguments:

```
mpiexec myprog infile1 : myprog infile2 : myprog infile3
```

Example 11.16. Start 5 instances of the `ocean` program on `x86_64` hosts and 10 instances of the `atmos` program on Power9 hosts (Form B):

```
mpiexec -n 5 -arch x86_64 ocean : -n 10 -arch power9 atmos
```

It is assumed that the implementation in this case has a method for choosing hosts of the appropriate type. Their ranks are in the order specified.

Example 11.17. Start the `ocean` program on five Suns and the `atmos` program on 10 RS/6000's (Form B):

```
mpiexec -configfile myfile
```

where `myfile` contains

```
-n 5 -arch sun ocean
-n 10 -arch rs6000 atmos
```

(End of advice to implementors.)

11.6 MPI and Threads

This section specifies the interaction between MPI calls and threads. Although thread compliance is not required, the standard specifies how threads are to work if they are provided. The section lists minimal requirements for **thread compliant** MPI implementations and defines functions that can be used for initializing the thread environment. MPI may be implemented in environments where threads are not supported or perform poorly. Therefore, MPI implementations are not required to be thread compliant as defined in this section. Regardless of whether or not the MPI implementation is thread compliant, a subset of MPI functions must always be thread safe. A complete list of such MPI functions is given in Table 11.1. When a thread is executing one of these routines, if another concurrently running thread also makes an MPI call, the outcome will be as if the calls executed in some order.

This section generally assumes a thread package similar to POSIX threads [44], but the syntax and semantics of thread calls are not specified here—these are beyond the scope

1 of this document.

3 11.6.1 General

4
5 In a thread-compliant implementation, an MPI process is a process that may be multi-
6 threaded. Each thread can issue MPI calls; however, threads are not separately addressable:
7 a rank in a send or receive call identifies a process, not a thread. A message sent to a process
8 can be received by any thread in this process.

9
10 *Rationale.* This model corresponds to the POSIX model of interprocess commu-
11 nication: the fact that a process is multithreaded, rather than single-threaded, does
12 not affect the external interface of this process. MPI implementations in which MPI
13 ‘processes’ are POSIX threads inside a single POSIX process are not thread-compliant
14 by this definition (indeed, their “processes” are single-threaded). (*End of rationale.*)

15
16 *Advice to users.* It is the user’s responsibility to prevent races when threads within
17 the same application post conflicting communication calls. The user can make sure
18 that two threads in the same process will not issue conflicting communication calls by
19 using distinct communicators at each thread. (*End of advice to users.*)

20 The two main requirements for a thread-compliant implementation are listed below.

- 21
22 1. All MPI calls are **thread-safe**, i.e., two concurrently running threads may make MPI
23 calls and the outcome will be as if the calls executed in some order, even if their
24 execution is interleaved.
- 25
26 2. Blocking MPI calls will block the calling thread only, allowing another thread to
27 execute, if available. The calling thread will be blocked until the event on which it
28 is waiting occurs. Once the blocked communication is enabled and can proceed, then
29 the call will complete and the thread will be marked runnable, within a finite time.
30 A blocked thread will not prevent *progress* of other runnable threads on the same
31 process, and will not prevent them from executing MPI calls.

32
33 **Example 11.18.** Process 0 consists of two threads. The first thread executes a blocking
34 send call `MPI_Send(buff1, count, type, 0, 0, comm)`, whereas the second thread executes
35 a blocking receive call `MPI_Recv(buff2, count, type, 0, 0, comm, &status)`, i.e., the first
36 thread sends a message that is received by the second thread. This communication should
37 always succeed. According to the first requirement, the execution will correspond to some
38 interleaving of the two calls. According to the second requirement, a call can only block
39 the calling thread and cannot prevent progress of the other thread. If the send call went
40 ahead of the receive call, then the sending thread may block, but this will not prevent
41 the receiving thread from executing. Thus, the receive call will occur. Once both calls
42 occur, the communication is enabled and both calls will complete. On the other hand, a
43 single-threaded process that posts a send, followed by a matching receive, may deadlock.
44 The progress requirement for multithreaded implementations is stronger, as a blocked call
45 cannot prevent progress in other threads.

46
47 *Advice to implementors.* MPI calls can be made thread-safe by executing only one at
48 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.*)

11.6.2 Clarifications

Initialization and Completion. When using the World Model, 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.*)

Threads and the Sessions Model. The Sessions Model provides a finer-grain approach to controlling the interaction between MPI calls and threads. When using this model, the desired level of thread support is specified at Session initialization time. See Section 11.3. Thus it is possible for communicators and other MPI objects derived from one Session to provide a different level of thread support than those created from another Session for which a different level of thread support was requested. Depending on the level of thread support requested at Session initialization time, different threads in a MPI process can make concurrent calls to MPI when using MPI objects derived from different *session handles*. Note that the requested and provided level of thread support when creating a Session may influence the granted level of thread support in a subsequent invocation of `MPI_SESSION_INIT`. Likewise, if the application at some point calls `MPI_INIT_THREAD`, the requested and granted level of thread support may influence the granted level of thread support for subsequent calls to `MPI_SESSION_INIT`. Similarly, if the application calls `MPI_INIT_THREAD` after a call to `MPI_SESSION_INIT`, the level of thread support returned from `MPI_INIT_THREAD` may be similarly influenced by the requested level of thread support in the prior call to `MPI_SESSION_INIT`.

In addition, if an MPI application is only using the Sessions Model, the provided thread support level returned by `MPI_QUERY_THREAD` is the same as that returned prior to invocation of `MPI_INIT_THREAD` or `MPI_INIT`. If the application also used the World Model in some component of the application, `MPI_QUERY_THREAD` will return the level of thread support returned by the original call to `MPI_INIT_THREAD`.

Multiple threads completing the same request. A program in which two threads block, waiting on the same request, is erroneous. Similarly, the same request cannot appear in the array of requests of two concurrent `MPI_{WAIT|TEST}{ANY|SOME|ALL}` calls. In MPI, a request can only be completed once. Any combination of wait or test that violates this rule is erroneous.

Rationale. This restriction is consistent with the view that a multithreaded execution corresponds to an interleaving of the MPI calls. In a single threaded implementation, once a wait is posted on a request the request handle will be nullified before it is

possible to post a second wait on the same handle. With threads, an `MPI_WAIT{ANY|SOME|ALL}` may be blocked without having nullified its request(s) so it becomes the user's responsibility to avoid using the same request in an `MPI_WAIT` on another thread. This constraint also simplifies implementation, as only one thread will be blocked on any communication or I/O event. (*End of rationale.*)

Probe. A receive call that uses source and tag values returned by a preceding call to `MPI_PROBE` or `MPI_IPROBE` will receive the message matched by the probe call only if there was no other matching receive after the probe and before that receive. In a multi-threaded environment, it is up to the user to enforce this condition using suitable mutual exclusion logic. This can be enforced by making sure that each communicator is used by only one thread on each process. Alternatively, `MPI_MPROBE` or `MPI_IMPROBE` can be used.

Collective calls. Matching of collective calls on a communicator, window, or file handle is done according to the order in which the calls are issued at each process. If concurrent threads issue such calls on the same communicator, window or file handle, it is up to the user to make sure the calls are correctly ordered, using interthread synchronization.

Advice to users. With three concurrent threads in each MPI process of a communicator `comm`, it is allowed that thread A in each MPI process calls a collective operation on `comm`, thread B calls a file operation on an existing file handle that was formerly opened on `comm`, and thread C invokes one-sided operations on an existing window handle that was also formerly created on `comm`. (*End of advice to users.*)

Rationale. As specified in `MPI_FILE_OPEN` and `MPI_WIN_CREATE`, a file handle and a window handle inherit only the group of processes of the underlying communicator, but not the communicator itself. Accesses to communicators, window handles and file handles cannot affect one another. (*End of rationale.*)

Advice to implementors. If the implementation of file or window operations internally uses MPI communication then a duplicated communicator may be cached on the file or window object. (*End of advice to implementors.*)

Error handlers. An error handler does not necessarily execute in the context of the thread that made the error-raising MPI call; the error 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 error handler to be executed on the thread where the error is raised. (*End of rationale.*)

Interaction with signals and cancellations. The outcome is undefined if a thread that executes an MPI call is cancelled (by another thread), or if a thread catches a signal while executing an MPI call. However, a thread of an MPI process may terminate, and may catch signals or be cancelled by another thread when not executing MPI calls.

Rationale. Few C library functions are signal safe, and many have cancellation points—points at which the thread executing them may be cancelled. The above restriction simplifies implementation (no need for the MPI library to be “async-cancel-safe” or “async-signal-safe”). (*End of rationale.*)

Advice to users. Users can catch signals in separate, non-MPI threads (e.g., by masking signals on MPI calling threads, and unmasking them in one or more non-MPI threads). A good programming practice is to have a distinct thread blocked in a call to `sigwait` for each user expected signal that may occur. Users must not catch signals used by the MPI implementation; as each MPI implementation is required to document the signals used internally, users can avoid these signals. (*End of advice to users.*)

Advice to implementors. The MPI library should not invoke library calls that are not thread safe, if multiple threads execute. (*End of advice to implementors.*)

11.7 The Dynamic Process Model

The dynamic process model allows for the creation and cooperative termination of processes after an MPI application has started. It provides a mechanism to establish communication between the newly created processes and the existing MPI application. It also provides a mechanism to establish communication between two existing MPI applications, even when one did not “start” the other.

The MPI procedures described in this section require the World Model, meaning that `MPI_INIT` or `MPI_INIT_THREAD` has been used to initialize MPI.

11.7.1 Starting Processes

MPI applications may start new processes through an interface to an external process manager.

`MPI_COMM_SPAWN` starts MPI processes and establishes communication with them, returning an inter-communicator. `MPI_COMM_SPAWN_MULTIPLE` starts several different binaries (or the same binary with different arguments), placing them in the same `MPI_COMM_WORLD` and returning an inter-communicator.

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

11.7.2 The Runtime Environment

The `MPI_COMM_SPAWN` and `MPI_COMM_SPAWN_MULTIPLE` routines provide an interface between MPI and the *runtime environment* of an MPI application. The difficulty is that there is an enormous range of runtime environments and application requirements, and MPI must not be tailored to any particular one.

MPI assumes, implicitly, the existence of an environment in which an application runs. It does not provide “operating system” services, such as a general ability to query what processes are running, to kill arbitrary processes, to find out properties of the runtime environment (how many processors, how much memory, etc.). Complex interaction of an MPI application with its runtime environment should be done through an environment-specific API.

1 At some low level, obviously, MPI must be able to interact with the runtime system,
2 but the interaction is not visible at the application level and the details of the interaction
3 are not specified by the MPI standard.

4 In many cases, it is impossible to keep environment-specific information out of the MPI
5 interface without seriously compromising MPI functionality. To permit applications to take
6 advantage of environment-specific functionality, many MPI routines take an `info` argument
7 that allows an application to specify environment-specific information. There is a tradeoff
8 between functionality and portability: applications that make use of environment-specific
9 `info` are not portable.

10 MPI does not require the existence of an underlying “virtual machine” model, in which
11 there is a consistent global view of an MPI application and an implicit “operating system”
12 managing resources and processes. For instance, processes spawned by one task may not
13 be visible to another; additional hosts added to the runtime environment by one process
14 may not be visible in another process; tasks spawned by different processes may not be
15 automatically distributed over available resources.

16 Interaction between MPI and the runtime environment is limited to the following areas:

- 17 • A process may start new processes with `MPI_COMM_SPAWN` and
18 `MPI_COMM_SPAWN_MULTIPLE`.
- 19 • When a process spawns a child process, it may optionally use an `info` argument to tell
20 the runtime environment where or how to start the process. This extra information
21 may be opaque to MPI.
- 22 • An attribute `MPI_UNIVERSE_SIZE` (See Section 11.10.1) on `MPI_COMM_WORLD` tells a
23 program how “large” the initial runtime environment is, namely how many processes
24 can usefully be started in all. One can subtract the size of `MPI_COMM_WORLD` from
25 this value to find out how many processes might usefully be started in addition to
26 those already running.
27
28
29

30 11.8 Process Manager Interface

31 11.8.1 Processes in MPI

32 A process is represented in MPI by a (group, rank) pair. A (group, rank) pair specifies a
33 unique process but a process does not determine a unique (group, rank) pair, since a process
34 may belong to several groups.

35 11.8.2 Starting Processes and Establishing Communication

36 The following routine starts a number of MPI processes and establishes communication with
37 them, returning an inter-communicator.

38 *Advice to users.* It is possible in MPI to start an SPMD or MPMD application with a
39 fixed number of processes after initialization by first starting one process and having
40 that process start its siblings with `MPI_COMM_SPAWN`. This practice is discouraged
41 primarily for reasons of performance. If possible, it is preferable to start all processes
42 at once, as a single MPI application. (*End of advice to users.*)
43
44
45
46
47
48

MPI_COMM_SPAWN(command, argv, maxprocs, info, root, comm, intercomm, array_of_errcodes)			1
			2
IN	command	name of program to be spawned (string, significant only at root)	3
			4
IN	argv	arguments to command (array of strings, significant only at root)	5
			6
IN	maxprocs	maximum number of processes to start (integer, significant only at root)	7
			8
IN	info	a set of key-value pairs telling the runtime system where and how to start the processes (handle, significant only at root)	9
			10
IN	root	rank of process in which previous arguments are examined (integer)	11
			12
IN	comm	intra-communicator containing group of spawning processes (handle)	13
			14
OUT	intercomm	inter-communicator between original group and the newly spawned group (handle)	15
			16
OUT	array_of_errcodes	one code per process (array of integers)	17
			18
			19
			20
			21
			22

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[])
```

Fortran 2008 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
```

Fortran 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 inter-communicator. 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

1 collective over `comm`, and also may not return until `MPI_INIT` has been called in the chil-
2 dren. Similarly, `MPI_INIT` in the children may not return until all parents have called
3 `MPI_COMM_SPAWN`. In this sense, `MPI_COMM_SPAWN` in the parents and `MPI_INIT` in
4 the children form a collective operation over the union of parent and child processes. The
5 inter-communicator returned by `MPI_COMM_SPAWN` contains the parent processes in the
6 local group and the child processes in the remote group. The ordering of processes in the
7 local and remote groups is the same as the ordering of the group of the `comm` in the parents
8 and of `MPI_COMM_WORLD` of the children, respectively. This inter-communicator can be
9 obtained in the children through the function `MPI_COMM_GET_PARENT`.

10 *Advice to users.* An implementation may automatically establish communication
11 before `MPI_INIT` is called by the children. Thus, completion of `MPI_COMM_SPAWN`
12 in the parent does not necessarily mean that `MPI_INIT` has been called in the children
13 (although the returned inter-communicator can be used immediately). (*End of advice*
14 *to users.*)

15
16 The arguments are:

17 **command:** The `command` argument is a string containing the name of a program to be
18 spawned. The string is null-terminated in C. In Fortran, leading and trailing spaces
19 are stripped. MPI does not specify how to find the executable or how the working
20 directory is determined. These rules are implementation-dependent and should be
21 appropriate for the runtime environment.

22
23 *Advice to implementors.* The implementation should use a natural rule for
24 finding executables and determining working directories. For instance, a homo-
25 geneous system with a global file system might look first in the working directory
26 of the spawning process, or might search the directories in a `PATH` environment
27 variable as do Unix shells. An implementation should document its rules for
28 finding executables and determining working directories, and a high-quality im-
29 plementation should give the user some control over these rules. (*End of advice*
30 *to implementors.*)

31 If the program named in `command` does not call `MPI_INIT`, but instead forks a process
32 that calls `MPI_INIT`, the results are undefined. Implementations may allow this case
33 to work but are not required to.

34
35 *Advice to users.* MPI does not say what happens if the program you start is a
36 shell script and that shell script starts a program that calls `MPI_INIT`. Though
37 some implementations may allow you to do this, they may also have restrictions,
38 such as requiring that arguments supplied to the shell script be supplied to the
39 program, or requiring that certain parts of the environment not be changed.
40 (*End of advice to users.*)

41 **argv:** `argv` is an array of strings containing arguments that are passed to the program. The
42 first element of `argv` is the first argument passed to `command`, not, as is conventional in
43 some contexts, the command itself. The argument list is terminated by `NULL` in C and
44 an empty string in Fortran. In Fortran, leading and trailing spaces are always stripped,
45 so that a string consisting of all spaces is considered an empty string. The constant
46 `MPI_ARGV_NULL` may be used in C and Fortran to indicate an empty argument list.
47 In C this constant is the same as `NULL`.

48

Example 11.19. 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`.

maxprocs: 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 11.8.4 for more information on the "soft" key for info.

Advice to users. By default, requests are hard and MPI errors are fatal. This means that by default there will be a fatal error if MPI cannot spawn all the requested processes. If you want the behavior "spawn as many processes as possible, up to N ," you should do a soft spawn, where the set of allowed values $\{m_i\}$ is $\{0, \dots, N\}$. However, this is not completely portable, as implementations are not required to support soft spawning. (*End of advice to users.*)

info: 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 10.

For the SPAWN calls, info provides additional (and possibly implementation-dependent) instructions to MPI and the runtime system on how to start processes. An application may pass MPI_INFO_NULL in C or Fortran. Portable programs not requiring detailed control over process locations should use MPI_INFO_NULL.

MPI does not specify the content of the info argument, except to reserve a number of special key values (see Section 11.8.4). 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.

root: 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.

array_of_errcodes: The array_of_errcodes is an array of length maxprocs in which MPI reports the status of each process that MPI was requested to start. If all maxprocs processes were spawned, array_of_errcodes is filled in with the value MPI_SUCCESS. If only m ($0 \leq m < \text{maxprocs}$) processes 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) 1

OUT parent the parent communicator (handle) 2

C binding 4

int MPI_Comm_get_parent(MPI_Comm *parent) 5

Fortran 2008 binding 7

MPI_Comm_get_parent(parent, ierror) 8

TYPE(MPI_Comm), INTENT(OUT) :: parent 9

INTEGER, OPTIONAL, INTENT(OUT) :: ierror 10

Fortran binding 12

MPI_COMM_GET_PARENT(PARENT, IERROR) 13

INTEGER PARENT, IERROR 14

If a process was started with MPI_COMM_SPAWN or MPI_COMM_SPAWN_MULTIPLE, MPI_COMM_GET_PARENT returns the “parent” inter-communicator of the current process. This parent inter-communicator is created implicitly inside of MPI_INIT and is the same inter-communicator returned by SPAWN in the parents. 15-18

If the process was not spawned, MPI_COMM_GET_PARENT returns MPI_COMM_NULL. 19

After the parent communicator is freed or disconnected, MPI_COMM_GET_PARENT returns MPI_COMM_NULL. 20-21

Advice to users. MPI_COMM_GET_PARENT returns a handle to a single inter-communicator. Calling MPI_COMM_GET_PARENT a second time returns a handle to the same inter-communicator. Freeing the handle with MPI_COMM_DISCONNECT or MPI_COMM_FREE will cause other references to the inter-communicator to become invalid (dangling). Note that calling MPI_COMM_FREE on the parent communicator is not useful. (*End of advice to users.*) 22-29

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.*) 30-33

11.8.3 Starting Multiple Executables and Establishing Communication 35

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. 36-40

MPI_COMM_SPAWN_MULTIPLE(count, array_of_commands, array_of_argv,
array_of_maxprocs, array_of_info, root, comm, intercomm,
array_of_errcodes) 42-44

IN count number of commands (positive integer, significant
only at root) 45-47

1	IN	array_of_commands	programs to be executed (array of strings, significant only at root)
2			
3	IN	array_of_argv	arguments for commands (array of array of strings, significant only at root)
4			
5	IN	array_of_maxprocs	maximum number of processes to start for each command (array of integers, significant only at root)
6			
7			
8	IN	array_of_info	info objects telling the runtime system where and how to start processes (array of handles, significant only at root)
9			
10			
11	IN	root	rank of process in which previous arguments are examined (integer)
12			
13	IN	comm	intra-communicator containing group of spawning processes (handle)
14			
15			
16	OUT	intercomm	inter-communicator between original group and the newly spawned group (handle)
17			
18	OUT	array_of_errcodes	one error code per process (array of integers)
19			

C binding

```

20
21 int MPI_Comm_spawn_multiple(int count, char *array_of_commands[],
22                             char **array_of_argv[], const int array_of_maxprocs[],
23                             const MPI_Info array_of_info[], int root, MPI_Comm comm,
24                             MPI_Comm *intercomm, int array_of_errcodes[])
25

```

Fortran 2008 binding

```

26 MPI_Comm_spawn_multiple(count, array_of_commands, array_of_argv,
27                         array_of_maxprocs, array_of_info, root, comm, intercomm,
28                         array_of_errcodes, ierror)
29
30 INTEGER, INTENT(IN) :: count, array_of_maxprocs(*), root
31 CHARACTER(LEN=*) , INTENT(IN) :: array_of_commands(*),
32 array_of_argv(count, *)
33 TYPE(MPI_Info), INTENT(IN) :: array_of_info(*)
34 TYPE(MPI_Comm), INTENT(IN) :: comm
35 TYPE(MPI_Comm), INTENT(OUT) :: intercomm
36 INTEGER :: array_of_errcodes(*)
37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38

```

Fortran binding

```

39 MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
40                         ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,
41                         ARRAY_OF_ERRCODES, IERROR)
42
43 INTEGER COUNT, ARRAY_OF_MAXPROCS(*), ARRAY_OF_INFO(*), ROOT, COMM,
44 INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR
45 CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)
46

```

46 MPI_COMM_SPAWN_MULTIPLE is identical to MPI_COMM_SPAWN except that there
47 are multiple executable specifications. The first argument, count, gives the number of
48 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 nonpositive value of `count` at a nonroot 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 nonroot 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 nonroot 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 $0, 1, \dots, m_1 - 1$. The processes in the second command have ranks $m_1, m_1 + 1, \dots, m_1 + m_2 - 1$. The processes in the third have ranks $m_1 + m_2, m_1 + m_2 + 1, \dots, m_1 + m_2 + m_3 - 1$, 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 $[\sum_{j=1}^i n_j] - 1$. Error codes are treated the same as with `MPI_COMM_SPAWN`.

Example 11.20. 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:

```

1  char *array_of_commands[2] = {"ocean", "atmos"};
2
3
4  char **array_of_argv[2];
5  char *argv0[] = {"-gridfile", "ocean1.grd", (char *)0};
6  char *argv1[] = {"atmos.grd", (char *)0};
7  array_of_argv[0] = argv0;
8  array_of_argv[1] = argv1;
9  MPI_Comm_spawn_multiple(2, array_of_commands, array_of_argv, ...);
10
11
12
13

```

Here is how you do it in Fortran:

```

14
15  CHARACTER*25 commands(2), array_of_argv(2, 3)
16  commands(1) = 'ocean'
17  array_of_argv(1, 1) = '-gridfile'
18  array_of_argv(1, 2) = 'ocean1.grd'
19  array_of_argv(1, 3) = ' '
20
21  commands(2) = 'atmos'
22  array_of_argv(2, 1) = 'atmos.grd'
23  array_of_argv(2, 2) = ' '
24
25  call MPI_COMM_SPAWN_MULTIPLE(2, commands, array_of_argv, ...)
26
27

```

11.8.4 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.

"mpi_initial_errhandler": Value is the name of an errhandler that will be set as the initial error handler. The "mpi_initial_errhandler" key can take the case insensitive values "mpi_errors_are_fatal", "mpi_errors_abort", and "mpi_errors_return" representing the pre-defined MPI error handlers (`MPI_ERRORS_ARE_FATAL`—the default,

MPI_ERRORS_ABORT, and MPI_ERRORS_RETURN, respectively). Other, nonstandard values may be supported by the implementation, which should document the resultant behavior.

"soft": Value specifies a set of numbers that 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 that 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 $a, a + 1, a + 2, \dots, b$
3. `a:b:c` means $a, a + c, a + 2c, \dots, a + ck$, 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 up to a maximum of 10,000 processes.
5. `2:10:2,7` allows 2, 4, 6, 7, 8, or 10 processes.

11.8.5 Spawn Example

Example 11.21. Manager-worker Example Using MPI_COMM_SPAWN

```

/* manager */
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{
    int world_size, universe_size, *universe_sizep, flag;
    MPI_Comm everyone;          /* inter-communicator */
    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) {

```

```

1     printf("This MPI does not support UNIVERSE_SIZE. How many\n\
2 processes total?");
3     scanf("%d", &universe_size);
4 } else universe_size = *universe_sizep;
5 if (universe_size == 1) error("No room to start workers");
6
7 /*
8  * Now spawn the workers. Note that there is a run-time determination
9  * of what type of worker to spawn, and presumably this calculation
10  * must be done at run time and cannot be calculated before starting
11  * the program. If everything is known when the application is
12  * first started, it is generally better to start them all at once
13  * in a single MPI_COMM_WORLD.
14  */
15 choose_worker_program(worker_program);
16 MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, universe_size-1,
17               MPI_INFO_NULL, 0, MPI_COMM_SELF, &everyone,
18               MPI_ERRCODES_IGNORE);
19 /*
20  * Parallel code here. The communicator "everyone" can be used
21  * to communicate with the spawned processes, which have ranks 0,..
22  * MPI_UNIVERSE_SIZE-1 in the remote group of the inter-communicator
23  * "everyone".
24  */
25 MPI_Finalize();
26 return 0;
27 }
28
29 /* worker */
30 #include "mpi.h"
31 int main(int argc, char *argv[])
32 {
33     int size;
34     MPI_Comm parent;
35     MPI_Init(&argc, &argv);
36     MPI_Comm_get_parent(&parent);
37     if (parent == MPI_COMM_NULL) error("No parent!");
38     MPI_Comm_remote_size(parent, &size);
39     if (size != 1) error("Something's wrong with the parent");
40
41     /*
42      * Parallel code here.
43      * The manager is represented as the process with rank 0 in (the
44      * remote group of) the parent communicator. If the workers need
45      * to communicate among themselves, they can use MPI_COMM_WORLD.
46      */
47     MPI_Finalize();
48     return 0;
49 }

```


11.9 Establishing Communication

This section provides functions that establish communication between two sets of MPI processes that do not share a communicator.

Some situations in which these functions are useful are:

1. Two parts of an application that are started independently need to communicate.
2. A visualization tool wants to attach to a running process.
3. A server wants to accept connections from multiple clients. Both clients and server may be parallel programs.

In each of these situations, MPI must establish communication channels where none existed before, and there is no parent/child relationship. The routines described in this section establish communication between the two sets of processes by creating an MPI inter-communicator, where the two groups of the inter-communicator are the original sets of processes.

Establishing contact between two groups of processes that do not share an existing communicator is a collective but asymmetric process. One group of processes indicates its willingness to accept connections from other groups of processes. We will call this group the (parallel) *server*, even if this is not a client/server type of application. The other group connects to the server; we will call it the *client*.

Advice to users. While the names *client* and *server* are used throughout this section, 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.*)

11.9.1 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.

11.9.2 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	<code>info</code>	implementation-specific information on how to establish an address (handle)
OUT	<code>port_name</code>	newly established port (string)

C binding

`int MPI_Open_port(MPI_Info info, char *port_name)`

Fortran 2008 binding

`MPI_Open_port(info, port_name, ierror)`

```

TYPE(MPI_Info), INTENT(IN) :: info
CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran 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)

```

Fortran 2008 binding

```

1 MPI_Close_port(port_name, ierror)
2   CHARACTER(LEN=*), INTENT(IN) :: port_name
3   INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

5 MPI_CLOSE_PORT(PORT_NAME, IERROR)
6   CHARACTER*(*) PORT_NAME
7   INTEGER IERROR

```

10 This function releases the network address represented by `port_name`.

```

12 MPI_COMM_ACCEPT(port_name, info, root, comm, newcomm)

```

14	IN	port_name	port name (string, significant only at root)
15	IN	info	implementation-dependent information (handle, significant only at root)
16			
17			
18	IN	root	rank in <code>comm</code> of root node (integer)
19	IN	comm	intra-communicator over which call is collective (handle)
20			
21	OUT	newcomm	inter-communicator with client as remote group (handle)
22			
23			

C binding

```

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

```

Fortran 2008 binding

```

26 MPI_Comm_accept(port_name, info, root, comm, newcomm, ierror)
27   CHARACTER(LEN=*), INTENT(IN) :: port_name
28   TYPE(MPI_Info), INTENT(IN) :: info
29   INTEGER, INTENT(IN) :: root
30   TYPE(MPI_Comm), INTENT(IN) :: comm
31   TYPE(MPI_Comm), INTENT(OUT) :: newcomm
32   INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

33 MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
34   CHARACTER*(*) PORT_NAME
35   INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR

```

41 `MPI_COMM_ACCEPT` establishes communication with a client. It is collective over the calling communicator. It returns an inter-communicator that allows communication with the client.

44 The `port_name` must have been established through a call to `MPI_OPEN_PORT`.

45 `info` can be used to provide directives that may influence the behavior of the `ACCEPT` call.

11.9.3 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 (integer)
IN	comm	intra-communicator over which call is collective (handle)
OUT	newcomm	inter-communicator 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)
```

Fortran 2008 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
```

Fortran binding

```
MPI_COMM_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
  CHARACTER*(*) PORT_NAME
  INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
```

This routine establishes communication with a server specified by `port_name`. It is collective over the calling communicator and returns an inter-communicator in which the remote group participated in an `MPI_COMM_ACCEPT`.

If the named port does not exist (or has been closed), `MPI_COMM_CONNECT` raises an error of class `MPI_ERR_PORT`.

If the port exists, but does not have a pending `MPI_COMM_ACCEPT`, the connection attempt will eventually time out after an implementation-defined time, or succeed when the server calls `MPI_COMM_ACCEPT`. In the case of a time out, `MPI_COMM_CONNECT` raises an error of class `MPI_ERR_PORT`.

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.

11.9.4 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	<code>service_name</code>	a service name to associate with the port (string)
IN	<code>info</code>	implementation-specific information (handle)
IN	<code>port_name</code>	a port name (string)

C binding

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

Fortran 2008 binding

```
MPI_Publish_name(service_name, info, port_name, ierror)
  CHARACTER(LEN=*) INTENT(IN) :: service_name, port_name
  TYPE(MPI_Info), INTENT(IN) :: info
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran 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	<code>service_name</code>	a service name (string)
IN	<code>info</code>	implementation-specific information (handle)
IN	<code>port_name</code>	a port name (string)

C binding

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

Fortran 2008 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
```

Fortran binding

```
MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
  CHARACTER*(*) SERVICE_NAME, PORT_NAME
```

1 INTEGER INFO, IERROR

2
3 This routine unpublishes a service name that has been previously published. Attempt-
4 ing to unpublish a name that has not been published or has already been unpublished is
5 erroneous and is indicated by the error class MPI_ERR_SERVICE.

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

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

13
14 MPI_LOOKUP_NAME(service_name, info, port_name)

15	IN	service_name	a service name (string)
16	IN	info	implementation-specific information (handle)
17	OUT	port_name	a port name (string)

20 C binding

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

23 Fortran 2008 binding

24 MPI_Lookup_name(service_name, info, port_name, ierror)
25 CHARACTER(LEN=*) , INTENT(IN) :: service_name
26 TYPE(MPI_Info), INTENT(IN) :: info
27 CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name
28 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

29 Fortran binding

30 MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
31 CHARACTER*(*) SERVICE_NAME, PORT_NAME
32 INTEGER INFO, IERROR

33
34 This function retrieves a port_name published by MPI_PUBLISH_NAME with
35 service_name. If service_name has not been published, it raises an error in the error class
36 MPI_ERR_NAME. The application must supply a port_name buffer large enough to hold the
37 largest possible port name (see discussion above under MPI_OPEN_PORT).

38 If an implementation allows multiple entries with the same service_name within the
39 same scope, a particular port_name is chosen in a way determined by the implementation.

40 If the info argument was used with MPI_PUBLISH_NAME to tell the implementation
41 how to publish names, a similar info argument may be required for MPI_LOOKUP_NAME.

43 11.9.5 Reserved Key Values

44 The following key values are reserved. An implementation is not required to interpret these
45 key values, but if it does interpret the key value, it must provide the functionality described.

46
47 "**ip_port**": Value contains IP port number at which to establish a port. (Reserved for
48 MPI_OPEN_PORT only).

"ip_address": Value contains IP address at which to establish a port. If the address is not a valid IP address of the host on which the MPI_OPEN_PORT call is made, the results are undefined. (Reserved for MPI_OPEN_PORT only).

11.9.6 Client/Server Examples

Example 11.22. 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);
```

Example 11.23. Ocean/Atmosphere—Relies on Name Publishing

In this example, the “ocean” application is the “server” side of a coupled ocean-atmosphere climate model. It assumes that the MPI implementation publishes names.

```
MPI_Open_port(MPI_INFO_NULL, port_name);
MPI_Publish_name("ocean", MPI_INFO_NULL, port_name);

MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF,
                &intercomm);
/* do something with intercomm */
MPI_Unpublish_name("ocean", MPI_INFO_NULL, port_name);
```

On the client side:

```
MPI_Lookup_name("ocean", MPI_INFO_NULL, port_name);
MPI_Comm_connect(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF,
                &intercomm);
```

Example 11.24. 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.

```

1  #include "mpi.h"
2  int main(int argc, char *argv[])
3  {
4      MPI_Comm client;
5      MPI_Status status;
6      char port_name[MPI_MAX_PORT_NAME];
7      double buf[MAX_DATA];
8      int size, again;
9
10     MPI_Init(&argc, &argv);
11     MPI_Comm_size(MPI_COMM_WORLD, &size);
12     if (size != 1) error(FATAL, "Server too big");
13     MPI_Open_port(MPI_INFO_NULL, port_name);
14     printf("server available at %s\n", port_name);
15     while (1) {
16         MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
17                         &client);
18         again = 1;
19         while (again) {
20             MPI_Recv(buf, MAX_DATA, MPI_DOUBLE,
21                     MPI_ANY_SOURCE, MPI_ANY_TAG, client, &status);
22             switch (status.MPI_TAG) {
23                 case 0: MPI_Comm_free(&client);
24                         MPI_Close_port(port_name);
25                         MPI_Finalize();
26                         return 0;
27                 case 1: MPI_Comm_disconnect(&client);
28                         again = 0;
29                         break;
30                 case 2: /* do something */
31                         ...
32                 default:
33                         /* Unexpected message type */
34                         MPI_Abort(MPI_COMM_WORLD, 1);
35             }
36         }
37     }
38 }

```

Here is the client.

```

39 #include "mpi.h"
40 int main(int argc, char *argv[])
41 {
42     MPI_Comm server;
43     int done = 0;
44     double buf[MAX_DATA];
45     char port_name[MPI_MAX_PORT_NAME];
46
47     MPI_Init(&argc, &argv);
48     strcpy(port_name, argv[1]); /* assume server's name is cmd-line arg */

```

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
MPI_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;
}

```

11.10 Other Functionality

11.10.1 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 running one of these quasi-static applications, the user (or possibly a batch system) will usually specify a number of processes to start and a total number of processes that are expected. An application simply needs to know how many slots there are, i.e., how many processes it should spawn.

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

1 they are requested. However, it is likely that the user only wants to spawn one process per
2 processor.

3 MPI_UNIVERSE_SIZE is assumed to have been specified when an application was started,
4 and is in essence a portable mechanism to allow the user to pass to the application (through
5 the MPI process startup mechanism, such as `mpiexec`) a piece of critical runtime informa-
6 tion. Note that no interaction with the runtime environment is required. If the runtime
7 environment changes size while an application is running, MPI_UNIVERSE_SIZE is not up-
8 dated, and the application must find out about the change through direct communication
9 with the runtime system.

11 11.10.2 Singleton MPI Initialization

12 A high-quality implementation will allow any process (including those not started with a
13 “parallel application” mechanism) to become an MPI process by calling `MPI_INIT`,
14 `MPI_INIT_THREAD`, or `MPI_SESSION_INIT`. Such a process can then connect to other MPI
15 processes using the `MPI_COMM_ACCEPT` and `MPI_COMM_CONNECT` routines, or spawn
16 other MPI processes. MPI does not mandate this behavior, but strongly encourages it where
17 technically feasible.

19 *Advice to implementors.* Special coordination is required to start MPI processes
20 belonging to the same `MPI_COMM_WORLD` in the case of the World Model, or the
21 same “`mpi://WORLD`” process set in the Sessions Model. The processes must be started
22 at the “same” time, they must have a mechanism to establish communication, etc.
23 Either the user or the operating system must take special steps beyond simply starting
24 processes.

26 Considering the World Model, when an application enters `MPI_INIT`, clearly it must be
27 able to determine if these special steps were taken. If a process enters `MPI_INIT` and
28 determines that no special steps were taken (i.e., it has not been given the information
29 to form an `MPI_COMM_WORLD` with other processes) it succeeds and forms a singleton
30 MPI program, that is, one in which `MPI_COMM_WORLD` has size 1.

31 In some implementations, MPI may not be able to function without an “MPI environ-
32 ment.” For example, MPI may require that daemons be running or MPI may not be
33 able to work at all on the front-end of an MPP. In this case, an MPI implementation
34 may either

- 35 1. Create the environment (e.g., start a daemon) or
- 36 2. Raise an error if it cannot create the environment and the environment has not
37 been started independently.

39 A high-quality implementation will try to create a singleton MPI process and not raise
40 an error. (*End of advice to implementors.*)

42 11.10.3 MPI_APPNUM

44 There is a predefined attribute `MPI_APPNUM` of `MPI_COMM_WORLD`. In Fortran, the at-
45 tribute is an integer value. In C, the attribute is a pointer to an integer value. If a process
46 was spawned with `MPI_COMM_SPAWN_MULTIPLE`, `MPI_APPNUM` is the command number
47 that generated the current process. Numbering starts from zero. If a process was spawned
48 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

```
mpirexec 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.*)

11.10.4 Releasing Connections

Before a client and a 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 a 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 file handle.
 2. If A is connected to B and B to C, then A is connected to C.
- Two processes are **disconnected** (also **independent**) if they are not connected.
- By the above definitions, connectivity is a transitive property, and divides the universe of MPI processes into disconnected (independent) sets (equivalence classes) of processes.

- Processes that 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.

Advice to implementors. In practice, it may be difficult to distinguish between an MPI process failure and an erroneous program that terminates without calling an MPI finalization function: an implementation that defines semantics for process failure management may have to exhibit the behavior defined for MPI process failures with such erroneous programs. A high quality implementation should exhibit a different behavior for erroneous programs and MPI process failures. (*End of advice to implementors.*)

`MPI_COMM_DISCONNECT(comm)`

INOUT `comm` communicator (handle)

C binding

```
int MPI_Comm_disconnect(MPI_Comm *comm)
```

Fortran 2008 binding

```
MPI_Comm_disconnect(comm, ierror)
    TYPE(MPI_Comm), INTENT(INOUT) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_COMM_DISCONNECT(COMM, IERROR)
    INTEGER COMM, IERROR
```

This function waits for all pending communication on `comm` to complete internally, deallocates the communicator object, and sets the handle to `MPI_COMM_NULL`. It is a collective operation.

It may not be called with the communicator `MPI_COMM_WORLD` or `MPI_COMM_SELF`.

`MPI_COMM_DISCONNECT` may be called only if all communication is complete and matched, so that buffered data can be delivered to its destination. This requirement is the same as for `MPI_FINALIZE`.

`MPI_COMM_DISCONNECT` has the same action as `MPI_COMM_FREE`, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes.

Advice to users. To disconnect two processes you may need to call `MPI_COMM_DISCONNECT`, `MPI_WIN_FREE`, and `MPI_FILE_CLOSE` to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (*End of advice to users.*)

Rationale. It would be nice to be able to use `MPI_COMM_FREE` instead, but that function explicitly does not wait for pending communication to complete. (*End of rationale.*)

11.10.5 Another Way to Establish MPI Communication

`MPI_COMM_JOIN(fd, intercomm)`

IN	fd	socket file descriptor
OUT	intercomm	new inter-communicator (handle)

C binding

```
int MPI_Comm_join(int fd, MPI_Comm *intercomm)
```

Fortran 2008 binding

```
MPI_Comm_join(fd, intercomm, ierror)
  INTEGER, INTENT(IN) :: fd
  TYPE(MPI_Comm), INTENT(OUT) :: intercomm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran 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 [50, 56]. 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 inter-communicator from the union of two MPI processes that 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.*)

1 fd is a file descriptor representing a socket of type SOCK_STREAM (a two-way reliable
2 byte-stream connection). Nonblocking I/O and asynchronous notification via SIGIO must
3 not be enabled for the socket. The socket must be in a connected state. The socket must
4 be quiescent when MPI_COMM_JOIN is called (see below). It is the responsibility of the
5 application to create the socket using standard socket API calls.

6 MPI_COMM_JOIN must be called by the process at each end of the socket. It does not
7 return until both processes have called MPI_COMM_JOIN. The two processes are referred
8 to as the local and remote processes.

9 MPI uses the socket to bootstrap creation of the inter-communicator, and for nothing
10 else. Upon return from MPI_COMM_JOIN, the file descriptor will be open and quiescent
11 (see below).

12 If MPI is unable to create an inter-communicator, but is able to leave the socket in its
13 original state, with no pending communication, it succeeds and sets intercomm to
14 MPI_COMM_NULL.

15 The socket must be quiescent before MPI_COMM_JOIN is called and after
16 MPI_COMM_JOIN returns. More specifically, on entry to MPI_COMM_JOIN, a read on the
17 socket will not read any data that was written to the socket before the remote process called
18 MPI_COMM_JOIN. On exit from MPI_COMM_JOIN, a read will not read any data that was
19 written to the socket before the remote process returned from MPI_COMM_JOIN. It is the
20 responsibility of the application to ensure the first condition, and the responsibility of the
21 MPI implementation to ensure the second. In a multithreaded application, the application
22 must ensure that one thread does not access the socket while another is calling
23 MPI_COMM_JOIN, or call MPI_COMM_JOIN concurrently.

24
25 *Advice to implementors.* MPI is free to use any available communication path(s)
26 for MPI messages in the new communicator; the socket is only used for the initial
27 handshaking. (*End of advice to implementors.*)

28
29 MPI_COMM_JOIN uses non-MPI communication to do its work. The interaction of
30 non-MPI communication with pending MPI communication is not defined. Therefore, the
31 result of calling MPI_COMM_JOIN on two connected processes (see Section 11.10.4 for the
32 definition of connected) is undefined.

33 The returned communicator may be used to establish MPI communication with addi-
34 tional processes, through the usual MPI communicator creation mechanisms.

Chapter 12

One-Sided Communications

12.1 Introduction

Remote Memory Access (RMA) extends the communication mechanisms of MPI by allowing one process to specify all communication parameters, both for the sending side and for the receiving side. This mode of communication facilitates the coding of some applications with dynamically changing data access patterns where the data distribution is fixed or slowly changing. In such a case, each process can compute what data it needs to access or to update at other processes. However, the programmer may not be able to easily determine which data in a process may need to be accessed or to be updated by operations executed by a different process, and may not even know which processes may perform such updates. Thus, the transfer parameters are all available only on one side. Regular send/receive communication requires matching operations by sender and receiver. In order to issue the matching operations, an application needs to distribute the transfer parameters. This distribution may require all processes to participate in a time-consuming global computation, or to poll for potential communication requests to receive and upon which to act periodically. The use of RMA communication mechanisms avoids the need for global computations or explicit polling. A generic example of this nature is the execution of an assignment of the form $A = B(\text{map})$, 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 12.4. 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`.

The use of terms such as nonblocking and local in this chapter follow the usage in MPI-3.1, and this chapter has not been updated to follow the definitions in Section 2.4. The MPI Forum intends to update this chapter in a subsequent version of the MPI standard to follow the definitions in Section 2.4.

12.2 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 intra-communicator. `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` and `MPI_WIN_ALLOCATE_SHARED` differ from `MPI_WIN_CREATE` in that the user does not pass allocated memory; instead `MPI_WIN_ALLOCATE` and `MPI_WIN_ALLOCATE_SHARED` return 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.

12.2.1 Window Creation

`MPI_WIN_CREATE(base, size, disp_unit, info, comm, win)`

IN	<code>base</code>	initial address of window (choice)
IN	<code>size</code>	size of window in bytes (non-negative integer)
IN	<code>disp_unit</code>	local unit size for displacements, in bytes (positive integer)
IN	<code>info</code>	info argument (handle)

IN	comm	intra-communicator (handle)	1
OUT	win	window object (handle)	2

C binding

```
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_c(void *base, MPI_Aint size, MPI_Aint disp_unit,
                    MPI_Info info, MPI_Comm comm, MPI_Win *win)
```

Fortran 2008 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
```

```
MPI_Win_create(base, size, disp_unit, info, comm, win, ierror) !(_c)
  TYPE(*), DIMENSION(..), ASYNCHRONOUS :: base
  INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size, 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
```

Fortran 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 19.1.12). 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" (boolean, default: "false"): 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" (string, default "rar,raw,war,waw"): controls the ordering of accumulate operations at the target. See Section 12.7.2 for details.

"accumulate_ops" (string, default: "same_op_no_op"): if set to "same_op", the implementation will assume that all concurrent accumulate calls to the same target address will use the same operator. 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 operator or `MPI_NO_OP`. This can eliminate the need to protect access for certain operators where the hardware can guarantee atomicity.

"same_size" (boolean, default: "false"): 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" (boolean, default: "false"): 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 12.2.7 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 9.2) will be better. Also, on some systems, performance is improved when window boundaries are aligned at “natural” boundaries (word, double-word, cache line, page frame, etc.). (*End of advice to users.*)

Advice to implementors. In cases where RMA operations use different mechanisms in different memory areas (e.g., load/store in a shared memory segment, and an asynchronous handler in private memory), the `MPI_WIN_CREATE` call needs to figure out which type of memory is used for the window. To do so, MPI maintains, internally, the list of memory segments allocated by `MPI_ALLOC_MEM`, or by other, implementation-specific, mechanisms, together with information on the type of memory segment allocated. When a call to `MPI_WIN_CREATE` occurs, then MPI checks which segment contains each window, and decides, accordingly, which mechanism to use for RMA operations.

Vendors may provide additional, implementation-specific mechanisms to allocate or to specify memory regions that are preferable for use in one-sided communication. In particular, such mechanisms can be used to place static variables into such preferred regions.

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

12.2.2 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 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)
```

```
int MPI_Win_allocate_c(MPI_Aint size, MPI_Aint disp_unit, MPI_Info info,
                    MPI_Comm comm, void *baseptr, MPI_Win *win)
```

Fortran 2008 binding

```

1 MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror)
2   USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
3   INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
4   INTEGER, INTENT(IN) :: disp_unit
5   TYPE(MPI_Info), INTENT(IN) :: info
6   TYPE(MPI_Comm), INTENT(IN) :: comm
7   TYPE(C_PTR), INTENT(OUT) :: baseptr
8   TYPE(MPI_Win), INTENT(OUT) :: win
9   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11 MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror) !(_c)
12   USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
13   INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size, disp_unit
14   TYPE(MPI_Info), INTENT(IN) :: info
15   TYPE(MPI_Comm), INTENT(IN) :: comm
16   TYPE(C_PTR), INTENT(OUT) :: baseptr
17   TYPE(MPI_Win), INTENT(OUT) :: win
18   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19

```

Fortran binding

```

20 MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
21   INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
22   INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
23
24

```

25 This is a collective call executed by all processes in the group of `comm`. On each
26 process, it allocates memory of at least `size` bytes, returns a pointer to it, and returns a
27 window object that can be used by all processes in `comm` to perform RMA operations. The
28 returned memory consists of `size` bytes local to each process, starting at address `baseptr` and
29 is associated with the window as if the user called `MPI_WIN_CREATE` on existing memory.
30 The size argument may be different at each process and `size = 0` is valid; however, a
31 library might allocate and expose more memory in order to create a fast, globally symmetric
32 allocation. The discussion of and rationales for `MPI_ALLOC_MEM` and `MPI_FREE_MEM` in
33 Section 9.2 also apply to `MPI_WIN_ALLOCATE`; in particular, see the rationale in Section 9.2
34 for an explanation of the type used for `baseptr`.

35 If the Fortran compiler provides `TYPE(C_PTR)`, then the following generic interface must
36 be provided in the `mpi` module and should be provided in `mpif.h` through overloading, i.e.,
37 with the same routine name as the routine with `INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR`,
38 but with a different specific procedure name:

```

39 INTERFACE MPI_WIN_ALLOCATE
40   SUBROUTINE MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
41     WIN, IERROR)
42     IMPORT :: MPI_ADDRESS_KIND
43     INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
44     INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR
45   END SUBROUTINE
46   SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
47     WIN, IERROR)
48

```

```

USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
IMPORT :: MPI_ADDRESS_KIND
INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
TYPE(C_PTR) :: BASEPTR
END SUBROUTINE
END INTERFACE

```

The base procedure name of this overloaded function is `MPI_WIN_ALLOCATE_CPTR`. The implied specific procedure names are described in Section 19.1.5.

Rationale. By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with remote direct memory access. This also permits the collective allocation of memory and supports what is sometimes called the “symmetric allocation” model that can be more scalable (for example, the implementation can arrange to return an address for the allocated memory that is the same on all processes). (*End of rationale.*)

The `info` argument can be used to specify hints similar to the `info` argument for `MPI_WIN_CREATE` and `MPI_ALLOC_MEM`.

The default memory alignment requirements and the “`mpi_minimum_memory_alignment`” `info` key described for `MPI_ALLOC_MEM` in Section 9.2 apply to all processes with nonzero size argument.

12.2.3 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)
```

```
int MPI_Win_allocate_shared_c(MPI_Aint size, MPI_Aint disp_unit, MPI_Info info,
    MPI_Comm comm, void *baseptr, MPI_Win *win)
```

Fortran 2008 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

```

```

1      INTEGER, INTENT(IN) :: disp_unit
2      TYPE(MPI_Info), INTENT(IN) :: info
3      TYPE(MPI_Comm), INTENT(IN) :: comm
4      TYPE(C_PTR), INTENT(OUT) :: baseptr
5      TYPE(MPI_Win), INTENT(OUT) :: win
6      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8      MPI_Win_allocate_shared(size, disp_unit, info, comm, baseptr, win, ierror)
9          !(_c)
10     USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
11     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size, disp_unit
12     TYPE(MPI_Info), INTENT(IN) :: info
13     TYPE(MPI_Comm), INTENT(IN) :: comm
14     TYPE(C_PTR), INTENT(OUT) :: baseptr
15     TYPE(MPI_Win), INTENT(OUT) :: win
16     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

17 MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
18     INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
19     INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR

```

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 9.2 also apply to `MPI_WIN_ALLOCATE_SHARED`; in particular, see the rationale in Section 9.2 for an explanation of the type used for `baseptr`. The allocated memory is contiguous across 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:

```

43 INTERFACE MPI_WIN_ALLOCATE_SHARED
44     SUBROUTINE MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, &
45         BASEPTR, WIN, IERROR)
46         IMPORT :: MPI_ADDRESS_KIND
47         INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
48

```



```

    INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR
1
END SUBROUTINE
2
SUBROUTINE MPI_WIN_ALLOCATE_SHARED_CPTR(SIZE, DISP_UNIT, INFO, COMM, &
3
    BASEPTR, WIN, IERROR)
4
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
5
    IMPORT :: MPI_ADDRESS_KIND
6
    INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
7
    INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
8
    TYPE(C_PTR) :: BASEPTR
9
END SUBROUTINE
10
END INTERFACE
11

```

The base procedure name of this overloaded function is `MPI_WIN_ALLOCATE_SHARED_CPTR`. The implied specific procedure names are described in Section 19.1.5.

The `info` argument can be used to specify hints similar to the `info` argument for `MPI_WIN_CREATE`, `MPI_WIN_ALLOCATE`, and `MPI_ALLOC_MEM`. The additional `info` key `"alloc_shared_noncontig"` allows the library to optimize the layout of the shared memory segments in memory.

Advice to users. If the `info` key `"alloc_shared_noncontig"` is not set to true, the allocation 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.*)

For contiguous shared memory allocations, the default alignment requirements outlined for `MPI_ALLOC_MEM` in Section 9.2 and the `"mpi_minimum_memory_alignment"` `info` key apply to the start of the contiguous memory that is returned in `baseptr` to the first process with nonzero `size` argument. For noncontiguous memory allocations, the default alignment requirements and the `"mpi_minimum_memory_alignment"` `info` key apply to all processes with nonzero `size` argument.

Advice to users. If the `info` key `"alloc_shared_noncontig"` is not set to true (or ignored by the MPI implementation), the alignment of the memory returned in `baseptr` to all but the first process with nonzero `size` argument depends on the value of the `size` argument provided by other processes. It is thus the user's responsibility to control the alignment of contiguous memory allocated for these processes by ensuring that each process provides a `size` argument that is an integral multiple of the alignment required for the application. (*End of advice to users.*)

The consistency of load/store accesses from/to the shared memory as observed by the user program depends on the architecture. A consistent view can be created in the *unified*

1 *memory model* (see Section 12.4) by utilizing the window synchronization functions (see
 2 Section 12.5) or explicitly completing outstanding store accesses (e.g., by calling
 3 MPI_WIN_FLUSH). MPI does not define semantics for accessing shared memory windows
 4 in the *separate memory model*.

5
 6
 7 MPI_WIN_SHARED_QUERY(win, rank, size, disp_unit, baseptr)

8 IN win shared memory window object (handle)
 9 IN rank rank in the group of window win or
 10 MPI_PROC_NULL (non-negative integer)
 11
 12 OUT size size of the window segment (non-negative integer)
 13 OUT disp_unit local unit size for displacements, in bytes (positive
 14 integer)
 15 OUT baseptr address for load/store access to window segment
 16 (choice)
 17

18 C binding

19 int MPI_Win_shared_query(MPI_Win win, int rank, MPI_Aint *size, int *disp_unit,
 20 void *baseptr)
 21

22 int MPI_Win_shared_query_c(MPI_Win win, int rank, MPI_Aint *size,
 23 MPI_Aint *disp_unit, void *baseptr)
 24

25 Fortran 2008 binding

26 MPI_Win_shared_query(win, rank, size, disp_unit, baseptr, ierror)
 27 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
 28 TYPE(MPI_Win), INTENT(IN) :: win
 29 INTEGER, INTENT(IN) :: rank
 30 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size
 31 INTEGER, INTENT(OUT) :: disp_unit
 32 TYPE(C_PTR), INTENT(OUT) :: baseptr
 33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

34 MPI_Win_shared_query(win, rank, size, disp_unit, baseptr, ierror) !(_c)
 35 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
 36 TYPE(MPI_Win), INTENT(IN) :: win
 37 INTEGER, INTENT(IN) :: rank
 38 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size, disp_unit
 39 TYPE(C_PTR), INTENT(OUT) :: baseptr
 40 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 41

42 Fortran binding

43 MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, BASEPTR, IERROR)
 44 INTEGER WIN, RANK, DISP_UNIT, IERROR
 45 INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR

46 This function queries the process-local address for remote memory segments created
 47 with MPI_WIN_ALLOCATE_SHARED. This function can return different process-local ad-
 48 dresses for the same physical memory on different processes. The returned memory can be

used for load/store accesses subject to the constraints defined in Section 12.7. This function can only be called with windows of flavor `MPI_WIN_FLAVOR_SHARED`. If the passed window is not of flavor `MPI_WIN_FLAVOR_SHARED`, the error `MPI_ERR_RMA_FLAVOR` is raised. When rank is `MPI_PROC_NULL`, the pointer, `disp_unit`, and size returned are the pointer, `disp_unit`, and size of the memory segment belonging the lowest rank that specified `size > 0`. If all processes in the group attached to the window specified `size = 0`, then the call returns `size = 0` and a `baseptr` as if `MPI_ALLOC_MEM` was called with `size = 0`.

If the Fortran compiler provides `TYPE(C_PTR)`, then the following generic interface must be provided in the `mpi` module and should be provided in `mpif.h` through overloading, i.e., with the same routine name as the routine with `INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR`, but with a different specific procedure name:

```

INTERFACE MPI_WIN_SHARED_QUERY
  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 19.1.5.

12.2.4 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`.

```

1 MPI_WIN_CREATE_DYNAMIC(info, comm, win)
2     IN      info                info argument (handle)
3
4     IN      comm                intra-communicator (handle)
5
6     OUT     win                 window object returned by the call (handle)
7

```

C binding

```

8 int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win)
9

```

Fortran 2008 binding

```

10 MPI_Win_create_dynamic(info, comm, win, ierror)
11     TYPE(MPI_Info), INTENT(IN) :: info
12     TYPE(MPI_Comm), INTENT(IN) :: comm
13     TYPE(MPI_Win), INTENT(OUT) :: win
14     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15

```

Fortran binding

```

16 MPI_WIN_CREATE_DYNAMIC(INFO, COMM, WIN, IERROR)
17     INTEGER INFO, COMM, WIN, IERROR
18
19

```

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 nonnegative values. Users should use `MPI_GET_ADDRESS` at the target process to determine the address of a target memory location and communicate this address to the origin process.

Advice to users. Users are cautioned that displacement arithmetic can overflow in variables of type `MPI_Aint` and result in unexpected values on some platforms. The `MPI_AINT_ADD` and `MPI_AINT_DIFF` functions can be used to safely perform address arithmetic with `MPI_Aint` displacements. (*End of advice to users.*)

Advice to implementors. In environments with heterogeneous data representations, care must be exercised in communicating addresses between processes. For example, it is possible that an address valid at the target process (for example, a 64-bit pointer) cannot be expressed as an address at the origin (for example, the origin uses 32-bit pointers). For this reason, a portable MPI implementation should ensure that the type `MPI_AINT` (see Table 3.3) is able to store addresses from any process. (*End of advice to implementors.*)

Memory at the target cannot be accessed with this window until that memory has been attached using the function `MPI_WIN_ATTACH`. That is, in addition to using

MPI_WIN_CREATE_DYNAMIC to create an MPI window, the user must use MPI_WIN_ATTACH before any local memory may be the target of an MPI RMA operation. Only memory that is currently accessible may be attached.

MPI_WIN_ATTACH(win, base, size)

IN	win	window object (handle)
IN	base	initial address of memory to be attached (choice)
IN	size	size of memory to be attached in bytes (non-negative integer)

C binding

```
int MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size)
```

Fortran 2008 binding

```
MPI_Win_attach(win, base, size, ierror)
  TYPE(MPI_Win), INTENT(IN) :: win
  TYPE(*), DIMENSION(..), ASYNCHRONOUS :: base
  INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_WIN_ATTACH(WIN, BASE, SIZE, IERROR)
  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 19.1.12). Multiple (but nonoverlapping) 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.

1 Performing an RMA operation to memory that has not been attached to a window
 2 created with `MPI_WIN_CREATE_DYNAMIC` is erroneous. (*End of advice to users.*)

3
 4 *Advice to implementors.* A high-quality implementation will attempt to make as
 5 much memory available for attaching as possible. Any limitations should be docu-
 6 mented by the implementor. (*End of advice to implementors.*)

7
 8 Attaching memory is a local operation as defined by MPI, which means that the call
 9 is not collective and completes without requiring any MPI routine to be called in any other
 10 process. Memory may be detached with the routine `MPI_WIN_DETACH`. After memory has
 11 been detached, it may not be the target of an MPI RMA operation on that window (unless
 12 the memory is re-attached with `MPI_WIN_ATTACH`).

13
 14 `MPI_WIN_DETACH(win, base)`

15
 16 IN win window object (handle)
 17 IN base initial address of memory to be detached (choice)

18
 19 **C binding**

20 `int MPI_Win_detach(MPI_Win win, const void *base)`

21
 22 **Fortran 2008 binding**

23 `MPI_Win_detach(win, base, ierror)`
 24 TYPE(MPI_Win), INTENT(IN) :: win
 25 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: base
 26 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

27
 28 **Fortran binding**

29 `MPI_WIN_DETACH(WIN, BASE, IERROR)`
 30 INTEGER WIN, IERROR
 31 <type> BASE(*)

32 Detaches a previously attached memory region beginning at `base`. The arguments `base`
 33 and `win` must match the arguments passed to a previous call to `MPI_WIN_ATTACH`.

34
 35 *Advice to users.* Detaching memory may permit the implementation to make more
 36 efficient use of special memory or provide memory that may be needed by a subsequent
 37 `MPI_WIN_ATTACH`. Users are encouraged to detach memory that is no longer needed.
 38 Memory should be detached before it is freed by the user. (*End of advice to users.*)

39
 40 Memory becomes detached when the associated dynamic memory window is freed, see
 41 Section [12.2.5](#).

12.2.5 Window Destruction

`MPI_WIN_FREE(win)`

INOUT win window object (handle)

C binding

```
int MPI_Win_free(MPI_Win *win)
```

Fortran 2008 binding

```
MPI_Win_free(win, ierror)
  TYPE(MPI_Win), INTENT(INOUT) :: win
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran 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` 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`.

`MPI_WIN_FREE` is required to delay its return until all accesses to the local window using passive target synchronization have completed. Therefore, it is synchronizing unless the window was created with the "no_locks" info key set to "true".

12.2.6 Window Attributes

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

<code>MPI_WIN_BASE</code>	window base address.
<code>MPI_WIN_SIZE</code>	window size, in bytes.
<code>MPI_WIN_DISP_UNIT</code>	displacement unit associated with the window.
<code>MPI_WIN_CREATE_FLAVOR</code>	how the window was created.
<code>MPI_WIN_MODEL</code>	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)`,

Table 12.1: C types of attribute value argument to MPI_WIN_GET_ATTR and MPI_WIN_SET_ATTR

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 *

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 12.1.

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 12.4.

In the case of windows created with MPI_WIN_CREATE_DYNAMIC, the base address is MPI_BOTTOM and the size is 0. In C, pointers are returned, and in Fortran, the values are returned, for the respective attributes. (The window attribute access functions are defined in Section 7.7.3.) The value returned for an attribute on a window is constant over the lifetime of the window.

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

MPI_WIN_GET_GROUP(win, group)

IN win window object (handle)

OUT	group	group of processes that share access to the window (handle)	1 2 3
-----	-------	--	-------------

C binding

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

Fortran 2008 binding

```
MPI_Win_get_group(win, group, ierror)
  TYPE(MPI_Win), INTENT(IN) :: win
  TYPE(MPI_Group), INTENT(OUT) :: group
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_WIN_GET_GROUP(WIN, GROUP, IERROR)
  INTEGER WIN, GROUP, IERROR
```

MPI_WIN_GET_GROUP returns a duplicate of the group of the communicator used to create the window associated with win. The group is returned in group.

12.2.7 Window Info

Hints specified via info (see Section 10) allow a user to provide information to direct optimization. Providing hints may enable an implementation to deliver increased performance 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)	40 41
IN	info	info argument (handle)	42 43

C binding

```
int MPI_Win_set_info(MPI_Win win, MPI_Info info)
```

Fortran 2008 binding

```
MPI_Win_set_info(win, info, ierror)
```

```

1     TYPE(MPI_Win), INTENT(IN) :: win
2     TYPE(MPI_Info), INTENT(IN) :: info
3     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4

```

Fortran binding

```

5 MPI_WIN_SET_INFO(WIN, INFO, IERROR)
6     INTEGER WIN, INFO, IERROR
7

```

8 MPI_WIN_SET_INFO updates the hints of the window associated with win using the
9 hints provided in info. This operation has no effect on previously set or defaulted hints
10 that are not specified by info. It also has no effect on previously set or defaulted hints that
11 are specified by info, but are ignored by the MPI implementation in this call to
12 MPI_WIN_SET_INFO. The call is collective on the group of win. The info object may be
13 different on each process, but any info entries that an implementation requires to be the
14 same on all processes must appear with the same value in each process's info object.

16 *Advice to users.* Some info items that an implementation can use when it creates
17 a window cannot easily be changed once the window has been created. Thus, an
18 implementation may ignore hints issued in this call that it would have accepted in a
19 creation call. An implementation may also be unable to update certain info hints in a
20 call to MPI_WIN_SET_INFO. MPI_WIN_GET_INFO can be used to determine whether
21 info changes were ignored by the implementation. (*End of advice to users.*)

```

22
23
24 MPI_WIN_GET_INFO(win, info_used)
25
26     IN        win                window object (handle)
27     OUT       info_used          new info object (handle)
28

```

C binding

```

29
30 int MPI_Win_get_info(MPI_Win win, MPI_Info *info_used)
31

```

Fortran 2008 binding

```

32 MPI_Win_get_info(win, info_used, ierror)
33     TYPE(MPI_Win), INTENT(IN) :: win
34     TYPE(MPI_Info), INTENT(OUT) :: info_used
35     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36

```

Fortran binding

```

37
38 MPI_WIN_GET_INFO(WIN, INFO_USED, IERROR)
39     INTEGER WIN, INFO_USED, IERROR
40

```

41 MPI_WIN_GET_INFO returns a new info object containing the hints of the window
42 associated with win. The current setting of all hints related to this window is returned in
43 info_used. An MPI implementation is required to return all hints that are supported by
44 the implementation and have default values specified; any user-supplied hints that were not
45 ignored by the implementation; and any additional hints that were set by the implementa-
46 tion. If no such hints exist, a handle to a newly created info object is returned that contains
47 no key/value pair. The user is responsible for freeing info_used via MPI_INFO_FREE.

```

48

```

12.3 Communication Calls

MPI supports the following RMA communication calls: `MPI_PUT` and `MPI_RPUT` transfer data from the caller memory (origin) to the target memory; `MPI_GET` and `MPI_RGET` transfer data from the target memory to the caller memory; `MPI_ACCUMULATE` and `MPI_RACCUMULATE` update locations in the target memory, e.g., by adding to these locations values sent from the caller memory; `MPI_GET_ACCUMULATE`, `MPI_RGET_ACCUMULATE`, and `MPI_FETCH_AND_OP` perform atomic read-modify-write and return the data before the accumulate operation; and `MPI_COMPARE_AND_SWAP` performs a remote atomic compare and swap operation. These operations are *nonblocking*: the call initiates the transfer, but the transfer may continue after the call returns. The transfer is completed, at the origin or both the origin and the target, when a subsequent *synchronization* call is issued by the caller on the involved window object. These synchronization calls are described in Section 12.5. Transfers can also be completed with calls to flush routines; see Section 12.5.4 for details. For the `MPI_RPUT`, `MPI_RGET`, `MPI_RACCUMULATE`, and `MPI_RGET_ACCUMULATE` calls, the transfer can be locally completed by using the MPI test or wait operations described in Section 3.7.3.

The local communication buffer of an RMA call should not be updated, and the local communication buffer of a get call should not be accessed after the RMA call until the operation completes at the origin.

The resulting data values, or outcome, of concurrent conflicting accesses to the same 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 12.7.

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.

12.3.1 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.

```

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

```

3	IN	origin_addr	initial address of origin buffer (choice)
4			
5	IN	origin_count	number of entries in origin buffer (non-negative integer)
6			
7	IN	origin_datatype	datatype of each entry in origin buffer (handle)
8			
9	IN	target_rank	rank of target (non-negative integer)
10	IN	target_disp	displacement from start of window to target buffer (non-negative integer)
11			
12	IN	target_count	number of entries in target buffer (non-negative integer)
13			
14	IN	target_datatype	datatype of each entry in target buffer (handle)
15			
16	IN	win	window object used for communication (handle)

18 C binding

```

19 int MPI_Put(const void *origin_addr, int origin_count,
20            MPI_Datatype origin_datatype, int target_rank,
21            MPI_Aint target_disp, int target_count,
22            MPI_Datatype target_datatype, MPI_Win win)

```

```

23
24 int MPI_Put_c(const void *origin_addr, MPI_Count origin_count,
25              MPI_Datatype origin_datatype, int target_rank,
26              MPI_Aint target_disp, MPI_Count target_count,
27              MPI_Datatype target_datatype, MPI_Win win)

```

28 Fortran 2008 binding

```

29 MPI_Put(origin_addr, origin_count, origin_datatype, target_rank, target_disp,
30         target_count, target_datatype, win, ierror)

```

```

31     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
32     INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
33     TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
34     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
35     TYPE(MPI_Win), INTENT(IN) :: win
36     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

37
38 MPI_Put(origin_addr, origin_count, origin_datatype, target_rank, target_disp,
39         target_count, target_datatype, win, ierror) !(_c)

```

```

40     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
41     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
42     TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
43     INTEGER, INTENT(IN) :: target_rank
44     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
45     TYPE(MPI_Win), INTENT(IN) :: win
46     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP,
        TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
<type> ORIGIN_ADDR(*)
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
        TARGET_DATATYPE, WIN, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP

```

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 error 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.*)

12.3.2 Get

`MPI_GET(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win)`

OUT	<code>origin_addr</code>	initial address of origin buffer (choice)
IN	<code>origin_count</code>	number of entries in origin buffer (non-negative integer)
IN	<code>origin_datatype</code>	datatype of each entry in origin buffer (handle)
IN	<code>target_rank</code>	rank of target (non-negative integer)
IN	<code>target_disp</code>	displacement from window start to the beginning of the target buffer (non-negative integer)
IN	<code>target_count</code>	number of entries in target buffer (non-negative integer)
IN	<code>target_datatype</code>	datatype of each entry in target buffer (handle)
IN	<code>win</code>	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)
```

```
int MPI_Get_c(void *origin_addr, MPI_Count origin_count,
              MPI_Datatype origin_datatype, int target_rank,
              MPI_Aint target_disp, MPI_Count target_count,
              MPI_Datatype target_datatype, MPI_Win win)
```

Fortran 2008 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
```

```
MPI_Get(origin_addr, origin_count, origin_datatype, target_rank, target_disp,
        target_count, target_datatype, win, ierror) !(_c)
```

```
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: origin_addr
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_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

```

Fortran binding

```

MPI_GET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP,
        TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
<type> ORIGIN_ADDR(*)
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
        TARGET_DATATYPE, WIN, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP

```

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.

12.3.3 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 12.5.

Example 12.1. We show how to implement the generic indirect assignment $A = B(\text{map})$, 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, i, j, k
INTEGER(KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aaint

! 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

```

```

1 DO i=1,p
2   count(i) = 0
3 END DO
4 DO i=1,m
5   j = map(i)/m+1
6   count(j) = count(j)+1
7 END DO
8 total(1) = 0
9 DO i=2,p
10  total(i) = total(i-1) + count(i-1)
11 END DO
12 DO i=1,p
13  count(i) = 0
14 END DO
15 ! compute origin and target indices of entries.
16 ! entry i at current process is received from location
17 ! k at process (j-1), where map(i) = (j-1)*m + (k-1),
18 ! j = 1..p and k = 1..m
19 DO i=1,m
20  j = map(i)/m+1
21  k = MOD(map(i),m)+1
22  count(j) = count(j)+1
23  oindex(total(j) + count(j)) = i
24  tindex(total(j) + count(j)) = k
25 END DO
26 ! create origin and target datatypes for each get operation
27 DO i=1,p
28   CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, &
29   oindex(total(i)+1:total(i)+count(i)), &
30   MPI_REAL, otype(i), ierr)
31   CALL MPI_TYPE_COMMIT(otype(i), ierr)
32   CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, &
33   tindex(total(i)+1:total(i)+count(i)), &
34   MPI_REAL, ttype(i), ierr)
35   CALL MPI_TYPE_COMMIT(ttype(i), ierr)
36 END DO
37 ! this part does the assignment itself
38 CALL MPI_WIN_FENCE(0, win, ierr)
39 disp_aint = 0
40 DO i=1,p
41   CALL MPI_GET(A, 1, otype(i), i-1, disp_aint, 1, ttype(i), win, ierr)
42 END DO
43 CALL MPI_WIN_FENCE(0, win, ierr)
44 CALL MPI_WIN_FREE(win, ierr)
45 DO i=1,p
46   CALL MPI_TYPE_FREE(otype(i), ierr)
47   CALL MPI_TYPE_FREE(ttype(i), ierr)
48 END DO
49 RETURN
50 END

```


Example 12.2. A simpler version can be written that does not require that a datatype be built for the target buffer. 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, i, j, 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

```

12.3.4 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 12.7 for details.

Accumulate

```

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)

1	IN	target_count	number of entries in target buffer (non-negative integer)
2			
3	IN	target_datatype	datatype of each entry in target buffer (handle)
4			
5	IN	op	accumulate operator (handle)
6	IN	win	window object (handle)
7			

C binding

```

9 int MPI_Accumulate(const void *origin_addr, int origin_count,
10                 MPI_Datatype origin_datatype, int target_rank,
11                 MPI_Aint target_disp, int target_count,
12                 MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)

```

```

14 int MPI_Accumulate_c(const void *origin_addr, MPI_Count origin_count,
15                    MPI_Datatype origin_datatype, int target_rank,
16                    MPI_Aint target_disp, MPI_Count target_count,
17                    MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)

```

Fortran 2008 binding

```

19 MPI_Accumulate(origin_addr, origin_count, origin_datatype, target_rank,
20              target_disp, target_count, target_datatype, op, win, ierror)
21 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
22 INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
23 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
24 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
25 TYPE(MPI_Op), INTENT(IN) :: op
26 TYPE(MPI_Win), INTENT(IN) :: win
27 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

29 MPI_Accumulate_c(origin_addr, origin_count, origin_datatype, target_rank,
30                target_disp, target_count, target_datatype, op, win, ierror)
31 !(_c)
32 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
33 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
34 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
35 INTEGER, INTENT(IN) :: target_rank
36 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
37 TYPE(MPI_Op), INTENT(IN) :: op
38 TYPE(MPI_Win), INTENT(IN) :: win
39 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

41 MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
42              TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
43 <type> ORIGIN_ADDR(*)
44 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
45              TARGET_DATATYPE, OP, WIN, IERROR
46 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP

```

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 operator `op`. This is like `MPI_PUT` except that data is combined into the target area instead of overwriting it.

Any of the predefined operators 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 operator `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 operator, `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 operator `MPI_REPLACE`. Note, however, that `MPI_PUT` and `MPI_ACCUMULATE` have different constraints on concurrent updates. (*End of advice to users.*)

Example 12.3. We want to compute $B(j) = \sum_{\text{map}(i)=j} A(i)$. 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, i, j
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 12.2, except that a call to get has been replaced by a call to accumulate. (Note that, if map is one-to-one, the code computes $B = A(\text{map}^{-1})$, which is the reverse assignment to the one computed in that previous example.) In a similar manner, we can replace in Example 12.1, the call to get by a call to accumulate, thus performing the computation with only one communication between any two processes.

Get Accumulate

It is often useful to have fetch-and-accumulate semantics such that the remote data is returned to the caller before the sent data is accumulated into the remote data. The get and accumulate steps are executed atomically for each basic element in the datatype (see Section 12.7 for details). The predefined operator 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	accumulate operator (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)

int MPI_Get_accumulate_c(const void *origin_addr, MPI_Count origin_count,
                        MPI_Datatype origin_datatype, void *result_addr,
```

```

MPI_Count result_count, MPI_Datatype result_datatype,
int target_rank, MPI_Aint target_disp, MPI_Count target_count,
MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)

```

Fortran 2008 binding

```

MPI_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

```

```

INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
target_count

```

```

TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
target_datatype

```

```

TYPE(*), DIMENSION(..), ASYNCHRONOUS :: result_addr

```

```

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_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
result_count, result_datatype, target_rank, target_disp,
target_count, target_datatype, op, win, ierror) !(_c)

```

```

TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr

```

```

INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, result_count,
target_count

```

```

TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
target_datatype

```

```

TYPE(*), DIMENSION(..), ASYNCHRONOUS :: result_addr

```

```

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

```

Fortran 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 ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR

```

```

INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP

```

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 operator `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,

1 in the target buffer. Likewise, the data copied from target to origin must fit, without
2 truncation, in the result buffer.

3 The origin and result buffers (`origin_addr` and `result_addr`) must be disjoint. Each
4 datatype argument must be a predefined datatype or a derived datatype where all basic
5 components are of the same predefined datatype. All datatype arguments must be con-
6 structed from the same predefined datatype. The operator `op` applies to elements of that
7 predefined type. `target_datatype` must not specify overlapping entries, and the target buffer
8 must fit in the target window or in attached memory in a dynamic window. The operation
9 is executed atomically for each basic datatype; see Section 12.7 for details.

10 Any of the predefined operators for `MPI_REDUCE`, as well as `MPI_NO_OP` or
11 `MPI_REPLACE` can be specified as `op`. User-defined functions cannot be used. A new
12 predefined operator, `MPI_NO_OP`, is defined. It corresponds to the associative function
13 $f(a, b) = a$; i.e., the current value in the target memory is returned in the result buffer at
14 the origin and no operation is performed on the target buffer. If `MPI_NO_OP` is specified
15 as the operator, the `origin_addr`, `origin_count`, and `origin_datatype` arguments are ignored.
16 `MPI_NO_OP` can be used only in `MPI_GET_ACCUMULATE`, `MPI_RGET_ACCUMULATE`,
17 and `MPI_FETCH_AND_OP`. `MPI_NO_OP` cannot be used in `MPI_ACCUMULATE`,
18 `MPI_RACCUMULATE`, or collective reduction operations, such as `MPI_REDUCE` and others.

19 *Advice to users.* `MPI_GET` is similar to `MPI_GET_ACCUMULATE`, with the oper-
20 ator `MPI_NO_OP`. Note, however, that `MPI_GET` and `MPI_GET_ACCUMULATE` have
21 different constraints on concurrent updates. (*End of advice to users.*)
22

23 Fetch and Op

24 The generic functionality of `MPI_GET_ACCUMULATE` might limit the performance of fetch-
25 and-increment or fetch-and-add calls that might be supported by special hardware oper-
26 ations. `MPI_FETCH_AND_OP` thus allows for a fast implementation of a commonly used
27 subset of the functionality of `MPI_GET_ACCUMULATE`.
28

29
30 `MPI_FETCH_AND_OP`(`origin_addr`, `result_addr`, `datatype`, `target_rank`, `target_disp`, `op`, `win`)

31	IN	<code>origin_addr</code>	initial address of buffer (choice)
32	OUT	<code>result_addr</code>	initial address of result buffer (choice)
33	IN	<code>datatype</code>	datatype of the entry in origin, result, and target 34 buffers (handle)
35	IN	<code>target_rank</code>	rank of target (non-negative integer)
36	IN	<code>target_disp</code>	displacement from start of window to beginning of 37 target buffer (non-negative integer)
38	IN	<code>op</code>	accumulate operator (handle)
39	IN	<code>win</code>	window object (handle)

40 C binding

41
42 `int MPI_Fetch_and_op`(`const void *origin_addr`, `void *result_addr`,
43 `MPI_Datatype datatype`, `int target_rank`, `MPI_Aint target_disp`,
44 `MPI_Op op`, `MPI_Win win`)

Fortran 2008 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

```

Fortran binding

```

MPI_FETCH_AND_OP(ORIGIN_ADDR, RESULT_ADDR, DATATYPE, TARGET_RANK, TARGET_DISP,
                 OP, WIN, IERROR)
<type> ORIGIN_ADDR(*), RESULT_ADDR(*)
INTEGER DATATYPE, TARGET_RANK, OP, WIN, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP

```

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 operator `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 operators 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

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	<code>origin_addr</code>	initial address of buffer (choice)
IN	<code>compare_addr</code>	initial address of compare buffer (choice)
OUT	<code>result_addr</code>	initial address of result buffer (choice)
IN	<code>datatype</code>	datatype of the element in all buffers (handle)
IN	<code>target_rank</code>	rank of target (non-negative integer)
IN	<code>target_disp</code>	displacement from start of window to beginning of target buffer (non-negative integer)
IN	<code>win</code>	window object (handle)

C binding

```

1 int MPI_Compare_and_swap(const void *origin_addr, const void *compare_addr,
2     void *result_addr, MPI_Datatype datatype, int target_rank,
3     MPI_Aint target_disp, MPI_Win win)
4
5

```

Fortran 2008 binding

```

6 MPI_Compare_and_swap(origin_addr, compare_addr, result_addr, datatype,
7     target_rank, target_disp, win, ierror)
8     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr,
9     compare_addr
10    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: result_addr
11    TYPE(MPI_Datatype), INTENT(IN) :: datatype
12    INTEGER, INTENT(IN) :: target_rank
13    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
14    TYPE(MPI_Win), INTENT(IN) :: win
15    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16

```

Fortran binding

```

17 MPI_COMPARE_AND_SWAP(ORIGIN_ADDR, COMPARE_ADDR, RESULT_ADDR, DATATYPE,
18     TARGET_RANK, TARGET_DISP, WIN, IERROR)
19     <type> ORIGIN_ADDR(*), COMPARE_ADDR(*), RESULT_ADDR(*)
20     INTEGER DATATYPE, TARGET_RANK, WIN, IERROR
21     INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
22
23

```

24 This function compares one element of type `datatype` in the compare buffer
25 `compare_addr` with the buffer at offset `target_disp` in the target window specified by
26 `target_rank` and `win` and replaces the value at the target with the value in the origin buffer
27 `origin_addr` if the compare buffer and the target buffer are identical. The original value at
28 the target is returned in the buffer `result_addr`. The parameter `datatype` must belong to
29 one of the following categories of predefined datatypes: C integer, Fortran integer, Logical,
30 Multi-language types, or Byte as specified in Section 6.9.2. The origin and result buffers
31 (`origin_addr` and `result_addr`) must be disjoint.

12.3.5 Request-based RMA Communication Operations

34 Request-based RMA communication operations allow the user to associate a request handle
35 with the RMA operations and test or wait for the completion of these requests using the
36 functions described in Section 3.7.3. Request-based RMA operations are only valid within
37 a passive target epoch (see Section 12.5).

38 Upon returning from a completion call in which an RMA operation completes, all fields
39 of the status object, if any, and the results of status query functions (e.g.,
40 `MPI_GET_COUNT`) are undefined with the exception of `MPI_ERROR` if appropriate (see
41 Section 3.2.5). It is valid to mix different request types (e.g., any combination of RMA
42 requests, collective requests, I/O requests, generalized requests, or point-to-point requests)
43 in functions that enable multiple completions (e.g., `MPI_WAITALL`). It is erroneous to call
44 `MPI_REQUEST_FREE` or `MPI_CANCEL` for a request associated with an RMA operation.
45 RMA requests are not persistent.

46 The end of the epoch, or explicit bulk synchronization using `MPI_WIN_FLUSH`,
47 `MPI_WIN_FLUSH_ALL`, `MPI_WIN_FLUSH_LOCAL`, or `MPI_WIN_FLUSH_LOCAL_ALL`, also
48

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	<code>origin_addr</code>	initial address of origin buffer (choice)	8
IN	<code>origin_count</code>	number of entries in origin buffer (non-negative integer)	9
IN	<code>origin_datatype</code>	datatype of each entry in origin buffer (handle)	10
IN	<code>target_rank</code>	rank of target (non-negative integer)	11
IN	<code>target_disp</code>	displacement from start of window to target buffer (non-negative integer)	12
IN	<code>target_count</code>	number of entries in target buffer (non-negative integer)	13
IN	<code>target_datatype</code>	datatype of each entry in target buffer (handle)	14
IN	<code>win</code>	window object used for communication (handle)	15
OUT	<code>request</code>	RMA request (handle)	16

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)
```

```
int MPI_Rput_c(const void *origin_addr, MPI_Count origin_count,
              MPI_Datatype origin_datatype, int target_rank,
              MPI_Aint target_disp, MPI_Count target_count,
              MPI_Datatype target_datatype, MPI_Win win, MPI_Request *request)
```

Fortran 2008 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
```

```
MPI_Rput(origin_addr, origin_count, origin_datatype, target_rank, target_disp,
        target_count, target_datatype, win, request, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
```

```

1     TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
2     INTEGER, INTENT(IN) :: target_rank
3     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
4     TYPE(MPI_Win), INTENT(IN) :: win
5     TYPE(MPI_Request), INTENT(OUT) :: request
6     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

8     MPI_RPUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP,
9             TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST, IERROR)
10    <type> ORIGIN_ADDR(*)
11    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
12            TARGET_DATATYPE, WIN, REQUEST, IERROR
13    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP

```

MPI_RPUT is similar to MPI_PUT (Section 12.3.1), 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.

```

24    MPI_RGET(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count,
25            target_datatype, win, request)

```

26	OUT	origin_addr	initial address of origin buffer (choice)
27	IN	origin_count	number of entries in origin buffer (non-negative integer)
28			
29			
30	IN	origin_datatype	datatype of each entry in origin buffer (handle)
31	IN	target_rank	rank of target (non-negative integer)
32	IN	target_disp	displacement from window start to the beginning of the target buffer (non-negative integer)
33			
34			
35	IN	target_count	number of entries in target buffer (non-negative integer)
36			
37	IN	target_datatype	datatype of each entry in target buffer (handle)
38	IN	win	window object used for communication (handle)
39			
40	OUT	request	RMA request (handle)

C binding

```

43    int MPI_Rget(void *origin_addr, int origin_count, MPI_Datatype origin_datatype,
44                int target_rank, MPI_Aint target_disp, int target_count,
45                MPI_Datatype target_datatype, MPI_Win win, MPI_Request *request)

```

```

46    int MPI_Rget_c(void *origin_addr, MPI_Count origin_count,
47                  MPI_Datatype origin_datatype, int target_rank,
48

```

```

MPI_Aint target_disp, MPI_Count target_count,
MPI_Datatype target_datatype, MPI_Win win, MPI_Request *request)

```

Fortran 2008 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

```

```

MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank, target_disp,
        target_count, target_datatype, win, request, ierror) !(_c)
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: origin_addr
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
INTEGER, INTENT(IN) :: target_rank
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

```

Fortran binding

```

MPI_RGET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP,
        TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST, IERROR)
<type> ORIGIN_ADDR(*)
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
        TARGET_DATATYPE, WIN, REQUEST, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP

```

MPI_RGET is similar to MPI_GET (Section 12.3.2), except that it allocates a communication request object and associates it with the request handle (the argument `request`) that can be used to wait or test for completion. The completion of an MPI_RGET operation indicates that the data is available in the origin buffer. If `origin_addr` points to memory attached to a window, then the data becomes available in the private copy of this window.

```

MPI_RACCUMULATE(origin_addr, origin_count, origin_datatype, target_rank, target_disp,
        target_count, target_datatype, op, win, request)

```

IN	<code>origin_addr</code>	initial address of buffer (choice)
IN	<code>origin_count</code>	number of entries in buffer (non-negative integer)
IN	<code>origin_datatype</code>	datatype of each entry in origin buffer (handle)
IN	<code>target_rank</code>	rank of target (non-negative integer)
IN	<code>target_disp</code>	displacement from start of window to beginning of target buffer (non-negative integer)

1	IN	target_count	number of entries in target buffer (non-negative integer)
2			
3	IN	target_datatype	datatype of each entry in target buffer (handle)
4			
5	IN	op	accumulate operator (handle)
6	IN	win	window object (handle)
7	OUT	request	RMA request (handle)
8			

C binding

```

10 int MPI_Raccumulate(const void *origin_addr, int origin_count,
11                   MPI_Datatype origin_datatype, int target_rank,
12                   MPI_Aint target_disp, int target_count,
13                   MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
14                   MPI_Request *request)
15 
```

```

16 int MPI_Raccumulate_c(const void *origin_addr, MPI_Count origin_count,
17                      MPI_Datatype origin_datatype, int target_rank,
18                      MPI_Aint target_disp, MPI_Count target_count,
19                      MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
20                      MPI_Request *request)
21 
```

Fortran 2008 binding

```

22 MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
23               target_disp, target_count, target_datatype, op, win, request,
24               ierror)
25
26 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
27 INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
28 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
29 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
30 TYPE(MPI_Op), INTENT(IN) :: op
31 TYPE(MPI_Win), INTENT(IN) :: win
32 TYPE(MPI_Request), INTENT(OUT) :: request
33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

34 MPI_Raccumulate_c(origin_addr, origin_count, origin_datatype, target_rank,
35                  target_disp, target_count, target_datatype, op, win, request,
36                  ierror) !(_c)
37
38 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
39 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
40 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
41 INTEGER, INTENT(IN) :: target_rank
42 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
43 TYPE(MPI_Op), INTENT(IN) :: op
44 TYPE(MPI_Win), INTENT(IN) :: win
45 TYPE(MPI_Request), INTENT(OUT) :: request
46 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

47 MPI_RACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
48 
```

```

        TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
        IERROR)
<type> ORIGIN_ADDR(*)
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
        TARGET_DATATYPE, OP, WIN, REQUEST, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP

```

MPI_RACCUMULATE is similar to MPI_ACCUMULATE (Section 12.3.4), except that it allocates a communication request object and associates it with the request handle (the argument request) that can be used to wait or test for completion. The completion of an MPI_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 entries 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	accumulate operator (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)
int MPI_Rget_accumulate_c(const void *origin_addr, MPI_Count origin_count,
        MPI_Datatype origin_datatype, void *result_addr,

```

```

1         MPI_Count result_count, MPI_Datatype result_datatype,
2         int target_rank, MPI_Aint target_disp, MPI_Count target_count,
3         MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
4         MPI_Request *request)

```

Fortran 2008 binding

```

6 MPI_Rget_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
7         result_count, result_datatype, target_rank, target_disp,
8         target_count, target_datatype, op, win, request, ierror)
9
10 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
11 INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
12         target_count
13 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
14         target_datatype
15 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: result_addr
16 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
17 TYPE(MPI_Op), INTENT(IN) :: op
18 TYPE(MPI_Win), INTENT(IN) :: win
19 TYPE(MPI_Request), INTENT(OUT) :: request
20 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

21 MPI_Rget_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
22         result_count, result_datatype, target_rank, target_disp,
23         target_count, target_datatype, op, win, request, ierror) !(_c)
24 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
25 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, result_count,
26         target_count
27 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
28         target_datatype
29 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: result_addr
30 INTEGER, INTENT(IN) :: target_rank
31 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
32 TYPE(MPI_Op), INTENT(IN) :: op
33 TYPE(MPI_Win), INTENT(IN) :: win
34 TYPE(MPI_Request), INTENT(OUT) :: request
35 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

37 MPI_RGET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_ADDR,
38         RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, TARGET_DISP,
39         TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST, IERROR)
40
41 <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
42 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
43         TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
44         IERROR
45 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP

```

46 MPI_RGET_ACCUMULATE is similar to MPI_GET_ACCUMULATE (Section 12.3.4), ex-
47 cept that it allocates a communication request object and associates it with the request
48

handle (the argument `request`) that can be used to wait or test for completion. The completion of an `MPI_RGET_ACCUMULATE` operation indicates that the data is available in the result buffer and the origin buffer is free to be updated. It does not indicate that the operation has been completed at the target window.

12.4 Memory Model

The memory semantics of RMA are best understood by using the concept of *public* and *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 noncoherent, 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. Noncoherent systems, however, need to call RMA functions in order to reflect updates to the 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 noncoherent 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 12.1.

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 12.5.3), load and store operations might observe changes to the memory while they are in progress. The order in which data is written is not specified unless further synchronization is used. This might lead to inconsistent views on memory and programs that assume that a transfer is complete by only checking parts of the message are erroneous. (*End of advice to users.*)

The memory model for a particular RMA window can be determined by accessing the attribute `MPI_WIN_MODEL`. If the memory model is the unified model, the value of this attribute is `MPI_WIN_UNIFIED`; otherwise, the value is `MPI_WIN_SEPARATE`.

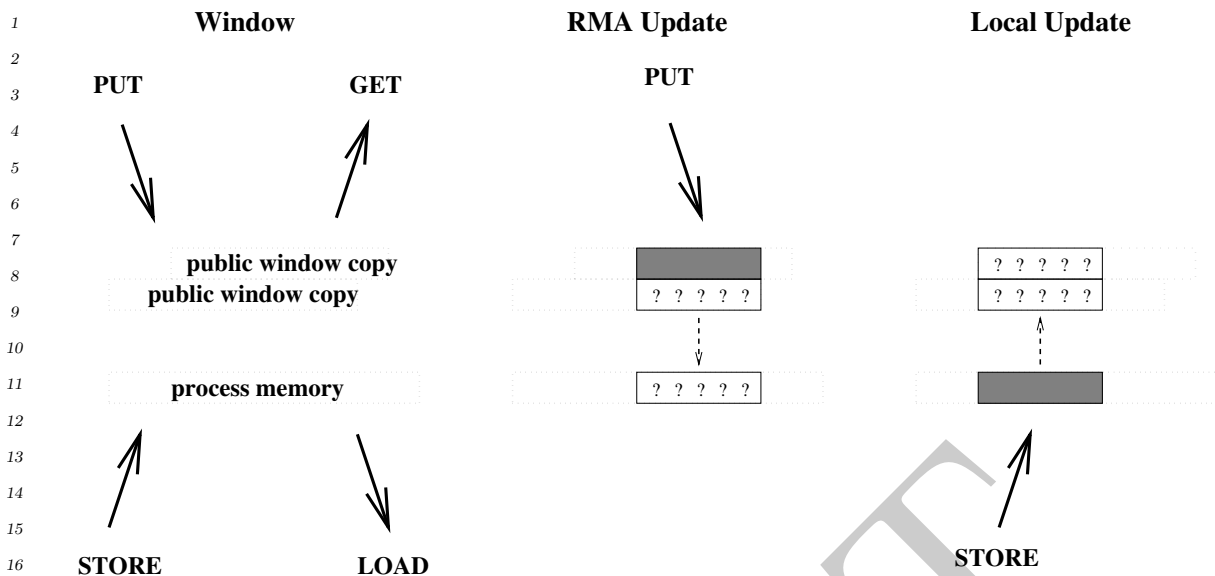


Figure 12.1: Schematic description of the public/private window operations in the MPI_WIN_SEPARATE memory model for two overlapping windows

12.5 Synchronization Calls

RMA communications fall in two categories:

active target communication, where data is moved from the memory of one process to the memory of another, and both are explicitly involved in the communication. This communication pattern is similar to message passing, except that all the data transfer arguments are provided by one process, and the second process only participates in the synchronization.

passive target communication, where data is moved from the memory of one process to the memory of another, and only the origin process is explicitly involved in the transfer. Thus, two origin processes may communicate by accessing the same location in a target window. The process that owns the target window may be distinct from the two communicating processes, in which case it does not participate explicitly in the communication. This communication paradigm is closest to a shared memory model, where shared data can be accessed by all processes, irrespective of location.

RMA communication calls with argument `win` must occur at a process only within an **access epoch** for `win`. Such an epoch starts with an RMA synchronization call on `win`; it proceeds with zero or more RMA communication calls (e.g., `MPI_PUT`, `MPI_GET` or `MPI_ACCUMULATE`) on `win`; it completes with another synchronization call on `win`. This allows users to amortize one synchronization with multiple data transfers and provide implementors more flexibility in the implementation of RMA operations.

Distinct access epochs for `win` at the same process must be disjoint. On the other hand, epochs pertaining to different `win` arguments may overlap. Local operations or other MPI calls may also occur during an epoch.

In active target communication, a target window can be accessed by RMA operations only within an **exposure epoch**. Such an epoch is started and completed by RMA synchronization calls executed by the target process. Distinct exposure epochs at a process on

the same window must be disjoint, but such an exposure epoch may overlap with exposure epochs on other windows or with access epochs for the same or other win arguments. There is a one-to-one matching between access epochs at origin processes and exposure epochs on target processes: RMA operations issued by an origin process for a target window will access that target window during the same exposure epoch if and only if they were issued during the same access epoch.

In passive target communication the target process does not execute RMA synchronization calls, and there is no concept of an exposure epoch.

MPI provides three synchronization mechanisms:

1. 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 “bulletin board” model, where processes can, at random times, access or update different parts of the bulletin board.

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 12.2 illustrates the general synchronization pattern for active target communication. The synchronization between `post` and `start` ensures that the `put` call of the origin

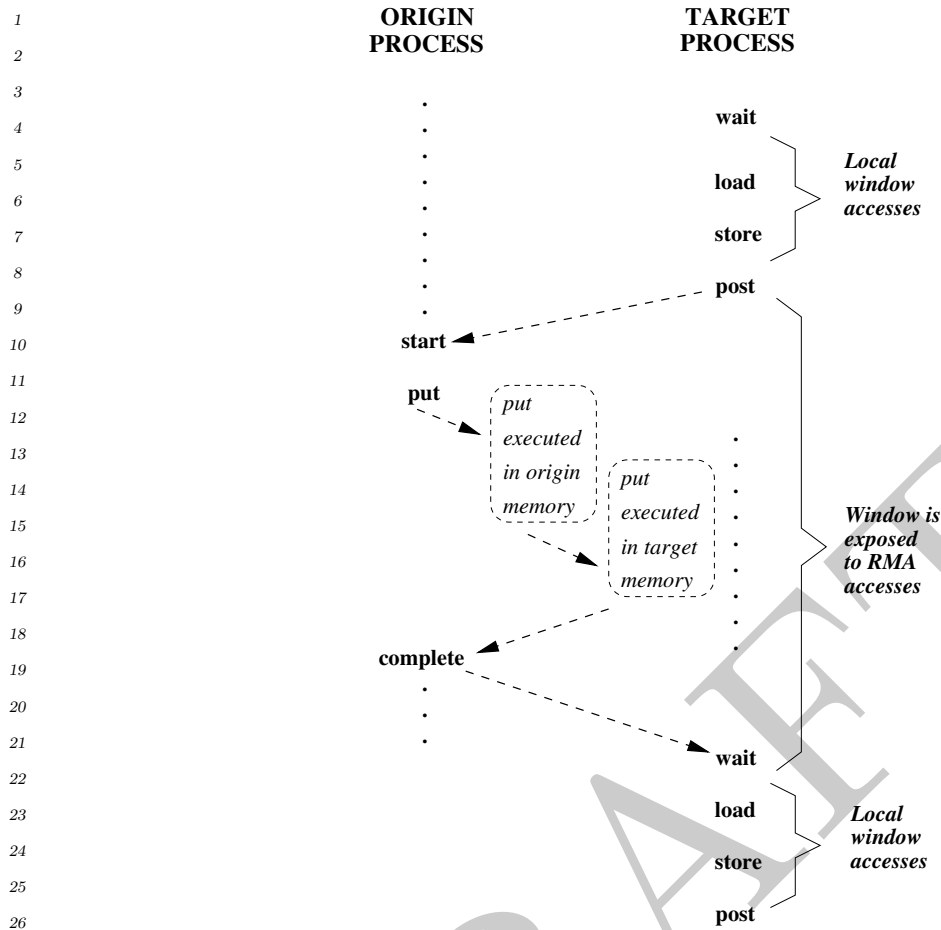


Figure 12.2: Active target communication. Dashed arrows represent synchronizations (ordering of events).

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 12.2 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 12.3. The access to the target window is delayed until the window is exposed, after the `post`. However the `start` may complete earlier; the `put` and `complete` may also terminate earlier, if `put` data is buffered by the implementation. The synchronization calls order correctly window accesses, but do not necessarily synchronize other operations. This weaker synchronization semantic allows for more efficient implementations.

Figure 12.4 illustrates the general synchronization pattern for passive target communication. The first origin process communicates data to the second origin process, through the memory of the target process; the target process is not explicitly involved in the com-

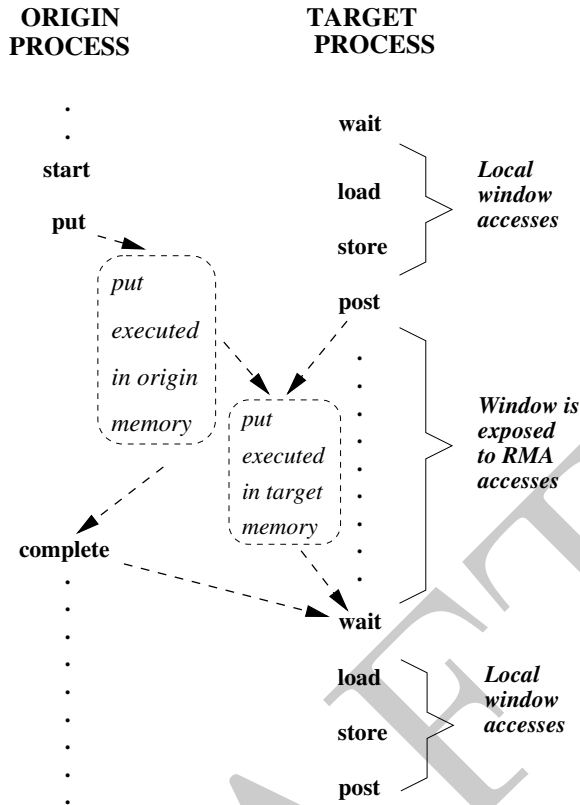


Figure 12.3: Active target communication, with weak synchronization. Dashed arrows represent synchronizations (ordering of events).

munication. The lock and unlock calls ensure that the two RMA accesses do not occur concurrently. However, they do *not* ensure that the put by origin 1 will precede the get by origin 2.

Rationale. RMA does not define fine-grained mutexes in memory (only logical 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.*)

12.5.1 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)

Fortran 2008 binding

MPI_Win_fence(assert, win, ierror)

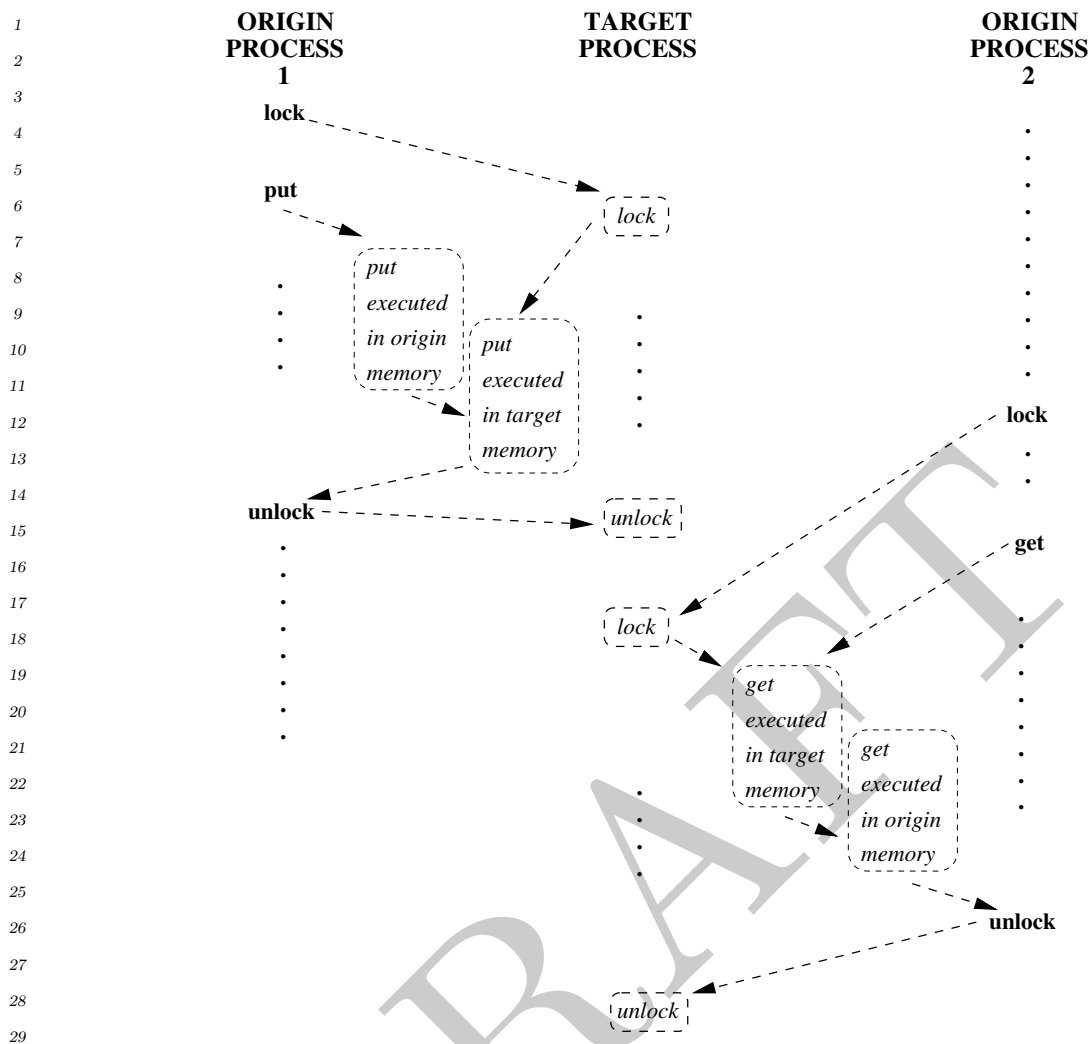


Figure 12.4: Passive target communication. Dashed arrows represent synchronizations (ordering of events).

```

34 INTEGER, INTENT(IN) :: assert
35 TYPE(MPI_Win), INTENT(IN) :: win
36 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
    
```

Fortran binding

```

38 MPI_WIN_FENCE(ASSERT, WIN, IERROR)
39     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 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 12.5.5. 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.*)

12.5.2 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)
```

Fortran 2008 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
```

Fortran 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 12.5.5. 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)
```

Fortran 2008 binding

```
MPI_Win_complete(win, ierror)
  TYPE(MPI_Win), INTENT(IN) :: win
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_WIN_COMPLETE(WIN, IERROR)
  INTEGER WIN, IERROR
```

Completes an RMA access epoch on `win` started by a call to `MPI_WIN_START`. All RMA communication calls issued on `win` during this epoch will have completed at the origin when the call returns.

`MPI_WIN_COMPLETE` enforces completion of preceding RMA calls at the origin, but not at the target. A put or accumulate call may not have completed at the target when it has completed at the origin.

Consider the sequence of calls in the example below.

Example 12.4. Use of `MPI_WIN_START` and `MPI_WIN_COMPLETE`.

```
MPI_Win_start(group, flag, win);
MPI_Put(..., win);
MPI_Win_complete(win);
```

The call to `MPI_WIN_COMPLETE` does not return until the put call has completed at the origin; and the target window will be accessed by the put operation only after the call to `MPI_WIN_START` has matched a call to `MPI_WIN_POST` by the target process. This still leaves much choice to implementors. The call to `MPI_WIN_START` can block until the matching call to `MPI_WIN_POST` occurs at all target processes. One can also have implementations where the call to `MPI_WIN_START` is nonblocking, but the call to `MPI_PUT` blocks until the matching call to `MPI_WIN_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)	1
IN	assert	program assertion (integer)	2
IN	win	window object (handle)	3
			4
			5

C binding

```
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
```

Fortran 2008 binding

```
MPI_Win_post(group, assert, win, ierror)
  TYPE(MPI_Group), INTENT(IN) :: group
  INTEGER, INTENT(IN) :: assert
  TYPE(MPI_Win), INTENT(IN) :: win
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_WIN_POST(GROUP, ASSERT, WIN, IERROR)
  INTEGER GROUP, ASSERT, WIN, IERROR
```

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)	23
			24
			25
			26

C binding

```
int MPI_Win_wait(MPI_Win win)
```

Fortran 2008 binding

```
MPI_Win_wait(win, ierror)
  TYPE(MPI_Win), INTENT(IN) :: win
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran 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 12.5 illustrates the use of these four functions. Process 0 puts data in the windows of processes 1 and 2 and process 3 puts data in the window of process 2. Each start call lists the ranks of the processes whose windows will be accessed; each post call lists the ranks of the processes that access the local window. The figure illustrates a possible

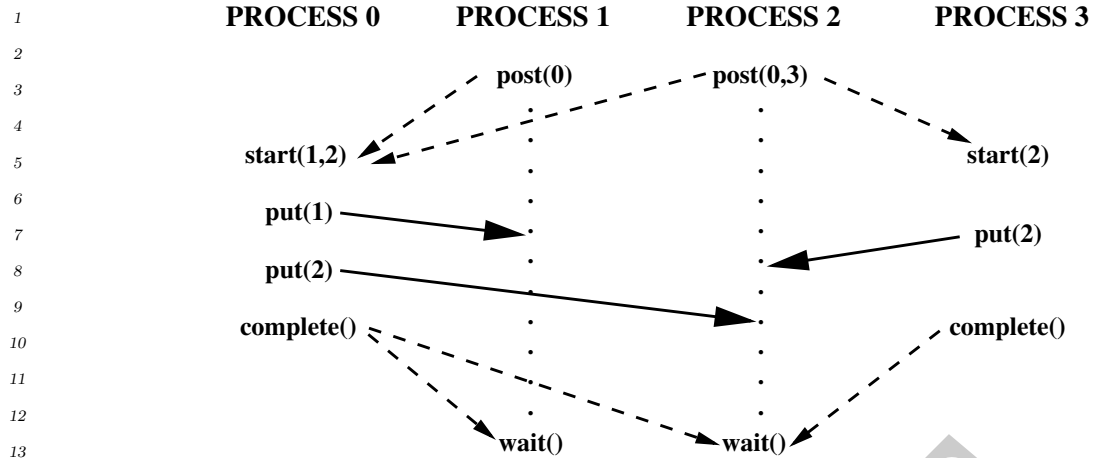


Figure 12.5: Active target communication. Dashed arrows represent synchronizations and solid arrows represent data transfer.

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)
```

Fortran 2008 binding

```
MPI_Win_test(win, flag, ierror)
  TYPE(MPI_Win), INTENT(IN) :: win
  LOGICAL, INTENT(OUT) :: flag
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_WIN_TEST(WIN, FLAG, IERROR)
  INTEGER WIN, IERROR
  LOGICAL FLAG
```

MPI_WIN_TEST is a local procedure. Repeated calls to **MPI_WIN_TEST** with the same **win** argument will eventually return **flag = true** once 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 indicated 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 $V = \{0, \dots, n - 1\}$ 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(ingroupi, ...)`, followed by a call to `MPI_WIN_START(outgroupi, ...)`, where $outgroup_i = \{j : ij \in E\}$ and $ingroup_i = \{j : ji \in E\}$. A call is a noop, and can be skipped, if the `group` argument is empty. After the communications calls, each process that issued a start will issue a complete. Finally, each process that issued a post will issue a wait.

Note that each process may call with a `group` argument that has different members. (*End of advice to users.*)

12.5.3 Lock

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

1 that are protected by an **exclusive lock** (acquired using `MPI_LOCK_EXCLUSIVE`) will not
 2 be concurrent at the window site with other accesses to the same window that are lock
 3 protected. Accesses that are protected by a **shared lock** (acquired using
 4 `MPI_LOCK_SHARED`) will not be concurrent at the window site with accesses protected by
 5 an *exclusive lock* to the same window.
 6

7
 8 `MPI_WIN_LOCK(lock_type, rank, assert, win)`

9	IN	lock_type	either <code>MPI_LOCK_EXCLUSIVE</code> or 10 <code>MPI_LOCK_SHARED</code> (state)
11	IN	rank	rank of locked window (non-negative integer)
12	IN	assert	program assertion (integer)
13	IN	win	window object (handle)
14	IN	win	window object (handle)

15
 16 **C binding**

17 `int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)`

18
 19 **Fortran 2008 binding**

20 `MPI_Win_lock(lock_type, rank, assert, win, ierror)`
 21 `INTEGER, INTENT(IN) :: lock_type, rank, assert`
 22 `TYPE(MPI_Win), INTENT(IN) :: win`
 23 `INTEGER, OPTIONAL, INTENT(OUT) :: ierror`

24 **Fortran binding**

25 `MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR)`
 26 `INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR`

27
 28 Starts an RMA access epoch. The window at the process with rank `rank` can be accessed
 29 by RMA operations on `win` during that epoch. Multiple RMA access epochs (with calls
 30 to `MPI_WIN_LOCK`) can occur simultaneously; however, each access epoch must target a
 31 different process.
 32

33
 34 `MPI_WIN_LOCK_ALL(assert, win)`

35	IN	assert	program assertion (integer)
36	IN	win	window object (handle)

37
 38 **C binding**

39 `int MPI_Win_lock_all(int assert, MPI_Win win)`

40
 41 **Fortran 2008 binding**

42 `MPI_Win_lock_all(assert, win, ierror)`
 43 `INTEGER, INTENT(IN) :: assert`
 44 `TYPE(MPI_Win), INTENT(IN) :: win`
 45 `INTEGER, OPTIONAL, INTENT(OUT) :: ierror`

46 **Fortran binding**

47 `MPI_WIN_LOCK_ALL(ASSERT, WIN, IERROR)`
 48

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 12.5.5). (*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)
```

Fortran 2008 binding

```
MPI_Win_unlock(rank, win, ierror)
  INTEGER, INTENT(IN) :: rank
  TYPE(MPI_Win), INTENT(IN) :: win
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran 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)
```

Fortran 2008 binding

```
MPI_Win_unlock_all(win, ierror)
  TYPE(MPI_Win), INTENT(IN) :: win
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_WIN_UNLOCK_ALL(WIN, IERROR)
```

1 INTEGER WIN, IERROR

2
3 Completes a shared RMA access epoch started by a call to `MPI_WIN_LOCK_ALL` on
4 window `win`. RMA operations issued during this epoch will have completed both at the
5 origin and at the target when the call returns.

6
7 It is erroneous to have a window locked and exposed (in an exposure epoch) concu-
8 rrently. For example, a process may not call `MPI_WIN_LOCK` to lock a target window if
9 the target process has called `MPI_WIN_POST` and has not yet called `MPI_WIN_WAIT`; it is
10 erroneous to call `MPI_WIN_POST` while the local window is locked.

11 *Rationale.* An alternative is to require MPI to enforce mutual exclusion between
12 exposure epochs and locking periods. But this would entail additional overheads
13 when locks or active target synchronization do not interact in support of those rare
14 interactions between the two mechanisms. The programming style that we encourage
15 here is that a set of windows is used with only one synchronization mechanism at
16 a time, with shifts from one mechanism to another being rare and involving global
17 synchronization. (*End of rationale.*)

18
19 *Advice to users.* Users need to use explicit synchronization code in order to enforce
20 mutual exclusion between locking periods and exposure epochs on a window. (*End of*
21 *advice to users.*)

22
23 Implementors may restrict the use of RMA communication that is synchronized by
24 lock calls to windows in memory allocated by `MPI_ALLOC_MEM` (Section 9.2),
25 `MPI_WIN_ALLOCATE` (Section 12.2.2), `MPI_WIN_ALLOCATE_SHARED` (Section 12.2.3),
26 or attached with `MPI_WIN_ATTACH` (Section 12.2.4). Locks can be used portably only in
27 such memory.

28 *Rationale.* The implementation of passive target communication when memory
29 is not shared may require an asynchronous software agent. Such an agent can be
30 implemented more easily, and can achieve better performance, if restricted to specially
31 allocated memory. It can be avoided altogether if shared memory is used. It seems
32 natural to impose restrictions that allows one to use shared memory for third party
33 communication in shared memory machines.

34 (*End of rationale.*)

35
36 Consider the sequence of calls in the example below.

37
38 **Example 12.5.** Use of `MPI_WIN_LOCK` and `MPI_WIN_UNLOCK`.

```
39 MPI_Win_lock(MPI_LOCK_EXCLUSIVE, rank, assert, win);
40 MPI_Put(..., rank, ..., win);
41 MPI_Win_unlock(rank, win);
```

42 The call to `MPI_WIN_UNLOCK` will not return until the put transfer has completed at
43 the origin and at the target. This still leaves much freedom to implementors. The call
44 to `MPI_WIN_LOCK` may block until an *exclusive lock* on the window is acquired; or, the
45 first two calls may not block, while `MPI_WIN_UNLOCK` blocks until a lock is acquired—the
46 update of the target window is then postponed until the call to `MPI_WIN_UNLOCK` occurs.
47 However, if the call to `MPI_WIN_LOCK` is used to lock a local window, then the call must
48

block until the lock is acquired, since the lock may protect local load/store accesses to the window issued after the lock call returns.

12.5.4 Flush and Sync

All flush and sync functions can be called only within passive target epochs.

MPI_WIN_FLUSH(rank, win)

IN	rank	rank of target window (non-negative integer)
IN	win	window object (handle)

C binding

```
int MPI_Win_flush(int rank, MPI_Win win)
```

Fortran 2008 binding

```
MPI_Win_flush(rank, win, ierror)
  INTEGER, INTENT(IN) :: rank
  TYPE(MPI_Win), INTENT(IN) :: win
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran 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)
```

Fortran 2008 binding

```
MPI_Win_flush_all(win, ierror)
  TYPE(MPI_Win), INTENT(IN) :: win
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran 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.

```

1 MPI_WIN_FLUSH_LOCAL(rank, win)
2     IN      rank                rank of target window (non-negative integer)
3
4     IN      win                  window object (handle)
5

```

C binding

```

7 int MPI_Win_flush_local(int rank, MPI_Win win)
8

```

Fortran 2008 binding

```

9 MPI_Win_flush_local(rank, win, ierror)
10     INTEGER, INTENT(IN) :: rank
11     TYPE(MPI_Win), INTENT(IN) :: win
12     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13

```

Fortran binding

```

14 MPI_WIN_FLUSH_LOCAL(RANK, WIN, IERROR)
15     INTEGER RANK, WIN, IERROR
16

```

17 Locally completes at the origin all outstanding RMA operations initiated by the calling
18 process to the target process specified by rank on the specified window. For example, after
19 this routine completes, the user may reuse any buffers provided to put, get, or accumulate
20 operations.
21

```

22
23 MPI_WIN_FLUSH_LOCAL_ALL(win)
24     IN      win                  window object (handle)
25
26

```

C binding

```

27 int MPI_Win_flush_local_all(MPI_Win win)
28

```

Fortran 2008 binding

```

29 MPI_Win_flush_local_all(win, ierror)
30     TYPE(MPI_Win), INTENT(IN) :: win
31     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33

```

Fortran binding

```

34 MPI_WIN_FLUSH_LOCAL_ALL(WIN, IERROR)
35     INTEGER WIN, IERROR
36

```

37 All RMA operations issued to any target prior to this call in this window will have
38 completed at the origin when MPI_WIN_FLUSH_LOCAL_ALL returns.
39

```

40
41 MPI_WIN_SYNC(win)
42     IN      win                  window object (handle)
43
44

```

C binding

```

45 int MPI_Win_sync(MPI_Win win)
46
47
48

```

Fortran 2008 binding

```

MPI_Win_sync(win, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

MPI_WIN_SYNC(WIN, IERROR)
    INTEGER WIN, IERROR

```

For windows in the separate memory model, a call to `MPI_WIN_SYNC` synchronizes the private and public window copies of `win` at the calling MPI process, as described in Section 12.7.

In the unified memory model, `MPI_WIN_SYNC` may be used to order load and store accesses to shared memory and to ensure visibility of store updates in shared memory for other threads and MPI processes.

A call to `MPI_WIN_SYNC` does not open or close an epoch and does not complete any pending RMA operations. A call to `MPI_WIN_SYNC` does not guarantee *progress* of any pending MPI operation.

12.5.5 Assertions

The `assert` argument in the calls `MPI_WIN_POST`, `MPI_WIN_START`, `MPI_WIN_FENCE`, `MPI_WIN_LOCK`, and `MPI_WIN_LOCK_ALL` is used to provide assertions on the context of 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.*)

12.5.6 Miscellaneous Clarifications

Once an RMA routine completes, it is safe to free any opaque objects passed as arguments to that routine. For example, the datatype argument of a MPI_PUT call can be freed as soon as the call returns, even though the communication may not be complete.

Table 12.2: Error classes in one-sided communication routines		1
MPI_ERR_WIN	invalid win argument	2
MPI_ERR_BASE	invalid base argument	3
MPI_ERR_SIZE	invalid size argument	4
MPI_ERR_DISP	invalid disp argument	5
MPI_ERR_LOCKTYPE	invalid locktype argument	6
MPI_ERR_ASSERT	invalid assert argument	7
MPI_ERR_RMA_CONFLICT	conflicting accesses to window	8
MPI_ERR_RMA_SYNC	invalid synchronization of RMA calls	9
MPI_ERR_RMA_RANGE	target memory is not part of the window (in the case of a window created with MPI_WIN_CREATE_DYNAMIC, target memory is not attached)	10
MPI_ERR_RMA_ATTACH	memory cannot be attached (e.g., because of resource exhaustion)	11
MPI_ERR_RMA_SHARED	memory cannot be shared (e.g., some process in the group of the specified communicator cannot expose shared memory)	12
MPI_ERR_RMA_FLAVOR	passed window has the wrong flavor for the called function	13
		14
		15
		16
		17
		18
		19
		20
		21
		22
		23
		24
		25
		26
		27
		28
		29
		30
		31
		32
		33
		34
		35
		36
		37
		38
		39
		40
		41
		42
		43
		44
		45
		46
		47
		48

As in message-passing, datatypes must be committed before they can be used in RMA communication.

12.6 Error Handling

12.6.1 Error Handlers

Errors occurring during calls to routines that create MPI windows (e.g., MPI_WIN_CREATE (...comm,...)) cause the error handler currently associated with `comm` to be invoked. All other RMA calls have an input `win` argument. When an error occurs during such a call, the error handler currently associated with `win` is invoked.

The error handler `MPI_ERRORS_ARE_FATAL` is associated with `win` during its creation. Users may change this default by explicitly associating a new error handler with `win` (see Section 9.3).

12.6.2 Error Classes

The error classes for one-sided communication are defined in Table 12.2. RMA routines may (and almost certainly will) use other MPI error classes, such as `MPI_ERR_OP` or `MPI_ERR_RANK`.

12.7 Semantics and Correctness

The following rules specify the latest time at which an operation must complete at the origin or the target. The update performed by a get call in the origin process memory is

1 visible when the get operation is complete at the origin (or earlier); the update performed
2 by a put or accumulate call in the public copy of the target window is visible when the put
3 or accumulate has completed at the target (or earlier). The rules also specify the latest
4 time at which an update of one window copy becomes visible in another overlapping copy.

- 5
6 1. An RMA operation is completed at the origin by the ensuing call to
7 MPI_WIN_COMPLETE, MPI_WIN_FENCE, MPI_WIN_FLUSH,
8 MPI_WIN_FLUSH_ALL, MPI_WIN_FLUSH_LOCAL, MPI_WIN_FLUSH_LOCAL_ALL,
9 MPI_WIN_UNLOCK, or MPI_WIN_UNLOCK_ALL that synchronizes this access at the
10 origin.
- 11
12 2. If an RMA operation is completed at the origin by a call to MPI_WIN_FENCE then
13 the operation is completed at the target by the matching call to MPI_WIN_FENCE by
14 the target process.
- 15
16 3. If an RMA operation is completed at the origin by a call to MPI_WIN_COMPLETE
17 then the operation is completed at the target by the matching call to MPI_WIN_WAIT
18 by the target process.
- 19
20 4. If an RMA operation is completed at the origin by a call to MPI_WIN_UNLOCK,
21 MPI_WIN_UNLOCK_ALL, MPI_WIN_FLUSH(rank=target), or MPI_WIN_FLUSH_ALL,
22 then the operation is completed at the target by that same call.
- 23
24 5. An update of a location in a private window copy in process memory becomes visible
25 in the public window copy at latest when an ensuing call to MPI_WIN_POST,
26 MPI_WIN_FENCE, MPI_WIN_UNLOCK, MPI_WIN_UNLOCK_ALL, or
27 MPI_WIN_SYNC is executed on that window by the window owner. In the RMA
28 unified memory model, an update of a location in a private window in process memory
29 becomes visible without additional RMA calls.
- 30
31 6. An update by a put or accumulate call to a public window copy becomes visible in
32 the private copy in process memory at latest when an ensuing call to
33 MPI_WIN_WAIT, MPI_WIN_FENCE, MPI_WIN_LOCK, MPI_WIN_LOCK_ALL, or
34 MPI_WIN_SYNC is executed on that window by the window owner. In the RMA
35 unified memory model, an update by a put or accumulate call to a public window copy
36 eventually becomes visible in the private copy in process memory without additional
37 RMA calls.

38 The MPI_WIN_FENCE or MPI_WIN_WAIT call that completes the transfer from public
39 copy to private copy (6) is the same call that completes the put or accumulate operation in
40 the window copy (2, 3). If a put or accumulate access was synchronized with a lock, then
41 the update of the public window copy is complete as soon as the updating process executed
42 MPI_WIN_UNLOCK or MPI_WIN_UNLOCK_ALL. In the RMA separate memory model, the
43 update of a private copy in the process memory may be delayed until the target process
44 executes a synchronization call on that window (6). Thus, updates to process memory can
45 always be delayed in the RMA separate memory model until the process executes a suitable
46 synchronization call, while they must complete in the RMA unified model without additional
47 synchronization calls. If fence or post-start-complete-wait synchronization is used, updates
48 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 raise an error. Thus, user programs or tools that used MPI RMA could not portably permit such operations, even if the application code could function correctly with such an undefined result. Starting with MPI-3, these operations are not erroneous, but do not have a defined behavior.

Rationale. As discussed in [7], 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 starting with MPI-3, such operations must not raise an error. (*End of advice to implementors.*)

A program with a well-defined outcome in the `MPI_WIN_SEPARATE` memory model must obey the following rules.

- S1. A location in a window must not be accessed with load/store operations once an update to that location has started, until the update becomes visible in the private window copy in process memory.
- S2. A location in a window must not be accessed as a target of an RMA operation once an update to that location has started, until the update becomes visible in the public window copy. There is one exception to this rule, in the case where the same variable is updated by two concurrent accumulates with the same predefined datatype, on the same window. Additional restrictions on the operation apply, see the info key `accumulate_ops` in Section 12.2.1.
- 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

1 the target window, until the update becomes visible in the public copy of the win-
2 dow. Conversely, a store to process memory to a location in a window must not start
3 once a put or accumulate update to that target window has started, until the put or
4 accumulate update becomes visible in process memory. In both cases, the restriction
5 applies to operations even if they access disjoint locations in the window.
6

7 *Rationale.* The last constraint on correct RMA accesses may seem unduly restric-
8 tive, as it forbids concurrent accesses to nonoverlapping locations in a window. The
9 reason for this constraint is that, on some architectures, explicit coherence restor-
10 ing operations may be needed at synchronization points. A different operation may
11 be needed for locations that were updated by stores and for locations that were re-
12 motely updated by put or accumulate operations. Without this constraint, the MPI
13 library would have to track precisely which locations in a window were updated by a
14 put or accumulate call. The additional overhead of maintaining such information is
15 considered prohibitive. (*End of rationale.*)
16

17 Note that `MPI_WIN_SYNC` may be used within a passive target epoch to synchronize
18 the private and public window copies (that is, updates to one are made visible to the other).

19 In the `MPI_WIN_UNIFIED` memory model, the rules are simpler because the public and
20 private windows are the same. However, there are restrictions to avoid concurrent access
21 to the same memory locations by different processes. The rules that a program with a
22 well-defined outcome must obey in this case are:

- 23 U1. A location in a window must not be accessed with load/store operations once an
24 update to that location has started, until the update is complete, subject to the
25 following special case.
26
- 27 U2. Accessing a location in the window that is also the target of a remote update is valid
28 (not erroneous) but the precise result will depend on the behavior of the implemen-
29 tation. Updates from a remote process will appear in the memory of the target, but
30 there are no atomicity or ordering guarantees if more than one byte is updated. Up-
31 dates are stable in the sense that once data appears in memory of the target, the data
32 remains until replaced by another update. This permits polling on a location for a
33 change from zero to nonzero or for a particular value, but not polling and comparing
34 the relative magnitude of values. Users are cautioned that polling on one memory
35 location and then accessing a different memory location has defined behavior only if
36 the other rules given here and in this chapter are followed.
37

38 *Advice to users.* Some compiler optimizations can result in code that maintains
39 the sequential semantics of the program, but violates this rule by introducing
40 temporary values into locations in memory. Most compilers only apply such
41 transformations under very high levels of optimization and users should be aware
42 that such aggressive optimization may produce unexpected results. (*End of*
43 *advice to users.*)

- 44 U3. Updating a location in the window with a store operation that is also the target
45 of a remote read (but not update) is valid (not erroneous) but the precise result
46 will depend on the behavior of the implementation. Store updates will appear in
47 memory, but there are no atomicity or ordering guarantees if more than one byte is
48 updated. Updates are stable in the sense that once data appears in memory, the data

remains until replaced by another update. This permits updates to memory with store operations without requiring an RMA epoch. Users are cautioned that remote accesses to a window that is updated by the local process has defined behavior only if the other rules given here and elsewhere in this chapter are followed.

- U4. A location in a window must not be accessed as a target of an RMA operation once an update to that location has started and until the update completes at the target. There is one exception to this rule: in the case where the same location is updated by two concurrent accumulates with the same predefined datatype on the same window. Additional restrictions on the operation apply; see the info key `accumulate_ops` in Section 12.2.1.
- U5. A put or accumulate must not access a target window once a store, put, or accumulate update to another (overlapping) target window has started on the same location in the target window and until the update completes at the target window. Conversely, a store operation to a location in a window must not start once a put or accumulate update to the same location in that target window has started and until the put or accumulate update completes at the target.

Advice to users. In the unified memory model, in the case where the window is in shared memory, `MPI_WIN_SYNC` can be used to order store operations and make store updates to the window visible to other processes and threads. Use of this 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 both the reader and the writer of a shared memory variable between any non-RMA synchronization and access to that variable, as shown in Example 12.22. The calls to `MPI_WIN_SYNC` can be replaced by language level memory synchronization operations, if available. (*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 12.6. The following example demonstrates updating a memory location inside a window for the separate memory model, according to Rule 5. The `MPI_WIN_LOCK` 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
	<code>MPI_Win_lock(EXCLUSIVE, B)</code>
	store X /* local update to private copy of B */
	<code>MPI_Win_unlock(B)</code>
	/* now visible in public window copy */
<code>MPI_Barrier</code>	<code>MPI_Barrier</code>
<code>MPI_Win_lock(EXCLUSIVE, B)</code>	
<code>MPI_Get(X) /* ok, read from public window */</code>	
<code>MPI_Win_unlock(B)</code>	

Example 12.7. In the RMA unified model, although the public and private copies of the windows are synchronized, caution must be used when combining load/stores and 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 */
<code>MPI_Barrier</code>	<code>MPI_Barrier</code>

```

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 12.6, also ensures the correct result.

Example 12.8. The following example demonstrates the reading of a memory location updated by a remote process (Rule 6) in the RMA separate memory model. Although the MPI_WIN_UNLOCK on process A and the MPI_BARRIER ensure that the public copy on process B reflects the updated value of *X*, the call to MPI_WIN_LOCK by process B is necessary to synchronize the private copy with the public copy.

Process A	Process B
	window location <i>X</i>
MPI_Win_lock(EXCLUSIVE, B)	
MPI_Put(<i>X</i>) /* 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 <i>X</i>
	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 12.9. Similar to Example 12.7, 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 <i>X</i>
MPI_Win_lock_all	
MPI_Put(<i>X</i>) /* update to window */	

```

1 MPI_Win_flush(B)
2
3 MPI_Barrier MPI_Barrier
4 load X /* may return an obsolete value */
5 MPI_Win_unlock_all
6

```

Example 12.10. 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 12.11. Similar to the previous example, Rule 5 can have unexpected implications for general active target synchronization with the RMA separate memory model. It is *not* guaranteed that process B reads the value of X as per the local update by process A, because neither MPI_WIN_WAIT nor MPI_WIN_COMPLETE calls by process A ensure visibility in the public window copy.

Process 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 12.12. 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.

12.7.1 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 12.2.1. Concurrent accumulate operations with different origin and target pairs are not ordered. Thus, there is no guarantee that the entire call to an accumulate operation is 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

1 extent that computer arithmetics are not truly associative or commutative. The outcome
2 of accumulate operations with overlapping types of different sizes or target displacements
3 is undefined.
4

5 12.7.2 Ordering 6

7 Accumulate calls enable element-wise atomic read and write to remote memory locations.
8 MPI specifies ordering between accumulate operations from an origin process to the same
9 (or overlapping) memory locations at a target process on a per-datatype granularity. The
10 default ordering is strict ordering, which guarantees that overlapping updates from the
11 same origin to a remote location are committed in program order and that reads (e.g., with
12 `MPI_GET_ACCUMULATE`) and writes (e.g., with `MPI_ACCUMULATE`) are executed and
13 committed in program order. Ordering only applies to operations originating at the same
14 origin that access overlapping target memory regions. MPI does not provide any guarantees
15 for accesses or updates from different origin processes to overlapping target memory regions.

16 The default strict ordering may incur a significant performance penalty. MPI specifies
17 the info key "accumulate_ordering" to allow relaxation of the ordering semantics when specified
18 to any window creation function. The values for this key are as follows. If set to "none",
19 then no ordering will be guaranteed for accumulate calls. This was the behavior for RMA in
20 MPI-2 but has *not* been the default since MPI-3. The key can be set to a comma-separated
21 list of required access orderings at the target. Allowed values in the comma-separated list
22 are "rar", "war", "raw", and "waw" for read-after-read, write-after-read, read-after-write, and
23 write-after-write ordering, respectively. These indicate whether operations of the specified
24 type complete in the order they were issued. For example, "raw" means that any writes must
25 complete at the target before subsequent reads. These ordering requirements apply only to
26 operations issued by the same origin process and targeting the same target process. The
27 default value for "accumulate_ordering" is "rar,raw,war,waw", which implies that writes complete
28 at the target in the order in which they were issued, reads complete at the target before any
29 writes that are issued after the reads, and writes complete at the target before any reads
30 that are issued after the writes. Any subset of these four orderings can be specified. For
31 example, if only read-after-read and write-after-write ordering is required, then the value of
32 the "accumulate_ordering" key could be set to "rar,waw". The order of values is not significant.

33 Note that the above ordering semantics apply only to accumulate operations, not put
34 and get. Put and get within an epoch are unordered.
35

36 12.7.3 Progress 37

38 One-sided communication has the same progress requirements as point-to-point communi-
39 cation: once a communication is enabled it is guaranteed to complete. RMA calls must have
40 local semantics, except when required for synchronization with other RMA calls.

41 There is some fuzziness in the definition of the time when a RMA communication
42 becomes enabled. This fuzziness provides to the implementor more flexibility than with
43 point-to-point communication. Access to a target window becomes enabled once the corre-
44 sponding synchronization (such as `MPI_WIN_FENCE` or `MPI_WIN_POST`) has executed. On
45 the origin process, an RMA communication may become enabled as soon as the correspond-
46 ing put, get or accumulate call has executed, or as late as when the ensuing synchronization
47 call is issued. Once the communication is enabled both at the origin and at the target, the
48 communication must complete.

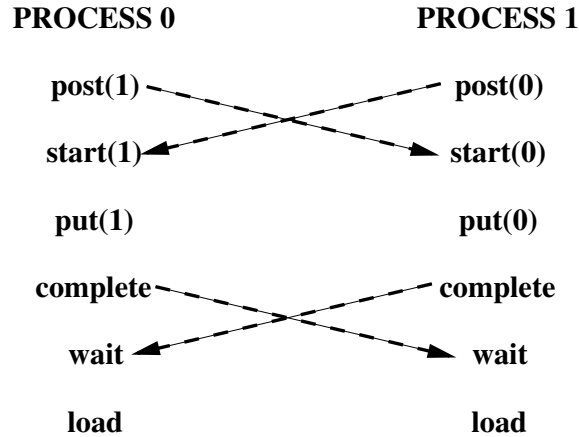


Figure 12.6: Symmetric communication

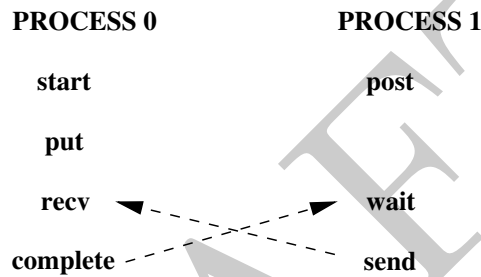


Figure 12.7: Deadlock situation

Consider the code fragment in Example 12.4. Some of the calls may block if the target window is not posted. However, if the target window is posted, then the code fragment must complete. The data transfer may start as soon as the put call occurs, but may be delayed until the ensuing complete call occurs.

Consider the code fragment in Example 12.5. Some of the calls may block if another process holds a conflicting lock. However, if no conflicting lock is held, then the code fragment must complete.

Consider the code illustrated in Figure 12.6. Each process updates the window of the other process using a put operation, then accesses its own window. The post calls are nonblocking, and should complete. Once the post calls occur, RMA access to the windows is enabled, so that each process should complete the sequence of calls start-put-complete. Once these are done, the wait calls should complete at both processes. Thus, this communication should not deadlock, irrespective of the amount of data transferred.

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

The following two examples illustrate the fact that the synchronization between complete and wait is not symmetric: the wait call blocks until the complete executes, but not vice versa. Consider the code illustrated in Figure 12.7. 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 12.8. This

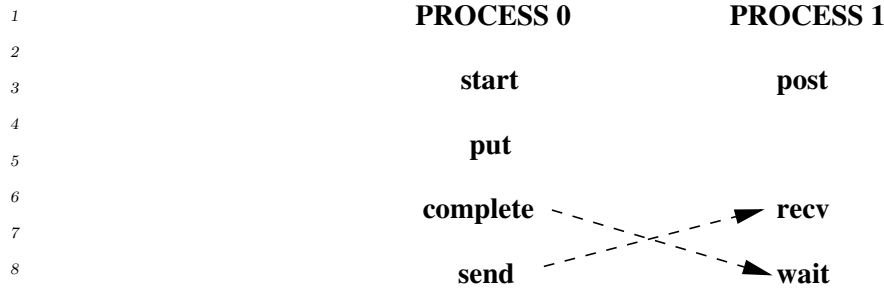


Figure 12.8: No deadlock

code will not deadlock. Once process 1 calls `post`, then the sequence `start`, `put`, `complete` on process 0 can proceed to completion. Process 0 will reach the `send` call, allowing the receive call of process 1 to complete.

Rationale. MPI implementations must guarantee that a process makes *progress* on all enabled communications it participates in, while blocked on an MPI call. This is true for send-receive communication and applies to RMA communication as well. Thus, in the example in Figure 12.8, the `put` and `complete` calls of process 0 should complete while process 1 is blocked on the receive call. This may require the involvement of process 1, e.g., to transfer the data put, while it is blocked on the receive call.

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.*)

12.7.4 Registers and Compiler Optimizations

Advice to users. All the material in this section is an advice to users. (*End of advice to users.*)

A coherence problem exists between variables kept in registers and the memory values of these variables. An RMA call may access a variable in memory (or cache), while the up-to-date value of this variable is in register. A `get` will not return the latest variable value, and a `put` may be overwritten when the register is stored back in memory. Note that these issues are unrelated to the RMA memory model; that is, these issues apply even if the memory model is `MPI_WIN_UNIFIED`.

The problem is illustrated in Example 12.13.

Example 12.13.

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.

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

This problem, which also afflicts in some cases send/receive communication, is discussed more at length in Section 19.1.16.

Programs written in C avoid this problem, because of the semantics of C. Many Fortran compilers will avoid this problem, without disabling compiler optimizations. However, in order to avoid register coherence problems in a completely portable manner, users should restrict their use of RMA windows to variables stored in modules or COMMON blocks. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 19.1.10–19.1.20. Sections 19.1.17 to 19.1.17 discuss several solutions for the problem in this example.

12.8 Examples

Example 12.14. 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 12.15. 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.

```

1  ...
2  while (!converged(A)) {
3      update_boundary(A);
4      MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win);
5      for(i=0; i < fromneighbors; i++)
6          MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
7                  fromdisp[i], 1, fromtype[i], win);
8      update_core(A);
9      MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
10 }

```

The get communication can be concurrent with the core update, since they do not access the same locations, and the local update of the origin buffer by the get call can be concurrent with the local update of the core by the `update_core` call. In order to get similar overlap with put communication we would need to use separate windows for the core and for the boundary. This is required because we do not allow local stores to be concurrent with puts on the same, or on overlapping, windows.

Example 12.16. Same code as in Example 12.14, rewritten using post-start-complete-wait.

```

21 ...
22 while (!converged(A)) {
23     update(A);
24     MPI_Win_post(fromgroup, 0, win);
25     MPI_Win_start(togroup, 0, win);
26     for(i=0; i < toneighbors; i++)
27         MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
28                todisp[i], 1, totype[i], win);
29     MPI_Win_complete(win);
30     MPI_Win_wait(win);
31 }

```

Example 12.17. Same example, with split phases, as in Example 12.15.

```

34 ...
35 while (!converged(A)) {
36     update_boundary(A);
37     MPI_Win_post(togroup, MPI_MODE_NOPUT, win);
38     MPI_Win_start(fromgroup, 0, win);
39     for(i=0; i < fromneighbors; i++)
40         MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
41                fromdisp[i], 1, fromtype[i], win);
42     update_core(A);
43     MPI_Win_complete(win);
44     MPI_Win_wait(win);
45 }

```

Example 12.18. A checkerboard, or double buffer communication pattern, that allows more computation/communication overlap. Array A_0 is updated using values of array A_1 , and vice versa. We assume that communication is symmetric: if process A gets data from process B, then process B gets data from process A. Window win_i consists of array A_i .

```

...
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 win_0 before it completes RMA accesses to the remote windows associated with win_1 . When the $wait(win_1)$ call returns, then all neighbors of the calling process have posted the windows associated with win_0 . Conversely, when the $wait(win_0)$ call returns, then all neighbors of the calling process have posted the windows associated with win_1 . 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 A_0 (resp. A_1) 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 and MPI_COMM_SIZE. Process B... refers to any process other than A.

Example 12.19. The following example implements a naive, nonscalable 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...
<code>MPI_Win_lock_all</code>	<code>MPI_Win_lock_all</code>
window location X	
<code>X=MPI_Comm_size()</code>	
<code>MPI_Win_sync</code>	
<code>MPI_Barrier</code>	<code>MPI_Barrier</code>
<code>MPI_Accumulate(X, MPI_SUM, -1)</code>	<code>MPI_Accumulate(X, MPI_SUM, -1)</code>
stack variable z	stack variable z
do	do
<code>z = MPI_Get_accumulate(X,</code>	<code>z = MPI_Get_accumulate(X,</code>
<code>MPI_NO_OP, 0)</code>	<code>MPI_NO_OP, 0)</code>
<code>MPI_Win_flush(A)</code>	<code>MPI_Win_flush(A)</code>
while (z!=0)	while (z!=0)
<code>MPI_Win_unlock_all</code>	<code>MPI_Win_unlock_all</code>

Example 12.20. 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	
<code>MPI_Win_lock_all</code>	<code>MPI_Win_lock_all</code>
X=1	Y=1
<code>MPI_Win_sync</code>	<code>MPI_Win_sync</code>
<code>MPI_Barrier</code>	<code>MPI_Barrier</code>
<code>MPI_Accumulate(T, MPI_REPLACE, 1)</code>	<code>MPI_Accumulate(T, MPI_REPLACE, 0)</code>
stack variables t,y	stack variable t,x
t=1	t=0
y=MPI_Get_accumulate(Y,	x=MPI_Get_accumulate(X,
<code>MPI_NO_OP, 0)</code>	<code>MPI_NO_OP, 0)</code>
while (y==1 && t==1) do	while (x==1 && t==0) do
y=MPI_Get_accumulate(Y,	x=MPI_Get_accumulate(X,
<code>MPI_NO_OP, 0)</code>	<code>MPI_NO_OP, 0)</code>

<pre> t=MPI_Get_accumulate(T, MPI_NO_OP, 0) MPI_Win_flush_all done // critical region MPI_Accumulate(X, MPI_REPLACE, 0) MPI_Win_unlock_all </pre>	<pre> t=MPI_Get_accumulate(T, MPI_NO_OP, 0) MPI_Win_flush(A) done // critical region MPI_Accumulate(Y, MPI_REPLACE, 0) MPI_Win_unlock_all </pre>
---	--

Example 12.21. 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...
<pre> MPI_Win_lock_all atomic location A A=0 MPI_Win_sync MPI_Barrier stack variable r=1 while(r != 0) do r = MPI_Compare_and_swap(A, 0, 1) MPI_Win_flush(A) done // critical region r = MPI_Compare_and_swap(A, 1, 0) MPI_Win_unlock_all </pre>	<pre> MPI_Win_lock_all MPI_Barrier stack variable r=1 while(r != 0) do r = MPI_Compare_and_swap(A, 0, 1) MPI_Win_flush(A) done // critical region r = MPI_Compare_and_swap(A, 1, 0) MPI_Win_unlock_all </pre>

Example 12.22. 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 first call to `MPI_WIN_SYNC` performed by process A ensures completion of the load/store operations issued by process A. The first call to `MPI_WIN_SYNC` performed by process B ensures that process A's updates to X are visible to process B. Similarly, the second call to `MPI_WIN_SYNC` on each process ensures correct ordering of the point-to-point communication and thus that the load/store operations on process B have completed before any subsequent load/store operations to the variable X in process A.

Process A	Process B
-----------	-----------

```

1  MPI_WIN_LOCK_ALL (                MPI_WIN_LOCK_ALL (
2      MPI_MODE_NOCHECK ,win)        MPI_MODE_NOCHECK ,win)
3
4  DO ...                             DO ...
5      X=...
6
7      MPI_F_SYNC_REG(X)
8      MPI_WIN_SYNC(win)
9      MPI_SEND
10
11
12
13
14
15      MPI_RECV
16      MPI_WIN_SYNC(win)
17      MPI_F_SYNC_REG(X)
18  END DO                             END DO
19
20  MPI_WIN_UNLOCK_ALL(win)           MPI_WIN_UNLOCK_ALL(win)
21

```

Example 12.23. 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 N STEPS chunks of data. Instead of a single buffer, M local buffers are used to allow up to M communication operations to overlap with computation.

```

27  int          i, j;
28  MPI_Win      win;
29  MPI_Request  put_req[M] = { MPI_REQUEST_NULL };
30  MPI_Request  get_req;
31  double       *baseptr;
32  double       data[M][N];
33
34  MPI_Win_allocate(NSTEPS*N*sizeof(double), sizeof(double), MPI_INFO_NULL,
35                  MPI_COMM_WORLD, &baseptr, &win);
36
37  MPI_Win_lock_all(0, win);
38
39  for (i = 0; i < NSTEPS; i++) {
40      if (i < M)
41          j = i;
42      else
43          MPI_Waitany(M, put_req, &j, MPI_STATUS_IGNORE);
44
45      MPI_Rget(data[j], N, MPI_DOUBLE, target, i*N, N, MPI_DOUBLE, win,
46              &get_req);
47      MPI_Wait(&get_req, MPI_STATUS_IGNORE);
48      compute(i, data[j], ...);
49      MPI_Rput(data[j], N, MPI_DOUBLE, target, i*N, N, MPI_DOUBLE, win,
50              &put_req[j]);

```

```

}
MPI_Waitall(M, put_req, MPI_STATUSES_IGNORE);
MPI_Win_unlock_all(win);

```

Example 12.24. 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;
    }
}

```

```

1   my_elems = realloc(my_elems, my_elems_size*sizeof(void*));
2   }
3   my_elems[my_elems_count] = elem_ptr;
4   my_elems_count++;
5
6   MPI_Get_address(elem_ptr, &disp);
7   return disp;
8   }
9
10  int main(int argc, char *argv[]) {
11      int          procid, nproc, i;
12      MPI_Win      llist_win;
13      llist_ptr_t  head_ptr, tail_ptr;
14
15      MPI_Init(&argc, &argv);
16
17      MPI_Comm_rank(MPI_COMM_WORLD, &procid);
18      MPI_Comm_size(MPI_COMM_WORLD, &nproc);
19
20      MPI_Win_create_dynamic(MPI_INFO_NULL, MPI_COMM_WORLD, &llist_win);
21
22      /* Process 0 creates the head node */
23      if (procid == 0)
24          head_ptr.disp = alloc_elem(-1, llist_win);
25
26      /* Broadcast the head pointer to everyone */
27      head_ptr.rank = 0;
28      MPI_Bcast(&head_ptr.disp, 1, MPI_AINT, 0, MPI_COMM_WORLD);
29      tail_ptr = head_ptr;
30
31      /* Lock the window for shared access to all targets */
32      MPI_Win_lock_all(0, llist_win);
33
34      /* All processes concurrently append NUM_ELEMS elements to the list */
35      for (i = 0; i < NUM_ELEMS; i++) {
36          llist_ptr_t new_elem_ptr;
37          int success;
38
39          /* Create a new list element and attach it to the window */
40          new_elem_ptr.rank = procid;
41          new_elem_ptr.disp = alloc_elem(procid, llist_win);
42
43          /* Append the new node to the list. This might take multiple
44             attempts if others have already appended and our tail pointer
45             is stale. */
46          do {
47              llist_ptr_t next_tail_ptr = nil;
48
49              MPI_Compare_and_swap((void*) &new_elem_ptr.rank, (void*) &nil.rank,
50                                  (void*)&next_tail_ptr.rank, MPI_INT, tail_ptr.rank,
51                                  MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_RANK),
52                                  llist_win);
53
54              MPI_Win_flush(tail_ptr.rank, llist_win);
55              success = (next_tail_ptr.rank == nil.rank);
56
57              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);
...

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

DRAFT

Chapter 13

External Interfaces

13.1 Introduction

This chapter contains calls used to create **generalized requests**, which allow users to create new nonblocking operations with an interface similar to what is present in MPI. These calls can be used to layer new functionality on top of MPI. Section 13.3 deals with setting the information found in `status`. This functionality is needed for generalized requests.

13.2 Generalized Requests

The goal of generalized requests is to allow users to define new nonblocking operations. Such an outstanding nonblocking operation is represented by a (generalized) request. A fundamental property of nonblocking operations is that *progress* toward the completion of this operation occurs asynchronously, i.e., concurrently with normal program execution. Typically, this requires execution of code concurrently with the execution of the user code, e.g., in a separate thread or in a signal handler. Operating systems provide a variety of mechanisms in support of concurrent execution. MPI does not attempt to standardize or to replace these mechanisms: it is assumed programmers who wish to define new asynchronous operations will use the mechanisms provided by the underlying operating system. Thus, the calls in this section only provide a means for defining the effect of MPI calls such as `MPI_WAIT` or `MPI_CANCEL` when they apply to generalized requests, and for signaling to MPI the completion of a generalized operation.

Rationale. It is tempting to also define an MPI standard mechanism for achieving concurrent execution of user-defined nonblocking operations. However, it is difficult to define such a mechanism without consideration of the specific mechanisms used in the operating system. The Forum feels that concurrency mechanisms are a proper part of the underlying operating system and should not be standardized by MPI; the MPI standard should only deal with the interaction of such mechanisms with MPI. (*End of rationale.*)

For a regular request, the operation associated with the request is performed by the MPI implementation, and the operation completes without intervention by the application. For a generalized request, the operation associated with the request is performed by the application; therefore, the application must notify MPI through a call to `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

```

1 MPI_GREQUEST_START(query_fn, free_fn, cancel_fn, extra_state, request)
2     IN      query_fn      callback function invoked when request status is
3                          queried (function)
4
5     IN      free_fn       callback function invoked when request is freed
6                          (function)
7
8     IN      cancel_fn     callback function invoked when request is cancelled
9                          (function)
10
11    IN      extra_state    extra state
12
13    OUT     request        generalized request (handle)

```

C binding

```

14 int MPI_Grequest_start(MPI_Grequest_query_function *query_fn,
15                       MPI_Grequest_free_function *free_fn,
16                       MPI_Grequest_cancel_function *cancel_fn, void *extra_state,
17                       MPI_Request *request)

```

Fortran 2008 binding

```

18 MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request, ierror)
19
20 PROCEDURE(MPI_Grequest_query_function) :: query_fn
21 PROCEDURE(MPI_Grequest_free_function) :: free_fn
22 PROCEDURE(MPI_Grequest_cancel_function) :: cancel_fn
23 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
24 TYPE(MPI_Request), INTENT(OUT) :: request
25 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

26 MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST, IERROR)
27
28 EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN
29 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
30 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

```

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

```

in Fortran with the `mpi_f08` module

```

42 ABSTRACT INTERFACE
43
44 SUBROUTINE MPI_Grequest_query_function(extra_state, status, ierror)
45     INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
46     TYPE(MPI_Status) :: status

```



```
INTEGER :: ierror
```

in Fortran with the `mpi` module and `mpif.h`

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

The `query_fn` function computes the status that should be returned for the generalized request. The status also includes information about successful/unsuccesful 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(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
  INTEGER IERROR
```

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(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
  LOGICAL COMPLETE
  INTEGER IERROR
```

The `cancel_fn` function is invoked to start the cancelation of a generalized request. It is called by `MPI_CANCEL(request)`. MPI passes `complete = true` to the callback function if `MPI_GREQUEST_COMPLETE` was already called on the request, and `complete = false` otherwise.

All callback functions return an error code. The code is passed back and dealt with as appropriate for the error code by the MPI function that invoked the callback function. For example, if error codes are returned then the error code returned by the callback function will be returned by the MPI function that invoked the callback function. In the case of an `MPI_{WAIT|TEST}{ANY}` call that invokes both `query_fn` and `free_fn`, the MPI call will 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)
```

Fortran 2008 binding

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

Fortran binding

```
MPI_GREQUEST_COMPLETE(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
```

The call informs MPI that the operations represented by the generalized request `request` are complete (see definitions in Section 2.4). A call to `MPI_WAIT(request, status)` will return and a call to `MPI_TEST(request, flag, status)` will return `flag = true` only after a call to `MPI_GREQUEST_COMPLETE` has declared that these operations are complete.

MPI imposes no restrictions on the code executed by the callback functions. However, new nonblocking operations should be defined so that the general semantic rules about MPI calls such as `MPI_TEST`, `MPI_REQUEST_FREE`, or `MPI_CANCEL` still hold. For example, these calls are supposed to be local and nonblocking. Therefore, the callback functions `query_fn`, `free_fn`, or `cancel_fn` should invoke blocking MPI communication calls only if the context is such that these calls are guaranteed to return in finite time. Once `MPI_CANCEL` is invoked, the cancelled operation should complete in finite time, irrespective of the state of other processes (the operation has acquired “local” semantics). It should either succeed, or fail without side-effects. The user should guarantee these same properties for newly defined operations.

Advice to implementors. A call to `MPI_GREQUEST_COMPLETE` may unblock a blocked user process/thread. The MPI library should ensure that the blocked user computation will resume. (*End of advice to implementors.*)

13.2.1 Examples

Example 13.1. This example shows the code for a user-defined reduce operation on an `int` using a binary tree: each nonroot 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;
```

```

1  int valin;
2  int *valout;
3  MPI_Request request;
4  } ARGS;
5
6
7  int myreduce(MPI_Comm comm, int tag, int root,
8              int valin, int *valout, MPI_Request *request)
9  {
10     ARGS *args;
11     pthread_t thread;
12
13     /* start request */
14     MPI_Grequest_start(query_fn, free_fn, cancel_fn, NULL, request);
15
16     args = (ARGS*)malloc(sizeof(ARGS));
17     args->comm = comm;
18     args->tag = tag;
19     args->root = root;
20     args->valin = valin;
21     args->valout = valout;
22     args->request = *request;
23
24     /* spawn thread to handle request */
25     /* The availability of the pthread_create call is system dependent */
26     pthread_create(&thread, NULL, reduce_thread, args);
27
28     return MPI_SUCCESS;
29 }
30
31 /* thread code */
32 void* reduce_thread(void *ptr)
33 {
34     int lchild, rchild, parent, lval, rval, val;
35     MPI_Request req[2];
36     ARGS *args;
37
38     args = (ARGS*)ptr;
39
40     /* compute left and right child and parent in tree; set
41        to MPI_PROC_NULL if does not exist */
42     /* code not shown */
43     ...
44
45     MPI_Irecv(&lval, 1, MPI_INT, lchild, args->tag, args->comm, &req[0]);
46     MPI_Irecv(&rval, 1, MPI_INT, rchild, args->tag, args->comm, &req[1]);
47     MPI_Waitall(2, req, MPI_STATUSES_IGNORE);
48     val = lval + args->valin + rval;
49     MPI_Send(&val, 1, MPI_INT, parent, args->tag, args->comm);
50     if (parent == MPI_PROC_NULL) *(args->valout) = val;
51     MPI_Grequest_complete((args->request));
52     free(ptr);
53     return(NULL);

```

```

}
1
2
int query_fn(void *extra_state, MPI_Status *status)
3
{
4
    /* always send just one int */
5
    MPI_Status_set_elements(status, MPI_INT, 1);
6
    /* can never cancel so always true */
7
    MPI_Status_set_cancelled(status, 0);
8
    /* choose not to return a value for this */
9
    status->MPI_SOURCE = MPI_UNDEFINED;
10
    /* tag has no meaning for this generalized request */
11
    status->MPI_TAG = MPI_UNDEFINED;
12
    /* this generalized request never fails */
13
    return MPI_SUCCESS;
14
}
15
16
int free_fn(void *extra_state)
17
{
18
    /* this generalized request does not need to do any freeing */
19
    /* as a result it never fails here */
20
    return MPI_SUCCESS;
21
}
22
23
int cancel_fn(void *extra_state, int complete)
24
{
25
    /* This generalized request does not support cancelling.
26
       Abort if not already done.
27
       If done then treat as if cancel failed.*/
28
    if (!complete) {
29
        fprintf(stderr,
30
            "Cannot cancel generalized request - aborting program\n");
31
        MPI_Abort(MPI_COMM_WORLD, 99);
32
    }
33
    return MPI_SUCCESS;
34
}
35

```

13.3 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 (integer)

C binding

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

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

Fortran 2008 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
```

```
MPI_Status_set_elements(status, datatype, count, ierror) !(_c)
```

```
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
```

Fortran 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)
```

Fortran 2008 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

```

Fortran 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 calls to `MPI_GET_ELEMENTS`, `MPI_GET_ELEMENTS_X`, or `MPI_GET_ELEMENTS_C` will return `count`. Calls to `MPI_GET_COUNT` or `MPI_GET_COUNT_C` 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.*)

Subsequent calls to `MPI_GET_COUNT`, `MPI_GET_COUNT_C`, `MPI_GET_ELEMENTS`, `MPI_GET_ELEMENTS_X`, or `MPI_GET_ELEMENTS_C` 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`, `MPI_STATUS_SET_ELEMENTS_X`, or `MPI_STATUS_SET_ELEMENTS_C`.

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_COUNT_C`, `MPI_GET_ELEMENTS`, `MPI_GET_ELEMENTS_X`, and `MPI_GET_ELEMENTS_C` 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)

```

Fortran 2008 binding

```

MPI_Status_set_cancelled(status, flag, ierror)
  TYPE(MPI_Status), INTENT(INOUT) :: status
  LOGICAL, INTENT(IN) :: flag
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR)
  INTEGER STATUS(MPI_STATUS_SIZE), IERROR
  LOGICAL FLAG

```

1 If flag is set to true then a subsequent call to MPI_TEST_CANCELLED will also return
2 flag = true, otherwise it will return false.

3
4 *Advice to users.* Users are advised not to reuse the status fields for values other
5 than those for which they were intended. Doing so may lead to unexpected results
6 when using the status object. For example, calling MPI_GET_ELEMENTS may cause
7 an error if the value is out of range or it may be impossible to detect such an error.
8 The extra_state argument provided with a generalized request can be used to return
9 information that does not logically belong in status. Furthermore, modifying the
10 values in a status set internally by MPI, e.g., MPI_RECV, may lead to unpredictable
11 results and is strongly discouraged. (*End of advice to users.*)
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

DRAFT

Chapter 14

I/O

14.1 Introduction

POSIX provides a model of a widely portable file system, but the portability and optimization needed for parallel I/O cannot be achieved with the POSIX interface.

The significant optimizations required for efficiency (e.g., grouping [54], collective buffering [8, 16, 55, 59, 66], and disk-directed I/O [48]) can only be implemented if the parallel I/O system provides a high-level interface supporting partitioning of file data among processes and a collective interface supporting complete transfers of global data structures between process memories and files. In addition, further efficiencies can be gained via support for asynchronous I/O, strided accesses, and control over physical file layout on storage devices (disks). The I/O environment described in this chapter provides these facilities.

Instead of defining I/O access modes to express the common patterns for accessing a shared file (broadcast, reduction, scatter, gather), we chose another approach in which data partitioning is expressed using derived datatypes. Compared to a limited set of predefined access patterns, this approach has the advantage of added flexibility and expressiveness.

14.1.1 Definitions

file: An MPI file is an ordered collection of typed data items. MPI supports random or sequential access to any integral set of these items. A file is opened collectively by a group of processes. All collective I/O calls on a file are collective over this group.

displacement: A file *displacement* is an absolute byte position relative to the beginning of a file. The displacement defines the location where a *view* begins. Note that a “file displacement” is distinct from a “typemap displacement.”

etype: An *etype* (*elementary* datatype) is the unit of data access and positioning. It can be any MPI predefined or derived datatype. Derived etypes can be constructed using any of the MPI datatype constructor routines, provided all resulting typemap displacements are nonnegative and monotonically nondecreasing. Data access is performed in etype units, reading or writing whole data items of type etype. Offsets are expressed as a count of etypes; file pointers point to the beginning of etypes. Depending on context, the term “etype” is used to describe one of three aspects of an elementary datatype: a particular MPI type, a data item of that type, or the extent of that type.

filetype: A *filetype* is the basis for partitioning a file among processes and defines a template for accessing the file. A filetype is either a single etype or a derived MPI datatype constructed from multiple instances of the same etype. In addition, the extent of any hole in the filetype must be a multiple of the etype’s extent. The displacements in the

1 typemap of the filetype are not required to be distinct, but they must be nonnegative
 2 and monotonically nondecreasing.

3
 4 **view:** A *view* defines the current set of data visible and accessible from an open file as
 5 an ordered set of etypes. Each process has its own view of the file, defined by three
 6 quantities: a displacement, an etype, and a filetype. The pattern described by a
 7 filetype is repeated, beginning at the displacement, to define the view. The pattern
 8 of repetition is defined to be the same pattern that MPI_TYPE_CONTIGUOUS would
 9 produce if it were passed the filetype and an arbitrarily large count. Figure 14.1 shows
 10 how the tiling works; note that the filetype in this example must have explicit lower
 11 and upper bounds set in order for the initial and final holes to be repeated in the
 12 view. Views can be changed by the user during program execution. The default view
 13 is a linear byte stream (displacement is zero, etype and filetype equal to MPI_BYTE).

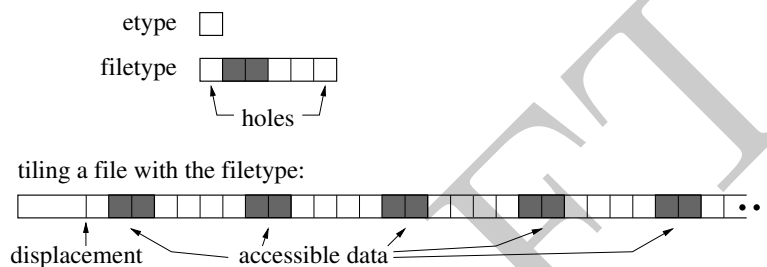


Figure 14.1: Etypes and filetypes

24 A group of processes can use complementary views to achieve a global data distribution
 25 such as a scatter/gather pattern (see Figure 14.2).



Figure 14.2: Partitioning a file among parallel processes

39 **offset:** An *offset* is a position in the file relative to the current view, expressed as a count of
 40 etypes. Holes in the view’s filetype are skipped when calculating this position. Offset 0
 41 is the location of the first etype visible in the view (after skipping the displacement and
 42 any initial holes in the view). For example, an offset of 2 for process 1 in Figure 14.2 is
 43 the position of the eighth etype in the file after the displacement. An “explicit offset”
 44 is an offset that is used as an argument in explicit data access routines.

46 **file size and end of file:** The *size* of an MPI file is measured in bytes from the beginning
 47 of the file. A newly created file has a size of zero bytes. Using the size as an absolute
 48 displacement gives the position of the byte immediately following the last byte in the

file. For any given view, the *end of file* is the offset of the first etype accessible in the current view starting after the last byte in the file.

file pointer: A *file pointer* is an implicit offset maintained by MPI. “Individual file pointers” are file pointers that are local to each process that opened the file. A “shared file pointer” is a file pointer that is shared by the group of processes that opened the file.

file handle: A *file handle* is an opaque object created by `MPI_FILE_OPEN` and freed by `MPI_FILE_CLOSE`. All operations on an open file reference the file through the file handle.

14.2 File Manipulation

14.2.1 Opening a File

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

IN	comm	communicator (handle)
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)
```

Fortran 2008 binding

```
MPI_File_open(comm, filename, amode, info, fh, ierror)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  CHARACTER(LEN=*), INTENT(IN) :: filename
  INTEGER, INTENT(IN) :: amode
  TYPE(MPI_Info), INTENT(IN) :: info
  TYPE(MPI_File), INTENT(OUT) :: fh
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR)
  INTEGER COMM, AMODE, INFO, FH, IERROR
  CHARACTER*(*) FILENAME
```

`MPI_FILE_OPEN` opens the file identified by the file name `filename` on all processes in the `comm` communicator group. `MPI_FILE_OPEN` is a collective routine: all processes must provide the same value for `amode`, and all processes must provide `filenames` that reference the same file. (Values for `info` may vary.) `comm` must be an intra-communicator; it is erroneous to pass an inter-communicator to `MPI_FILE_OPEN`. Errors in `MPI_FILE_OPEN` are raised using the default file error handler (see Section 14.7). When using the World

1 Model (Section 11.1), a process can open a file independently of other processes by using
 2 the MPI_COMM_SELF communicator. Applications using the Sessions Model (Section 11.3)
 3 can achieve the same result using communicators created from the "mpi://SELF" process
 4 set. The file handle returned, fh, can be subsequently used to access the file until the file is
 5 closed using MPI_FILE_CLOSE. Before calling MPI_FINALIZE, the user is required to close
 6 (via MPI_FILE_CLOSE) all files that were opened with MPI_FILE_OPEN. Note that the
 7 communicator comm is unaffected by MPI_FILE_OPEN and continues to be usable in all
 8 MPI routines (e.g., MPI_SEND). Furthermore, the use of comm will not interfere with I/O
 9 behavior.

10 The format for specifying the file name in the filename argument is implementation
 11 dependent and must be documented by the implementation.

12 *Advice to implementors.* An implementation may require that filename include a
 13 string or strings specifying additional information about the file. Examples include
 14 the type of filesystem (e.g., a prefix of ufs:), a remote hostname (e.g., a prefix of
 15 machine.univ.edu:), or a file password (e.g., a suffix of /PASSWORD=SECRET). (*End of*
 16 *advice to implementors.*)

17
 18 *Advice to users.* On some implementations of MPI, the file namespace may not be
 19 identical from all processes of all applications. For example, "/tmp/foo" may denote
 20 different files on different processes, or a single file may have many names, dependent
 21 on process location. The user is responsible for ensuring that a single file is referenced
 22 by the filename argument, as it may be impossible for an implementation to detect
 23 this type of namespace error. (*End of advice to users.*)

24
 25 Initially, all processes view the file as a linear byte stream, and each process views data
 26 in its own native representation (no data representation conversion is performed). (POSIX
 27 files are linear byte streams in the native representation.) The file view can be changed via
 28 the MPI_FILE_SET_VIEW routine.

29 The following access modes are supported (specified in amode, a bit vector OR of the
 30 following integer constants):

31 MPI_MODE_RDONLY	read only
32 MPI_MODE_RDWR	reading and writing
33 MPI_MODE_WRONLY	write only
34 MPI_MODE_CREATE	create the file if it does not exist
35 MPI_MODE_EXCL	error if creating file that already exists
36 MPI_MODE_DELETE_ON_CLOSE	delete file on close
37 MPI_MODE_UNIQUE_OPEN	file will not be concurrently opened elsewhere
38 MPI_MODE_SEQUENTIAL	file will only be accessed sequentially
39 MPI_MODE_APPEND	set initial position of all file pointers to end of file

40
 41 *Advice to users.* C users can use bit vector OR (|) to combine these constants; Fortran
 42 90 users can use the bit vector IOR intrinsic. Fortran 77 users can use (nonportably)
 43 bit vector IOR on systems that support it. Alternatively, Fortran users can portably
 44 use integer addition to OR the constants (each constant should appear at most once
 45 in the addition.). (*End of advice to users.*)

46 *Advice to implementors.* The values of these constants must be defined such that
 47 the bitwise OR and the sum of any distinct set of these constants is equivalent. (*End*
 48 *of advice to implementors.*)

The modes `MPI_MODE_RDONLY`, `MPI_MODE_RDWR`, `MPI_MODE_WRONLY`, `MPI_MODE_CREATE`, and `MPI_MODE_EXCL` have identical semantics to their POSIX counterparts [44]. Exactly one of `MPI_MODE_RDONLY`, `MPI_MODE_RDWR`, or `MPI_MODE_WRONLY`, must be specified. It is erroneous to specify `MPI_MODE_CREATE` or `MPI_MODE_EXCL` in conjunction with `MPI_MODE_RDONLY`; it is erroneous to specify `MPI_MODE_SEQUENTIAL` together with `MPI_MODE_RDWR`.

The `MPI_MODE_DELETE_ON_CLOSE` mode causes the file to be deleted (equivalent to performing an `MPI_FILE_DELETE`) when the file is closed.

The `MPI_MODE_UNIQUE_OPEN` mode allows an implementation to optimize access by eliminating the overhead of file locking. It is erroneous to open a file in this mode unless the file will not be concurrently opened elsewhere.

Advice to users. For `MPI_MODE_UNIQUE_OPEN`, *not opened elsewhere* includes both inside and outside the MPI environment. In particular, one needs to be aware of potential external events that may open files (e.g., automated backup facilities). When `MPI_MODE_UNIQUE_OPEN` is specified, the user is responsible for ensuring that no such external events take place. (*End of advice to users.*)

The `MPI_MODE_SEQUENTIAL` mode allows an implementation to optimize access to some sequential devices (tapes and network streams). It is erroneous to attempt nonsequential access to a file that has been opened in this mode.

Specifying `MPI_MODE_APPEND` only guarantees that all shared and individual file pointers are positioned at the initial end of file when `MPI_FILE_OPEN` returns. Subsequent positioning of file pointers is application dependent. In particular, the implementation does not ensure that all writes are appended.

Errors related to the access mode are raised in the class `MPI_ERR_AMODE`.

The `info` argument is used to provide information regarding file access patterns and file system specifics (see Section 14.2.8). The constant `MPI_INFO_NULL` can be used when no `info` needs to be specified.

Advice to users. Some file attributes are inherently implementation dependent (e.g., file permissions). These attributes must be set using either the `info` argument or facilities outside the scope of MPI. (*End of advice to users.*)

Files are opened by default using nonatomic mode file consistency semantics (see Section 14.6.1). The more stringent atomic mode consistency semantics, required for atomicity of conflicting accesses, can be set using `MPI_FILE_SET_ATOMICITY`.

14.2.2 Closing a File

`MPI_FILE_CLOSE(fh)`

INOUT fh file handle (handle)

C binding

`int MPI_File_close(MPI_File *fh)`

Fortran 2008 binding

`MPI_File_close(fh, ierror)`

```

1     TYPE(MPI_File), INTENT(INOUT) :: fh
2     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

4 MPI_FILE_CLOSE(FH, IERROR)
5     INTEGER FH, IERROR
6

```

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

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

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

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

14.2.3 Deleting a File

```

25 MPI_FILE_DELETE(filename, info)
26
27     IN     filename           name of file to delete (string)
28     IN     info              info object (handle)
29

```

C binding

```

31 int MPI_File_delete(const char *filename, MPI_Info info)
32

```

Fortran 2008 binding

```

34 MPI_File_delete(filename, info, ierror)
35     CHARACTER(LEN=*), INTENT(IN) :: filename
36     TYPE(MPI_Info), INTENT(IN) :: info
37     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38

```

Fortran binding

```

39 MPI_FILE_DELETE(FILENAME, INFO, IERROR)
40     CHARACTER*(*) FILENAME
41     INTEGER INFO, IERROR
42

```

43 MPI_FILE_DELETE deletes the file identified by the file name filename. If the file does
44 not exist, MPI_FILE_DELETE raises an error in the class MPI_ERR_NO_SUCH_FILE.

45 The info argument can be used to provide information regarding file system specifics
46 (see Section 14.2.8). The constant MPI_INFO_NULL refers to the null info, and can be used
47 when no info needs to be specified.

48

If a process currently has the file open, the behavior of any access to the file (as well as the behavior of any outstanding accesses) is implementation dependent. In addition, whether an open file is deleted or not is also implementation dependent. If the file is not deleted, an error in the class `MPI_ERR_FILE_IN_USE` or `MPI_ERR_ACCESS` will be raised. Errors are raised using the default file error handler (see Section 14.7).

14.2.4 Resizing a File

`MPI_FILE_SET_SIZE(fh, size)`

INOUT	fh	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)
```

Fortran 2008 binding

```
MPI_File_set_size(fh, size, ierror)
  TYPE(MPI_File), INTENT(IN) :: fh
  INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_FILE_SET_SIZE(FH, SIZE, IERROR)
  INTEGER FH, IERROR
  INTEGER(KIND=MPI_OFFSET_KIND) SIZE
```

`MPI_FILE_SET_SIZE` resizes the file associated with the file handle `fh`. `size` is measured in bytes from the beginning of the file. `MPI_FILE_SET_SIZE` is collective; all processes in the group must pass identical values for `size`.

If `size` is smaller than the current file size, the file is truncated at the position defined by `size`. The implementation is free to deallocate file blocks located beyond this position.

If `size` is larger than the current file size, the file size becomes `size`. Regions of the file that have been previously written are unaffected. The values of data in the new regions in the file (those locations with displacements between old file size and `size`) are undefined. It is implementation dependent whether the `MPI_FILE_SET_SIZE` routine allocates file space—use `MPI_FILE_PREALLOCATE` to force file space to be reserved.

`MPI_FILE_SET_SIZE` does not affect the individual file pointers or the shared file 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 14.6.1).

14.2.5 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)
```

Fortran 2008 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
```

Fortran 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 time-consuming.
(*End of advice to users.*)

14.2.6 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)
```


Fortran 2008 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

```

Fortran 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 14.6.1).

14.2.7 Querying File Parameters

MPI_FILE_GET_GROUP(*fh*, *group*)

IN	<i>fh</i>	file handle (handle)
OUT	<i>group</i>	group that opened the file (handle)

C binding

```
int MPI_File_get_group(MPI_File fh, MPI_Group *group)
```

Fortran 2008 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

```

Fortran 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	<i>fh</i>	file handle (handle)
OUT	<i>amode</i>	file access mode used to open the file (integer)

C binding

```
int MPI_File_get_amode(MPI_File fh, int *amode)
```

Fortran 2008 binding

```
MPI_File_get_amode(fh, amode, ierror)
```

```

1     TYPE(MPI_File), INTENT(IN) :: fh
2     INTEGER, INTENT(OUT) :: amode
3     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

5     MPI_FILE_GET_AMODE(FH, AMODE, IERROR)
6     INTEGER FH, AMODE, IERROR

```

8 MPI_FILE_GET_AMODE returns, in `amode`, the access mode of the file associated with `fh`.

10 **Example 14.1.** In Fortran 77, decoding an `amode` bit vector will require a routine such as the following:

```

13     SUBROUTINE BIT_QUERY(TEST_BIT, MAX_BIT, AMODE, BIT_FOUND)
14     !
15     ! TEST IF THE INPUT TEST_BIT IS SET IN THE INPUT AMODE
16     ! IF SET, RETURN 1 IN BIT_FOUND, 0 OTHERWISE
17     !
18     INTEGER TEST_BIT, AMODE, BIT_FOUND, CP_AMODE, HIFOUND
19     INTEGER L, LBIT, MATCHER, MAX_BIT
20     BIT_FOUND = 0
21     CP_AMODE = AMODE
22     100 CONTINUE
23     LBIT = 0
24     HIFOUND = 0
25     DO L = MAX_BIT, 0, -1
26         MATCHER = 2**L
27         IF (CP_AMODE .GE. MATCHER .AND. HIFOUND .EQ. 0) THEN
28             HIFOUND = 1
29             LBIT = MATCHER
30             CP_AMODE = CP_AMODE - MATCHER
31         END IF
32     END DO
33     IF (HIFOUND .EQ. 1 .AND. LBIT .EQ. TEST_BIT) BIT_FOUND = 1
34     IF (BIT_FOUND .EQ. 0 .AND. HIFOUND .EQ. 1 .AND. &
35         CP_AMODE .GT. 0) GO TO 100
36 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`.

```

37     CALL BIT_QUERY(MPI_MODE_RDONLY, 30, AMODE, BIT_FOUND)
38     IF (BIT_FOUND .EQ. 1) THEN
39         PRINT *, ' FOUND READ-ONLY BIT IN AMODE=', AMODE
40     ELSE
41         PRINT *, ' READ-ONLY BIT NOT FOUND IN AMODE=', AMODE
42     END IF

```

14.2.8 File Info

Hints specified via `info` (see Chapter 10) allow a user to provide information such as file access patterns and file system specifics to direct optimization. Providing hints may enable an implementation to deliver increased I/O performance or minimize the use of system

resources. An implementation is free to ignore all hints; however, applications must comply with any info hints they provide that are used by the MPI implementation (i.e., are returned by a call to `MPI_FILE_GET_INFO`) and that place a restriction on the behavior of the application. Hints are specified on a per file basis, in `MPI_FILE_OPEN`, `MPI_FILE_DELETE`, `MPI_FILE_SET_VIEW`, and `MPI_FILE_SET_INFO`, via the opaque info object. When an info object that specifies a subset of valid hints is passed to `MPI_FILE_SET_VIEW` or `MPI_FILE_SET_INFO`, there will be no effect on previously set or defaulted hints that the info does not specify.

Advice to implementors. It may happen that a program is coded with hints for one system, and later executes on another system that does not support these hints. In general, unsupported hints should simply be ignored.

However, for each hint used by a specific implementation, a default value must be provided when the user does not specify a value for this hint. (*End of advice to implementors.*)

`MPI_FILE_SET_INFO(fh, info)`

INOUT	fh	file handle (handle)
IN	info	info object (handle)

C binding

```
int MPI_File_set_info(MPI_File fh, MPI_Info info)
```

Fortran 2008 binding

```
MPI_File_set_info(fh, info, ierror)
  TYPE(MPI_File), INTENT(IN) :: fh
  TYPE(MPI_Info), INTENT(IN) :: info
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_FILE_SET_INFO(FH, INFO, IERROR)
  INTEGER FH, INFO, IERROR
```

`MPI_FILE_SET_INFO` updates the hints of the file associated with `fh` using the hints provided in `info`. This operation has no effect on previously set or defaulted hints that are not 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.*)

```

1 MPI_FILE_GET_INFO(fh, info_used)
2     IN      fh          file handle (handle)
3
4     OUT    info_used    new info object (handle)
5

```

C binding

```

7 int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)
8

```

Fortran 2008 binding

```

9 MPI_File_get_info(fh, info_used, ierror)
10     TYPE(MPI_File), INTENT(IN) :: fh
11     TYPE(MPI_Info), INTENT(OUT) :: info_used
12     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13

```

Fortran binding

```

14 MPI_FILE_GET_INFO(FH, INFO_USED, IERROR)
15     INTEGER FH, INFO_USED, IERROR
16

```

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 10.)

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 multidimensional array that is often accessed by subarrays. The value for this hint is a comma separated list of array dimensions, starting from the most significant one (for an array stored in row-major order, as in C, the most significant dimension is the first one; for an array stored in column-major order, as in Fortran, the most significant dimension is the last one, and array dimensions should be reversed).

"chunked_item" (comma separated list of integers) [SAME]: This hint specifies the size of each array entry, in bytes.

"chunked_size" (comma separated list of integers) [SAME]: This hint specifies the 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.

1 **"striping_unit"** (integer) [SAME]: This hint specifies the suggested striping unit to be
 2 used for this file. The striping unit is the amount of consecutive data assigned to one
 3 I/O device before progressing to the next device, when striping across a number of
 4 devices. It is expressed in bytes. This hint is relevant only when the file is created.
 5

6 14.3 File Views

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

11	INOUT	fh	file handle (handle)
12			
13	IN	disp	displacement (integer)
14	IN	etype	elementary datatype (handle)
15			
16	IN	filetype	filetype (handle)
17	IN	datarep	data representation (string)
18	IN	info	info object (handle)
19			

20 C binding

21 int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
 22 MPI_Datatype filetype, const char *datarep, MPI_Info info)
 23

24 Fortran 2008 binding

25 MPI_File_set_view(fh, disp, etype, filetype, datarep, info, ierror)
 26 TYPE(MPI_File), INTENT(IN) :: fh
 27 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: disp
 28 TYPE(MPI_Datatype), INTENT(IN) :: etype, filetype
 29 CHARACTER(LEN=*), INTENT(IN) :: datarep
 30 TYPE(MPI_Info), INTENT(IN) :: info
 31 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 32

32 Fortran binding

33 MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)
 34 INTEGER FH, ETYPE, FILETYPE, INFO, IERROR
 35 INTEGER(KIND=MPI_OFFSET_KIND) DISP
 36 CHARACTER*(*) DATAREP
 37

38 The MPI_FILE_SET_VIEW routine changes the process's view of the data in the file.
 39 The start of the view is set to **disp**; the type of data is set to **etype**; the distribution of data
 40 to processes is set to **filetype**; and the representation of data in the file is set to **datarep**. In
 41 addition, MPI_FILE_SET_VIEW resets the individual file pointers and the shared file pointer
 42 to zero. MPI_FILE_SET_VIEW is collective; the values for **datarep** and the extents of **etype**
 43 in the file data representation must be identical on all processes in the group; values for **disp**,
 44 **filetype**, and **info** may vary. The datatypes passed in **etype** and **filetype** must be committed.

45 The **etype** always specifies the data layout in the file. If **etype** is a portable datatype (see
 46 Section 2.4), the extent of **etype** is computed by scaling any displacements in the datatype
 47 to match the file data representation. If **etype** is not a portable datatype, no scaling is done
 48

when computing the extent of `etype`. The user must be careful when using nonportable `etypes` in heterogeneous environments; see Section 14.5.1 for further details.

If `MPI_MODE_SEQUENTIAL` mode was specified when the file was opened, the special displacement `MPI_DISPLACEMENT_CURRENT` must be passed in `disp`. This sets the displacement to the current position of the shared file pointer. `MPI_DISPLACEMENT_CURRENT` is invalid unless the `amode` for the file has `MPI_MODE_SEQUENTIAL` set.

Rationale. For some sequential files, such as those corresponding to magnetic tapes or streaming network connections, the *displacement* may not be meaningful. `MPI_DISPLACEMENT_CURRENT` allows the view to be changed for these types of files. (*End of rationale.*)

Advice to implementors. It is expected that a call to `MPI_FILE_SET_VIEW` will immediately follow `MPI_FILE_OPEN` in numerous instances. A high-quality implementation will ensure that this behavior is efficient. (*End of advice to implementors.*)

The `disp` displacement argument specifies the position (absolute offset in bytes from the beginning of the file) where the view begins.

Advice to users. `disp` can be used to skip headers or when the file includes a sequence of data segments that are to be accessed in different patterns (see Figure 14.3). Separate views, each using a different displacement and filetype, can be used to access each segment.

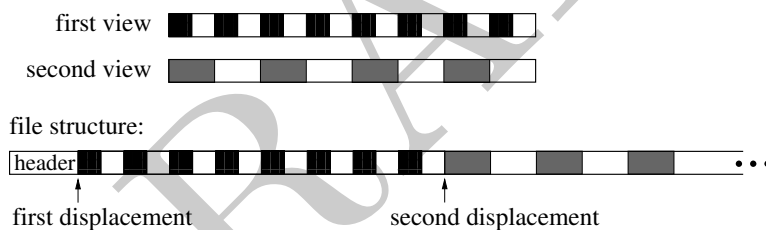


Figure 14.3: Displacements

(*End of advice to users.*)

An *etype* (*elementary* datatype) is the unit of data access and positioning. It can be any MPI predefined or derived datatype. Derived `etypes` can be constructed by using any of the MPI datatype constructor routines, provided all resulting typemap displacements are nonnegative and monotonically nondecreasing. Data access is performed in `etype` units, reading or writing whole data items of type `etype`. Offsets are expressed as a count of `etypes`; file pointers point to the beginning of `etypes`.

Advice to users. In order to ensure interoperability in a heterogeneous environment, additional restrictions must be observed when constructing the `etype` (see Section 14.5). (*End of advice to users.*)

A `filetype` is either a single `etype` or a derived MPI datatype constructed from multiple instances of the same `etype`. In addition, the extent of any hole in the `filetype` must be a multiple of the `etype`'s extent. These displacements are not required to be distinct, but they cannot be negative, and they must be monotonically nondecreasing.

1 If the file is opened for writing, neither the `etype` nor the `filetype` is permitted to
 2 contain overlapping regions. This restriction is equivalent to the “datatype used in a receive
 3 cannot specify overlapping regions” restriction for communication. Note that `filetypes` from
 4 different processes may still overlap each other.

5 If a `filetype` has holes in it, then the data in the holes is inaccessible to the calling
 6 process. However, the `disp`, `etype`, and `filetype` arguments can be changed via future calls to
 7 `MPI_FILE_SET_VIEW` to access a different part of the file.

8 It is erroneous to use absolute addresses in the construction of the `etype` and `filetype`.

9 The `info` argument is used to provide information regarding file access patterns and file
 10 system specifics to direct optimization (see Section 14.2.8). The constant `MPI_INFO_NULL`
 11 refers to the null `info` and can be used when no `info` needs to be specified.

12 The `datarep` argument is a string that specifies the representation of data in the file.
 13 See the file interoperability section (Section 14.5) for details and a discussion of valid values.

14 The user is responsible for ensuring that all nonblocking requests and split collective
 15 operations on `fh` have been completed before calling `MPI_FILE_SET_VIEW`—otherwise, the
 16 call to `MPI_FILE_SET_VIEW` is erroneous.

17
 18
 19 `MPI_FILE_GET_VIEW(fh, disp, etype, filetype, datarep)`

20	IN	<code>fh</code>	file handle (handle)
21	OUT	<code>disp</code>	displacement (integer)
22	OUT	<code>etype</code>	elementary datatype (handle)
23	OUT	<code>filetype</code>	filetype (handle)
24	OUT	<code>datarep</code>	data representation (string)

27 C binding

28 `int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype,`
 29 `MPI_Datatype *filetype, char *datarep)`

30 Fortran 2008 binding

31 `MPI_File_get_view(fh, disp, etype, filetype, datarep, ierror)`
 32 `TYPE(MPI_File), INTENT(IN) :: fh`
 33 `INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp`
 34 `TYPE(MPI_Datatype), INTENT(OUT) :: etype, filetype`
 35 `CHARACTER(LEN=*), INTENT(OUT) :: datarep`
 36 `INTEGER, OPTIONAL, INTENT(OUT) :: ierror`

37 Fortran binding

38 `MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)`
 39 `INTEGER FH, ETYPE, FILETYPE, IERROR`
 40 `INTEGER(KIND=MPI_OFFSET_KIND) DISP`
 41 `CHARACTER*(*) DATAREP`

42
 43
 44 `MPI_FILE_GET_VIEW` returns the process’s view of the data in the file. The current
 45 value of the displacement is returned in `disp`. The `etype` and `filetype` are new datatypes with
 46 typemaps equal to the typemaps of the current `etype` and `filetype`, respectively.

The data representation is returned in `datarep`. The user is responsible for ensuring that `datarep` is large enough to hold the returned data representation string. The length of a data representation string is limited to the value of `MPI_MAX_DATAREP_STRING`.

In addition, if a portable datatype was used to set the current view, then the corresponding datatype returned by `MPI_FILE_GET_VIEW` is also a portable datatype. If `etype` or `filetype` are derived datatypes, the user is responsible for freeing them. The `etype` and `filetype` returned are both in a committed state.

14.4 Data Access

14.4.1 Data Access Routines

Data is moved between files and processes by issuing read and write calls. There are three orthogonal aspects to data access: **positioning** (explicit offset vs. implicit file pointer), **synchronism** (blocking vs. nonblocking and split collective), and **coordination** (noncollective vs. collective). The following combinations of these data access routines, including two types of file pointers (individual and shared) are provided in Table 14.1.

Table 14.1: Data access routines

positioning	synchronism	coordination	
		noncollective	collective
explicit offsets	<i>blocking</i>	MPI_FILE_READ_AT MPI_FILE_WRITE_AT	MPI_FILE_READ_AT_ALL MPI_FILE_WRITE_AT_ALL
	<i>nonblocking</i>	MPI_FILE_IREAD_AT MPI_FILE_IWRITE_AT	MPI_FILE_IREAD_AT_ALL MPI_FILE_IWRITE_AT_ALL
	<i>split collective</i>	N/A	MPI_FILE_READ_AT_ALL_BEGIN MPI_FILE_READ_AT_ALL_END MPI_FILE_WRITE_AT_ALL_BEGIN MPI_FILE_WRITE_AT_ALL_END
individual file pointers	<i>blocking</i>	MPI_FILE_READ MPI_FILE_WRITE	MPI_FILE_READ_ALL MPI_FILE_WRITE_ALL
	<i>nonblocking</i>	MPI_FILE_IREAD MPI_FILE_IWRITE	MPI_FILE_IREAD_ALL MPI_FILE_IWRITE_ALL
	<i>split collective</i>	N/A	MPI_FILE_READ_ALL_BEGIN MPI_FILE_READ_ALL_END MPI_FILE_WRITE_ALL_BEGIN MPI_FILE_WRITE_ALL_END
shared file pointer	<i>blocking</i>	MPI_FILE_READ_SHARED MPI_FILE_WRITE_SHARED	MPI_FILE_READ_ORDERED MPI_FILE_WRITE_ORDERED
	<i>nonblocking</i>	MPI_FILE_IREAD_SHARED MPI_FILE_IWRITE_SHARED	N/A
	<i>split collective</i>	N/A	MPI_FILE_READ_ORDERED_BEGIN MPI_FILE_READ_ORDERED_END MPI_FILE_WRITE_ORDERED_BEGIN MPI_FILE_WRITE_ORDERED_END

POSIX `read()/pread()` 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

1 completes. For writes, however, the `MPI_FILE_SYNC` routine provides the only guarantee
 2 that data has been transferred to the storage device.

3 Positioning

4 MPI provides three types of positioning for data access routines: **explicit offsets**, **indi-**
 5 **vidual file pointers**, and **shared file pointers**. The different positioning methods may
 6 be mixed within the same program and do not affect each other.

7 The data access routines that accept explicit offsets contain `_AT` in their name (e.g.,
 8 `MPI_FILE_WRITE_AT`). Explicit offset operations perform data access at the file position
 9 given directly as an argument—no file pointer is used nor updated. Note that this is not
 10 equivalent to an atomic seek-and-read or seek-and-write operation, as no “seek” is issued.
 11 Operations with explicit offsets are described in Section 14.4.2.

12 The names of the individual file pointer routines contain no positional qualifier (e.g.,
 13 `MPI_FILE_WRITE`). Operations with individual file pointers are described in Section 14.4.3.
 14 The data access routines that use shared file pointers contain `_SHARED` or `_ORDERED`
 15 in their name (e.g., `MPI_FILE_WRITE_SHARED`). Operations with shared file pointers are
 16 described in Section 14.4.4.

17 The main semantic issues with MPI-maintained file pointers are how and when they are
 18 updated by I/O operations. In general, each I/O operation leaves the file pointer pointing to
 19 the next data item after the last one that is accessed by the operation. In a nonblocking or
 20 split collective operation, the pointer is updated by the call that initiates the I/O, possibly
 21 before the access completes.

22 More formally,

$$23 \quad new_file_offset = old_file_offset + \frac{elements(datatype)}{elements(etype)} \times count$$

24 where *count* is the number of *datatype* items to be accessed, *elements(X)* is the number of
 25 predefined datatypes in the typemap of *X*, and *old_file_offset* is the value of the implicit
 26 offset before the call. The file position, *new_file_offset*, is in terms of a count of etypes
 27 relative to the current view.

28 Synchronism

29 MPI supports blocking and nonblocking I/O routines.

30 A *blocking* I/O call will not return until the I/O request is completed.

31 A *nonblocking* I/O call initiates an I/O operation, but does not wait for it to complete.
 32 Given suitable hardware, this allows the transfer of data out of and into the user’s buffer
 33 to proceed concurrently with computation. A separate *request complete* call (`MPI_WAIT`,
 34 `MPI_TEST`, or any of their variants) is needed to complete the I/O request, i.e., to confirm
 35 that the data has been read or written and that it is safe for the user to reuse the buffer.
 36 The nonblocking versions of the routines are named `MPI_FILE_IXXX`, where the *I* stands for
 37 immediate.

38 It is erroneous to access the local buffer of a nonblocking data access operation, or to
 39 use that buffer as the source or target of other communications, between the initiation and
 40 completion of the operation.

41 The split collective routines support a restricted form of “nonblocking” operations for
 42 collective data access (see Section 14.4.5).

Coordination

Every noncollective data access routine `MPI_FILE_XXX` has a collective counterpart. For most routines, this counterpart is `MPI_FILE_XXX_ALL` or a pair of `MPI_FILE_XXX_BEGIN` and `MPI_FILE_XXX_END`. The counterparts to the `MPI_FILE_XXX_SHARED` routines are `MPI_FILE_XXX_ORDERED`.

The completion of a noncollective call only depends on the activity of the calling process. However, the completion of a collective call (which must be called by all members of the process group) may depend on the activity of the other processes participating in the collective call. See Section 14.6.4 for rules on semantics of collective calls.

Collective operations may perform much better than their noncollective counterparts, as global data accesses have significant potential for automatic optimization.

Data Access Conventions

Data is moved between files and processes by calling read and write routines. Read routines move data from a file into memory. Write routines move data from memory into a file. The file is designated by a file handle, `fh`. The location of the file data is specified by an offset into the current view. The data in memory is specified by a triple: `buf`, `count`, and `datatype`. Upon completion, the amount of data accessed by the calling process is returned in a `status`.

An offset designates the starting position in the file for an access. The offset is always in `etype` units relative to the current view. Explicit offset routines pass `offset` as an argument (negative values are erroneous). The file pointer routines use implicit offsets maintained by MPI.

A data access routine attempts to transfer (read or write) `count` data items of type `datatype` between the user's buffer `buf` and the file. The `datatype` passed to the routine must be a committed datatype. The layout of data in memory corresponding to `buf`, `count`, `datatype` is interpreted the same way as in MPI communication functions; see Section 3.2.2 and Section 5.1.11. The data is accessed from those parts of the file specified by the current view (Section 14.3). The type signature of `datatype` must match the type signature of some number of contiguous copies of the `etype` of the current view. As in a receive, it is erroneous to specify a `datatype` for reading that contains overlapping regions (areas of memory that would be stored into more than once).

The nonblocking data access routines indicate that MPI can start a data access and associate a request handle, `request`, with the I/O operation. Nonblocking operations are completed via `MPI_TEST`, `MPI_WAIT`, or any of their variants.

Data access operations, when completed, return the amount of data accessed in `status`.

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

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

1 status argument if the return value of this argument is not needed. The status can be
 2 passed to MPI_TEST_CANCELLED to determine if the operation was cancelled. All other
 3 fields of status are undefined.

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

9 14.4.2 Data Access with Explicit Offsets

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

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

13	IN	fh	file handle (handle)
14	IN	offset	file offset (integer)
15	OUT	buf	initial address of buffer (choice)
16	IN	count	number of elements in buffer (integer)
17	IN	datatype	datatype of each buffer element (handle)
18	OUT	status	status object (status)

19 C binding

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

22 int MPI_File_read_at_c(MPI_File fh, MPI_Offset offset, void *buf,
 23 MPI_Count count, MPI_Datatype datatype, MPI_Status *status)

24 Fortran 2008 binding

25 MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror)

26 TYPE(MPI_File), INTENT(IN) :: fh
 27 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
 28 TYPE(*), DIMENSION(..) :: buf
 29 INTEGER, INTENT(IN) :: count
 30 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 31 TYPE(MPI_Status) :: status
 32 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

33 MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror) !(_c)

34 TYPE(MPI_File), INTENT(IN) :: fh
 35 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
 36 TYPE(*), DIMENSION(..) :: buf
 37 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
 38 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 39 TYPE(MPI_Status) :: status
 40 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

```

MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
    <type> BUF(*)

```

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)

```

```

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

```

Fortran 2008 binding

```

MPI_File_read_at_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(fh, offset, buf, count, datatype, status, ierror) !(_c)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..) :: buf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
    <type> BUF(*)

```

1 MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT
2 interface.

3
4
5 MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status)

6	INOUT	fh	file handle (handle)
7	IN	offset	file offset (integer)
8	IN	buf	initial address of buffer (choice)
9	IN	count	number of elements in buffer (integer)
10	IN	datatype	datatype of each buffer element (handle)
11	IN	count	number of elements in buffer (integer)
12	IN	datatype	datatype of each buffer element (handle)
13	OUT	status	status object (status)

14 C binding

16 int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf,
17 int count, MPI_Datatype datatype, MPI_Status *status)

18 int MPI_File_write_at_c(MPI_File fh, MPI_Offset offset, const void *buf,
19 MPI_Count count, MPI_Datatype datatype, MPI_Status *status)

21 Fortran 2008 binding

22 MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror)
23 TYPE(MPI_File), INTENT(IN) :: fh
24 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
25 TYPE(*), DIMENSION(..), INTENT(IN) :: buf
26 INTEGER, INTENT(IN) :: count
27 TYPE(MPI_Datatype), INTENT(IN) :: datatype
28 TYPE(MPI_Status) :: status
29 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

30 MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror) !(_c)
31 TYPE(MPI_File), INTENT(IN) :: fh
32 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
33 TYPE(*), DIMENSION(..), INTENT(IN) :: buf
34 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
35 TYPE(MPI_Datatype), INTENT(IN) :: datatype
36 TYPE(MPI_Status) :: status
37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

39 Fortran binding

40 MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
41 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
42 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
43 <type> BUF(*)

44 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)			1
INOUT	fh	file handle (handle)	2
IN	offset	file offset (integer)	3
IN	buf	initial address of buffer (choice)	4
IN	count	number of elements in buffer (integer)	5
IN	datatype	datatype of each buffer element (handle)	6
OUT	status	status object (status)	7

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

```

Fortran 2008 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
MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror) !(_c)
  TYPE(MPI_File), INTENT(IN) :: fh
  INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
  TYPE(*), DIMENSION(..), INTENT(IN) :: buf
  INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Status) :: status
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

MPI_FILE_WRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
  INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
  INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
  <type> BUF(*)

```

MPI_FILE_WRITE_AT_ALL is a collective version of the blocking MPI_FILE_WRITE_AT interface.

```

1 MPI_FILE_IREAD_AT(fh, offset, buf, count, datatype, request)
2     IN      fh                file handle (handle)
3
4     IN      offset            file offset (integer)
5
6     OUT     buf                initial address of buffer (choice)
7
8     IN      count             number of elements in buffer (integer)
9
10    IN      datatype           datatype of each buffer element (handle)
11
12    OUT     request            request object (handle)

```

C binding

```

13 int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
14                      MPI_Datatype datatype, MPI_Request *request)
15
16 int MPI_File_iread_at_c(MPI_File fh, MPI_Offset offset, void *buf,
17                        MPI_Count count, MPI_Datatype datatype, MPI_Request *request)

```

Fortran 2008 binding

```

18 MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror)
19     TYPE(MPI_File), INTENT(IN) :: fh
20     INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
21     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
22     INTEGER, INTENT(IN) :: count
23     TYPE(MPI_Datatype), INTENT(IN) :: datatype
24     TYPE(MPI_Request), INTENT(OUT) :: request
25     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27 MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror) !(_c)
28     TYPE(MPI_File), INTENT(IN) :: fh
29     INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
30     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
31     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
32     TYPE(MPI_Datatype), INTENT(IN) :: datatype
33     TYPE(MPI_Request), INTENT(OUT) :: request
34     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

35 MPI_FILE_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
36     INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
37     INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
38     <type> BUF(*)

```

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)				1
IN	fh	file handle (handle)		2
IN	offset	file offset (integer)		3
OUT	buf	initial address of buffer (choice)		4
IN	count	number of elements in buffer (integer)		5
IN	datatype	datatype of each buffer element (handle)		6
OUT	request	request object (handle)		7
				8
				9
				10

C binding

```
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_at_all_c(MPI_File fh, MPI_Offset offset, void *buf,
                          MPI_Count count, MPI_Datatype datatype, MPI_Request *request)
```

Fortran 2008 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
```

```
MPI_File_iread_at_all(fh, offset, buf, count, datatype, request, ierror) !(_c)
```

```
TYPE(MPI_File), INTENT(IN) :: fh
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_FILE_IREAD_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
```

```
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
<type> BUF(*)
```

MPI_FILE_IREAD_AT_ALL is a nonblocking version of MPI_FILE_READ_AT_ALL. See Section 14.6.5 for semantics of nonblocking collective file operations.

```

1 MPI_FILE_IWRITE_AT(fh, offset, buf, count, datatype, request)
2     INOUT   fh                file handle (handle)
3
4     IN      offset            file offset (integer)
5
6     IN      buf               initial address of buffer (choice)
7
8     IN      count             number of elements in buffer (integer)
9
10    IN      datatype          datatype of each buffer element (handle)
11
12    OUT     request           request object (handle)

```

C binding

```

13 int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, const void *buf,
14                       int count, MPI_Datatype datatype, MPI_Request *request)
15
16 int MPI_File_iwrite_at_c(MPI_File fh, MPI_Offset offset, const void *buf,
17                          MPI_Count count, MPI_Datatype datatype, MPI_Request *request)

```

Fortran 2008 binding

```

18 MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror)
19     TYPE(MPI_File), INTENT(IN) :: fh
20     INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
21     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
22     INTEGER, INTENT(IN) :: count
23     TYPE(MPI_Datatype), INTENT(IN) :: datatype
24     TYPE(MPI_Request), INTENT(OUT) :: request
25     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27 MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror) !(_c)
28     TYPE(MPI_File), INTENT(IN) :: fh
29     INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
30     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
31     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
32     TYPE(MPI_Datatype), INTENT(IN) :: datatype
33     TYPE(MPI_Request), INTENT(OUT) :: request
34     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

35
36 MPI_FILE_IWRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
37     INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
38     INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
39     <type> BUF(*)
40

```

MPI_FILE_IWRITE_AT is a nonblocking version of the MPI_FILE_WRITE_AT interface.

MPI_FILE_IWRITE_AT_ALL(fh, offset, buf, count, datatype, request)			1
INOUT	fh	file handle (handle)	2
IN	offset	file offset (integer)	3
IN	buf	initial address of buffer (choice)	4
IN	count	number of elements in buffer (integer)	5
IN	datatype	datatype of each buffer element (handle)	6
OUT	request	request object (handle)	7

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)
int MPI_File_iwrite_at_all_c(MPI_File fh, MPI_Offset offset, const void *buf,
                             MPI_Count count, MPI_Datatype datatype, MPI_Request *request)

```

Fortran 2008 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
MPI_File_iwrite_at_all(fh, offset, buf, count, datatype, request, ierror) !(_c)
  TYPE(MPI_File), INTENT(IN) :: fh
  INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
  TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
  INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Request), INTENT(OUT) :: request
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

MPI_FILE_IWRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
  INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
  INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
  <type> BUF(*)

```

MPI_FILE_IWRITE_AT_ALL is a nonblocking version of MPI_FILE_WRITE_AT_ALL.

14.4.3 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 14.4.2, 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)
```

```
int MPI_File_read_c(MPI_File fh, void *buf, MPI_Count count,
                   MPI_Datatype datatype, MPI_Status *status)
```

Fortran 2008 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
```

```
MPI_File_read(fh, buf, count, datatype, status, ierror) !(_c)
  TYPE(MPI_File), INTENT(IN) :: fh
  TYPE(*), DIMENSION(..) :: buf
  INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Status) :: status
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
  INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
  <type> BUF(*)
```

MPI_FILE_READ reads a file using the individual file pointer.

Example 14.2. 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
end do

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)
```

```
int MPI_File_read_all_c(MPI_File fh, void *buf, MPI_Count count,
                      MPI_Datatype datatype, MPI_Status *status)
```

1 Fortran 2008 binding

```

2 MPI_File_read_all(fh, buf, count, datatype, status, ierror)
3     TYPE(MPI_File), INTENT(IN) :: fh
4     TYPE(*), DIMENSION(..) :: buf
5     INTEGER, INTENT(IN) :: count
6     TYPE(MPI_Datatype), INTENT(IN) :: datatype
7     TYPE(MPI_Status) :: status
8     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10 MPI_File_read_all(fh, buf, count, datatype, status, ierror) !(_c)
11     TYPE(MPI_File), INTENT(IN) :: fh
12     TYPE(*), DIMENSION(..) :: buf
13     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
14     TYPE(MPI_Datatype), INTENT(IN) :: datatype
15     TYPE(MPI_Status) :: status
16     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

17 Fortran binding

```

18 MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
19     INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
20     <type> BUF(*)

```

21 MPI_FILE_READ_ALL is a collective version of the blocking MPI_FILE_READ interface.

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

26	INOUT	fh	file handle (handle)
27	IN	buf	initial address of buffer (choice)
28	IN	count	number of elements in buffer (integer)
29	IN	datatype	datatype of each buffer element (handle)
30	IN	datatype	datatype of each buffer element (handle)
31	OUT	status	status object (status)

33 C binding

```

34 int MPI_File_write(MPI_File fh, const void *buf, int count,
35                   MPI_Datatype datatype, MPI_Status *status)
36
37 int MPI_File_write_c(MPI_File fh, const void *buf, MPI_Count count,
38                   MPI_Datatype datatype, MPI_Status *status)

```

39 Fortran 2008 binding

```

40 MPI_File_write(fh, buf, count, datatype, status, ierror)
41     TYPE(MPI_File), INTENT(IN) :: fh
42     TYPE(*), DIMENSION(..), INTENT(IN) :: buf
43     INTEGER, INTENT(IN) :: count
44     TYPE(MPI_Datatype), INTENT(IN) :: datatype
45     TYPE(MPI_Status) :: status
46     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48 MPI_File_write(fh, buf, count, datatype, status, ierror) !(_c)

```

```

TYPE(MPI_File), INTENT(IN) :: fh
TYPE(*), DIMENSION(..), INTENT(IN) :: buf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Status) :: status
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
<type> BUF(*)

```

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)
int MPI_File_write_all_c(MPI_File fh, const void *buf, MPI_Count count,
    MPI_Datatype datatype, MPI_Status *status)

```

Fortran 2008 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

```

MPI_File_write_all(fh, buf, count, datatype, status, ierror) !(_c)

```

TYPE(MPI_File), INTENT(IN) :: fh
TYPE(*), DIMENSION(..), INTENT(IN) :: buf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Status) :: status
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

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

```

1 <type> BUF(*)
 2
 3 MPI_FILE_WRITE_ALL is a collective version of the blocking MPI_FILE_WRITE inter-
 4 face.

5
 6 MPI_FILE_IREAD(fh, buf, count, datatype, request)
 7
 8 INOUT fh file handle (handle)
 9 OUT buf initial address of buffer (choice)
 10 IN count number of elements in buffer (integer)
 11 IN datatype datatype of each buffer element (handle)
 12 OUT request request object (handle)
 13

14 C binding

15 int MPI_File_iread(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
 16 MPI_Request *request)
 17
 18 int MPI_File_iread_c(MPI_File fh, void *buf, MPI_Count count,
 19 MPI_Datatype datatype, MPI_Request *request)
 20

21 Fortran 2008 binding

22 MPI_File_iread(fh, buf, count, datatype, request, ierror)
 23 TYPE(MPI_File), INTENT(IN) :: fh
 24 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
 25 INTEGER, INTENT(IN) :: count
 26 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 27 TYPE(MPI_Request), INTENT(OUT) :: request
 28 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 29
 30 MPI_File_iread(fh, buf, count, datatype, request, ierror) !(_c)
 31 TYPE(MPI_File), INTENT(IN) :: fh
 32 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
 33 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
 34 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 35 TYPE(MPI_Request), INTENT(OUT) :: request
 36 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

37 Fortran binding

38 MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
 39 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
 40 <type> BUF(*)
 41

42 MPI_FILE_IREAD is a nonblocking version of the MPI_FILE_READ interface.

43 **Example 14.3.** The following Fortran code fragment illustrates file pointer update se-
 44 mantics:

```
45        ! Read the first twenty real words in a file into two local
46        ! buffers. Note that when the first MPI_FILE_IREAD returns,
47        ! the file pointer has been updated to point to the
48
```



```

!   eleventh real word in the file.
1
2
3
integer    bufsize, req1, req2
4
integer, dimension(MPI_STATUS_SIZE) :: status1, status2
5
parameter (bufsize=10)
6
real      buf1(bufsize), buf2(bufsize)
7
integer(kind=MPI_OFFSET_KIND) zero
8
9
zero = 0
10
call MPI_FILE_OPEN(MPI_COMM_WORLD, 'myoldfile', &
11
                    MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr)
12
call MPI_FILE_SET_VIEW(myfh, zero, MPI_REAL, MPI_REAL, 'native', &
13
                        MPI_INFO_NULL, ierr)
14
call MPI_FILE_IREAD(myfh, buf1, bufsize, MPI_REAL, &
15
                    req1, ierr)
16
call MPI_FILE_IREAD(myfh, buf2, bufsize, MPI_REAL, &
17
                    req2, ierr)
18
19
call MPI_WAIT(req1, status1, ierr)
20
call MPI_WAIT(req2, status2, ierr)
21
22
call MPI_FILE_CLOSE(myfh, ierr)
23

```

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)

```

```

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

```

Fortran 2008 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

```

```

MPI_File_iread_all(fh, buf, count, datatype, request, ierror) !(_c)
TYPE(MPI_File), INTENT(IN) :: fh

```

```

1     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
2     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
3     TYPE(MPI_Datatype), INTENT(IN) :: datatype
4     TYPE(MPI_Request), INTENT(OUT) :: request
5     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

7 MPI_FILE_IREAD_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
8     INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
9     <type> BUF(*)

```

11 MPI_FILE_IREAD_ALL is a nonblocking version of MPI_FILE_READ_ALL.

```

13 MPI_FILE_IWRITE(fh, buf, count, datatype, request)

```

15	INOUT	fh	file handle (handle)
16	IN	buf	initial address of buffer (choice)
17	IN	count	number of elements in buffer (integer)
18	IN	datatype	datatype of each buffer element (handle)
19	IN	request	request object (handle)
20	OUT		
21			

C binding

```

22 int MPI_File_iread_all(MPI_File fh, void *buf, int count,
23     MPI_Datatype datatype, MPI_Request *request)
24
25 int MPI_File_iread_all_c(MPI_File fh, void *buf, MPI_Count count,
26     MPI_Datatype datatype, MPI_Request *request)
27
28

```

Fortran 2008 binding

```

29 MPI_FILE_IWRITE(fh, buf, count, datatype, request, ierror)
30     TYPE(MPI_File), INTENT(IN) :: fh
31     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
32     INTEGER, INTENT(IN) :: count
33     TYPE(MPI_Datatype), INTENT(IN) :: datatype
34     TYPE(MPI_Request), INTENT(OUT) :: request
35     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37 MPI_FILE_IWRITE(fh, buf, count, datatype, request, ierror) !(_c)
38     TYPE(MPI_File), INTENT(IN) :: fh
39     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
40     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
41     TYPE(MPI_Datatype), INTENT(IN) :: datatype
42     TYPE(MPI_Request), INTENT(OUT) :: request
43     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44

```

Fortran binding

```

45 MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
46     INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
47     <type> BUF(*)
48

```

MPI_FILE_IWRITE is a nonblocking version of MPI_FILE_WRITE.

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)
```

```
int MPI_File_iwrite_all_c(MPI_File fh, const void *buf, MPI_Count count,
                          MPI_Datatype datatype, MPI_Request *request)
```

Fortran 2008 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
```

```
MPI_File_iwrite_all(fh, buf, count, datatype, request, ierror) !(_c)
```

```
TYPE(MPI_File), INTENT(IN) :: fh
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_FILE_IWRITE_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
```

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

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)
```

Fortran 2008 binding

```
MPI_File_seek(fh, offset, whence, ierror)
  TYPE(MPI_File), INTENT(IN) :: fh
  INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
  INTEGER, INTENT(IN) :: whence
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran 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)
```

Fortran 2008 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
```

Fortran 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 ebyte 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)
```

Fortran 2008 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
```

Fortran 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`.

14.4.4 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 14.4.2, 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.

1 After a shared file pointer operation is initiated, the shared file pointer is updated to
 2 point to the next etype after the last one that will be accessed. The file pointer is updated
 3 relative to the current view of the file.
 4

5 Noncollective Operations

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

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

17 C binding

```
18 int MPI_File_read_shared(MPI_File fh, void *buf, int count,
19 MPI_Datatype datatype, MPI_Status *status)
```

```
20
21 int MPI_File_read_shared_c(MPI_File fh, void *buf, MPI_Count count,
22 MPI_Datatype datatype, MPI_Status *status)
```

23 Fortran 2008 binding

```
24 MPI_File_read_shared(fh, buf, count, datatype, status, ierror)
25 TYPE(MPI_File), INTENT(IN) :: fh
26 TYPE(*), DIMENSION(..) :: buf
27 INTEGER, INTENT(IN) :: count
28 TYPE(MPI_Datatype), INTENT(IN) :: datatype
29 TYPE(MPI_Status) :: status
30 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32 MPI_File_read_shared(fh, buf, count, datatype, status, ierror) !(_c)
33 TYPE(MPI_File), INTENT(IN) :: fh
34 TYPE(*), DIMENSION(..) :: buf
35 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
36 TYPE(MPI_Datatype), INTENT(IN) :: datatype
37 TYPE(MPI_Status) :: status
38 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

39 Fortran binding

```
40 MPI_FILE_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
41 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
42 <type> BUF(*)
```

43 MPI_FILE_READ_SHARED reads a file using the shared file pointer.
 44
 45
 46
 47
 48

MPI_FILE_WRITE_SHARED(fh, buf, count, datatype, status)			1
INOUT	fh	file handle (handle)	2
IN	buf	initial address of buffer (choice)	3
IN	count	number of elements in buffer (integer)	4
IN	datatype	datatype of each buffer element (handle)	5
OUT	status	status object (status)	6

C binding

```

int MPI_File_write_shared(MPI_File fh, const void *buf, int count,
                          MPI_Datatype datatype, MPI_Status *status)
int MPI_File_write_shared_c(MPI_File fh, const void *buf, MPI_Count count,
                            MPI_Datatype datatype, MPI_Status *status)

```

Fortran 2008 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
MPI_File_write_shared(fh, buf, count, datatype, status, ierror) !(_c)
  TYPE(MPI_File), INTENT(IN) :: fh
  TYPE(*), DIMENSION(..), INTENT(IN) :: buf
  INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Status) :: status
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
  INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
  <type> BUF(*)
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)	40
OUT	buf	initial address of buffer (choice)	41
IN	count	number of elements in buffer (integer)	42
IN	datatype	datatype of each buffer element (handle)	43
OUT	request	request object (handle)	44

```

1  C binding
2  int MPI_File_iread_shared(MPI_File fh, void *buf, int count,
3      MPI_Datatype datatype, MPI_Request *request)
4
5  int MPI_File_iread_shared_c(MPI_File fh, void *buf, MPI_Count count,
6      MPI_Datatype datatype, MPI_Request *request)
7
8  Fortran 2008 binding
9  MPI_File_iread_shared(fh, buf, count, datatype, request, ierror)
10     TYPE(MPI_File), INTENT(IN) :: fh
11     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
12     INTEGER, INTENT(IN) :: count
13     TYPE(MPI_Datatype), INTENT(IN) :: datatype
14     TYPE(MPI_Request), INTENT(OUT) :: request
15     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17 MPI_File_iread_shared(fh, buf, count, datatype, request, ierror) !(_c)
18     TYPE(MPI_File), INTENT(IN) :: fh
19     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
20     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
21     TYPE(MPI_Datatype), INTENT(IN) :: datatype
22     TYPE(MPI_Request), INTENT(OUT) :: request
23     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
25 Fortran binding
26 MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
27     INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
28     <type> BUF(*)
29
30     MPI_FILE_IREAD_SHARED is a nonblocking version of MPI_FILE_READ_SHARED.
31
32 MPI_FILE_IWRITE_SHARED(fh, buf, count, datatype, request)
33
34     INOUT   fh           file handle (handle)
35     IN      buf          initial address of buffer (choice)
36     IN      count        number of elements in buffer (integer)
37     IN      datatype     datatype of each buffer element (handle)
38     OUT     request      request object (handle)
39
40 C binding
41 int MPI_File_iwrite_shared(MPI_File fh, const void *buf, int count,
42     MPI_Datatype datatype, MPI_Request *request)
43
44 int MPI_File_iwrite_shared_c(MPI_File fh, const void *buf, MPI_Count count,
45     MPI_Datatype datatype, MPI_Request *request)
46
47 Fortran 2008 binding
48 MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror)
49     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_shared(fh, buf, count, datatype, request, ierror) !(_c)
TYPE(MPI_File), INTENT(IN) :: fh
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

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

```

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.*)

```

1 MPI_FILE_READ_ORDERED(fh, buf, count, datatype, status)
2   INOUT   fh           file handle (handle)
3
4   OUT     buf          initial address of buffer (choice)
5
6   IN      count        number of elements in buffer (integer)
7
8   IN      datatype     datatype of each buffer element (handle)
9
10  OUT     status       status object (status)

```

10 C binding

```

11 int MPI_File_read_ordered(MPI_File fh, void *buf, int count,
12                          MPI_Datatype datatype, MPI_Status *status)
13
14 int MPI_File_read_ordered_c(MPI_File fh, void *buf, MPI_Count count,
15                             MPI_Datatype datatype, MPI_Status *status)

```

16 Fortran 2008 binding

```

17 MPI_File_read_ordered(fh, buf, count, datatype, status, ierror)
18   TYPE(MPI_File), INTENT(IN) :: fh
19   TYPE(*), DIMENSION(..) :: buf
20   INTEGER, INTENT(IN) :: count
21   TYPE(MPI_Datatype), INTENT(IN) :: datatype
22   TYPE(MPI_Status) :: status
23   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
25 MPI_File_read_ordered(fh, buf, count, datatype, status, ierror) !(_c)
26   TYPE(MPI_File), INTENT(IN) :: fh
27   TYPE(*), DIMENSION(..) :: buf
28   INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
29   TYPE(MPI_Datatype), INTENT(IN) :: datatype
30   TYPE(MPI_Status) :: status
31   INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

32 Fortran binding

```

33 MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
34   INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
35   <type> BUF(*)

```

36 MPI_FILE_READ_ORDERED is a collective version of the MPI_FILE_READ_SHARED
37 interface.
38

```

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

```

```

41   INOUT   fh           file handle (handle)
42
43   IN      buf          initial address of buffer (choice)
44
45   IN      count        number of elements in buffer (integer)
46
47   IN      datatype     datatype of each buffer element (handle)
48
49   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)
int MPI_File_write_ordered_c(MPI_File fh, const void *buf, MPI_Count count,
                             MPI_Datatype datatype, MPI_Status *status)

```

Fortran 2008 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
MPI_File_write_ordered(fh, buf, count, datatype, status, ierror) !(_c)
  TYPE(MPI_File), INTENT(IN) :: fh
  TYPE(*), DIMENSION(..), INTENT(IN) :: buf
  INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Status) :: status
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
  INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
  <type> BUF(*)

```

`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)

```

Fortran 2008 binding

```

MPI_File_seek_shared(fh, offset, whence, ierror)
  TYPE(MPI_File), INTENT(IN) :: fh

```

```

1     INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
2     INTEGER, INTENT(IN) :: whence
3     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4

```

Fortran binding

```

5 MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR)
6     INTEGER FH, WHENCE, IERROR
7     INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
8

```

9 MPI_FILE_SEEK_SHARED updates the shared file pointer according to whence, which
10 has the following possible values:

11 MPI_SEEK_SET	the pointer is set to offset
12 MPI_SEEK_CUR	the pointer is set to the current pointer position 13 plus offset
14 MPI_SEEK_END	the pointer is set to the end of file plus offset

15 MPI_FILE_SEEK_SHARED is collective; all the processes in the communicator group
16 associated with the file handle fh must call MPI_FILE_SEEK_SHARED with the same values
17 for offset and whence.

18 The offset can be negative, which allows seeking backwards. It is erroneous to seek to
19 a negative position in the view.
20

```

21
22 MPI_FILE_GET_POSITION_SHARED(fh, offset)
23
24     IN      fh          file handle (handle)
25     OUT    offset      offset of shared pointer (integer)
26

```

C binding

```

27 int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)
28
29

```

Fortran 2008 binding

```

30 MPI_File_get_position_shared(fh, offset, ierror)
31     TYPE(MPI_File), INTENT(IN) :: fh
32     INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset
33     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34

```

Fortran binding

```

35 MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERROR)
36     INTEGER FH, IERROR
37     INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
38
39

```

40 MPI_FILE_GET_POSITION_SHARED returns, in offset, the current position of the
41 shared file pointer in etype units relative to the current view.

42 *Advice to users.* The offset can be used in a future call to MPI_FILE_SEEK_SHARED
43 using whence = MPI_SEEK_SET to return to the current position. To set the displacement
44 to the current file pointer position, first convert offset into an absolute byte
45 position using MPI_FILE_GET_BYTE_OFFSET, then call MPI_FILE_SET_VIEW with
46 the resulting displacement. (*End of advice to users.*)
47

48

14.4.5 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.

According to the definitions in Section 2.4.2, the begin procedures are incomplete. They are also nonlocal procedures because they may or may not return before they are called in all MPI processes of the process group.

Advice to users. This is one of the exceptions in which incomplete procedures are nonlocal and therefore blocking. (*End of advice to users.*)

- Split collective operations do not match the corresponding regular collective operation. For example, in a single collective read operation, an `MPI_FILE_READ_ALL` on one process does not match an `MPI_FILE_READ_ALL_BEGIN`/`MPI_FILE_READ_ALL_END` pair on another process.
- Split collective routines must specify a buffer in both the begin and end routines. By specifying the buffer that receives data in the end routine, we can avoid the problems described in [Problems with Code Movement and Register Optimization](#), Section 19.1.17, but not all of the problems, such as those described in Sections 19.1.12, 19.1.13, and 19.1.16.
- No collective I/O operations are permitted on a file handle concurrently with a split collective access on that file handle (i.e., between the begin and end of the access). That is

```

1  MPI_File_read_all_begin(fh, ...);
2  ...
3  MPI_File_read_all(fh, ...);
4  ...
5  MPI_File_read_all_end(fh, ...);

```

is erroneous.

- In a multithreaded implementation, any split collective begin and end operation called by a process must be called from the same thread. This restriction is made to simplify the implementation in the multithreaded case. (Note that we have already disallowed having two threads begin a split collective operation on the same file handle since only one split collective operation can be active on a file handle at any time.)

The arguments for these routines have the same meaning as for the equivalent collective versions (e.g., the argument definitions for `MPI_FILE_READ_ALL_BEGIN` and `MPI_FILE_READ_ALL_END` are equivalent to the arguments for `MPI_FILE_READ_ALL`). The begin routine (e.g., `MPI_FILE_READ_ALL_BEGIN`) begins a split collective operation that, when completed with the matching end routine (i.e., `MPI_FILE_READ_ALL_END`) produces the result as defined for the equivalent collective routine (i.e., `MPI_FILE_READ_ALL`).

For the purpose of consistency semantics (Section 14.6.1), a matched pair of split collective data access operations (e.g., `MPI_FILE_READ_ALL_BEGIN` and `MPI_FILE_READ_ALL_END`) compose a single data access.

`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)

```

```

int MPI_File_read_at_all_begin_c(MPI_File fh, MPI_Offset offset, void *buf,
                                 MPI_Count count, MPI_Datatype datatype)

```

Fortran 2008 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

```

```

MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror) !(_c) 1
    TYPE(MPI_File), INTENT(IN) :: fh 2
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset 3
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf 4
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count 5
    TYPE(MPI_Datatype), INTENT(IN) :: datatype 6
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror 7
 8
Fortran binding 9
MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR) 10
    INTEGER FH, COUNT, DATATYPE, IERROR 11
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET 12
    <type> BUF(*) 13
 14
MPI_FILE_READ_AT_ALL_END(fh, buf, status) 15
 16
    IN      fh      file handle (handle) 17
    OUT     buf      initial address of buffer (choice) 18
    OUT     status   status object (status) 19
 20
 21
C binding 22
int MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status) 23
 24
Fortran 2008 binding 25
MPI_File_read_at_all_end(fh, buf, status, ierror) 26
    TYPE(MPI_File), INTENT(IN) :: fh 27
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf 28
    TYPE(MPI_Status) :: status 29
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror 30
 31
Fortran binding 31
MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR) 32
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR 33
    <type> BUF(*) 34
 35
 36
MPI_FILE_WRITE_AT_ALL_BEGIN(fh, offset, buf, count, datatype) 37
 38
    INOUT   fh      file handle (handle) 39
    IN      offset   file offset (integer) 40
    IN      buf      initial address of buffer (choice) 41
    IN      count    number of elements in buffer (integer) 42
    IN      datatype datatype of each buffer element (handle) 43
 44
 45
C binding 46
int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset, 47
    const void *buf, int count, MPI_Datatype datatype) 48

```

```

1  int MPI_File_write_at_all_begin_c(MPI_File fh, MPI_Offset offset,
2      const void *buf, MPI_Count count, MPI_Datatype datatype)
3
4  Fortran 2008 binding
5  MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror)
6      TYPE(MPI_File), INTENT(IN) :: fh
7      INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
8      TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
9      INTEGER, INTENT(IN) :: count
10     TYPE(MPI_Datatype), INTENT(IN) :: datatype
11     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13 MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror) !(_c)
14     TYPE(MPI_File), INTENT(IN) :: fh
15     INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
16     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
17     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
18     TYPE(MPI_Datatype), INTENT(IN) :: datatype
19     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21 Fortran binding
22 MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
23     INTEGER FH, COUNT, DATATYPE, IERROR
24     INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
25     <type> BUF(*)
26
27 MPI_FILE_WRITE_AT_ALL_END(fh, buf, status)
28     INOUT   fh           file handle (handle)
29     IN      buf          initial address of buffer (choice)
30     OUT     status       status object (status)
31
32
33 C binding
34 int MPI_File_write_at_all_end(MPI_File fh, const void *buf, MPI_Status *status)
35
36 Fortran 2008 binding
37 MPI_File_write_at_all_end(fh, buf, status, ierror)
38     TYPE(MPI_File), INTENT(IN) :: fh
39     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
40     TYPE(MPI_Status) :: status
41     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43 Fortran binding
44 MPI_FILE_WRITE_AT_ALL_END(FH, BUF, STATUS, IERROR)
45     INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
46     <type> BUF(*)
47
48

```


MPI_FILE_READ_ALL_BEGIN(fh, buf, count, datatype)	1
INOUT fh file handle (handle)	2
OUT buf initial address of buffer (choice)	3
IN count number of elements in buffer (integer)	4
IN datatype datatype of each buffer element (handle)	5
	6
	7
	8
C binding	9
int MPI_File_read_all_begin(MPI_File fh, void *buf, int count,	10
MPI_Datatype datatype)	11
int MPI_File_read_all_begin_c(MPI_File fh, void *buf, MPI_Count count,	12
MPI_Datatype datatype)	13
	14
Fortran 2008 binding	15
MPI_File_read_all_begin(fh, buf, count, datatype, ierror)	16
TYPE(MPI_File), INTENT(IN) :: fh	17
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf	18
INTEGER, INTENT(IN) :: count	19
TYPE(MPI_Datatype), INTENT(IN) :: datatype	20
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	21
MPI_File_read_all_begin(fh, buf, count, datatype, ierror) !(_c)	22
TYPE(MPI_File), INTENT(IN) :: fh	23
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf	24
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count	25
TYPE(MPI_Datatype), INTENT(IN) :: datatype	26
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	27
	28
Fortran binding	29
MPI_FILE_READ_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)	30
INTEGER FH, COUNT, DATATYPE, IERROR	31
<type> BUF(*)	32
	33
	34
MPI_FILE_READ_ALL_END(fh, buf, status)	35
INOUT fh file handle (handle)	36
OUT buf initial address of buffer (choice)	37
OUT status status object (status)	38
	39
	40
	41
C binding	42
int MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status)	43
	44
Fortran 2008 binding	45
MPI_File_read_all_end(fh, buf, status, ierror)	46
TYPE(MPI_File), INTENT(IN) :: fh	47
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf	48
TYPE(MPI_Status) :: status	49

1 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

2

3 **Fortran binding**

4 **MPI_FILE_READ_ALL_END**(FH, BUF, STATUS, IERROR)
5 INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
6 <type> BUF(*)

7

8

9 **MPI_FILE_WRITE_ALL_BEGIN**(fh, buf, count, datatype)

10 INOUT fh file handle (handle)
11 IN buf initial address of buffer (choice)
12 IN count number of elements in buffer (integer)
13 IN datatype datatype of each buffer element (handle)

14

16 **C binding**

17 **int MPI_File_write_all_begin**(MPI_File fh, const void *buf, int count,
18 MPI_Datatype datatype)
19
20 **int MPI_File_write_all_begin_c**(MPI_File fh, const void *buf, MPI_Count count,
21 MPI_Datatype datatype)

22 **Fortran 2008 binding**

23 **MPI_File_write_all_begin**(fh, buf, count, datatype, ierror)
24 TYPE(MPI_File), INTENT(IN) :: fh
25 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
26 INTEGER, INTENT(IN) :: count
27 TYPE(MPI_Datatype), INTENT(IN) :: datatype
28 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30 **MPI_File_write_all_begin**(fh, buf, count, datatype, ierror) !(_c)
31 TYPE(MPI_File), INTENT(IN) :: fh
32 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
33 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
34 TYPE(MPI_Datatype), INTENT(IN) :: datatype
35 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

36 **Fortran binding**

37 **MPI_FILE_WRITE_ALL_BEGIN**(FH, BUF, COUNT, DATATYPE, IERROR)
38 INTEGER FH, COUNT, DATATYPE, IERROR
39 <type> BUF(*)

40

41

42 **MPI_FILE_WRITE_ALL_END**(fh, buf, status)

44 INOUT fh file handle (handle)
45 IN buf initial address of buffer (choice)
46 OUT status status object (status)

47

48

C binding

```
int MPI_File_write_all_end(MPI_File fh, const void *buf, MPI_Status *status)
```

Fortran 2008 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
```

Fortran binding

```
MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
    <type> BUF(*)
```

```
MPI_FILE_READ_ORDERED_BEGIN(fh, buf, count, datatype)
```

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

C binding

```
int MPI_File_read_ordered_begin(MPI_File fh, void *buf, int count,
    MPI_Datatype datatype)
```

```
int MPI_File_read_ordered_begin_c(MPI_File fh, void *buf, MPI_Count count,
    MPI_Datatype datatype)
```

Fortran 2008 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
```

```
MPI_File_read_ordered_begin(fh, buf, count, datatype, ierror) !(_c)
```

```
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_FILE_READ_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
    INTEGER FH, COUNT, DATATYPE, IERROR
    <type> BUF(*)
```

```

1 MPI_FILE_READ_ORDERED_END(fh, buf, status)
2     INOUT   fh                file handle (handle)
3
4     OUT     buf                initial address of buffer (choice)
5
6     OUT     status             status object (status)
7
8 C binding
9 int MPI_File_read_ordered_end(MPI_File fh, void *buf, MPI_Status *status)
10
11 Fortran 2008 binding
12 MPI_File_read_ordered_end(fh, buf, status, ierror)
13     TYPE(MPI_File), INTENT(IN) :: fh
14     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
15     TYPE(MPI_Status) :: status
16     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18 Fortran binding
19 MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR)
20     INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
21     <type> BUF(*)
22
23 MPI_FILE_WRITE_ORDERED_BEGIN(fh, buf, count, datatype)
24     INOUT   fh                file handle (handle)
25
26     IN      buf                initial address of buffer (choice)
27
28     IN      count              number of elements in buffer (integer)
29
30     IN      datatype           datatype of each buffer element (handle)
31
32 C binding
33 int MPI_File_write_ordered_begin(MPI_File fh, const void *buf, int count,
34     MPI_Datatype datatype)
35
36 int MPI_File_write_ordered_begin_c(MPI_File fh, const void *buf,
37     MPI_Count count, MPI_Datatype datatype)
38
39 Fortran 2008 binding
40 MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror)
41     TYPE(MPI_File), INTENT(IN) :: fh
42     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
43     INTEGER, INTENT(IN) :: count
44     TYPE(MPI_Datatype), INTENT(IN) :: datatype
45     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47 MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror) !(_c)
48     TYPE(MPI_File), INTENT(IN) :: fh
49     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
50     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
51     TYPE(MPI_Datatype), INTENT(IN) :: datatype

```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1
Fortran binding	2
MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)	3
INTEGER FH, COUNT, DATATYPE, IERROR	4
<type> BUF(*)	5
	6
	7
	8
MPI_FILE_WRITE_ORDERED_END(fh, buf, status)	9
INOUT fh file handle (handle)	10
IN buf initial address of buffer (choice)	11
OUT status status object (status)	12
	13
	14
C binding	15
int MPI_File_write_ordered_end(MPI_File fh, const void *buf,	16
MPI_Status *status)	17
	18
Fortran 2008 binding	19
MPI_File_write_ordered_end(fh, buf, status, ierror)	20
TYPE(MPI_File), INTENT(IN) :: fh	21
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf	22
TYPE(MPI_Status) :: status	23
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	24
	25
Fortran binding	26
MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)	27
INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR	28
<type> BUF(*)	29
	30

14.5 File Interoperability

At the most basic level, file interoperability is the ability to read the information previously written to a file—not just the bits of data, but the actual information the bits represent. MPI guarantees full interoperability within a single MPI environment, and supports increased interoperability outside that environment through the external data representation (Section 14.5.2) as well as the data conversion functions (Section 14.5.3).

Interoperability within a single MPI environment (which could be considered “operability”) ensures that file data written by one MPI process can be read by any other MPI process, subject to the consistency constraints (see Section 14.6.1), provided that it would have been possible to start the two processes simultaneously and have them reside in a single MPI_COMM_WORLD. Furthermore, both processes must see the same data values at every absolute byte offset in the file for which data was written.

This single environment file interoperability implies that file data is accessible regardless of the number of processes.

There are three aspects to file interoperability:

- transferring the bits,

- 1 • converting between different file structures, and
- 2
- 3 • converting between different machine representations.

4 The first two aspects of file interoperability are beyond the scope of this standard,
5 as both are highly machine dependent. However, transferring the bits of a file into and
6 out of the MPI environment (e.g., by writing a file to tape) is required to be supported
7 by all MPI implementations. In particular, an implementation must specify how familiar
8 operations similar to POSIX `cp`, `rm`, and `mv` can be performed on the file. Furthermore, it
9 is expected that the facility provided maintains the correspondence between absolute byte
10 offsets (e.g., after possible file structure conversion, the data bits at byte offset 102 in the
11 MPI environment are at byte offset 102 outside the MPI environment). As an example,
12 a simple off-line conversion utility that transfers and converts files between the native file
13 system and the MPI environment would suffice, provided it maintained the offset coherence
14 mentioned above. In a high-quality implementation of MPI, users will be able to manipulate
15 MPI files using the same or similar tools that the native file system offers for manipulating
16 its files.

17 The remaining aspect of file interoperability, converting between different machine
18 representations, is supported by the typing information specified in the `etype` and `filetype`.
19 This facility allows the information in files to be shared between any two applications,
20 regardless of whether they use MPI, and regardless of the machine architectures on which
21 they run.

22 MPI supports multiple data representations: "native", "internal", and "external32". An im-
23 plementation may support additional data representations. MPI also supports user-defined
24 data representations (see Section 14.5.3). The "native" and "internal" data representations
25 are implementation dependent, while the "external32" representation is common to all MPI
26 implementations and facilitates file interoperability. The data representation is specified in
27 the `datatype` argument to `MPI_FILE_SET_VIEW`.

28
29 *Advice to users.* MPI is not guaranteed to retain knowledge of what data representa-
30 tion was used when a file is written. Therefore, to correctly retrieve file data, an MPI
31 application is responsible for specifying the same data representation as was used to
32 create the file. (*End of advice to users.*)

33
34 **"native":** Data in this representation is stored in a file exactly as it is in memory. The ad-
35 vantage of this data representation is that data precision and I/O performance are not
36 lost in type conversions with a purely homogeneous environment. The disadvantage
37 is the loss of transparent interoperability within a heterogeneous MPI environment.

38 *Advice to users.* This data representation should only be used in a homogeneous
39 MPI environment, or when the MPI application is capable of performing the
40 datatype conversions itself. (*End of advice to users.*)

41 *Advice to implementors.* When implementing read and write operations on
42 top of MPI message-passing, the message data should be typed as `MPI_BYTE` to
43 ensure that the message routines do not perform any type conversions on the
44 data. (*End of advice to implementors.*)

45
46 **"internal":** This data representation can be used for I/O operations in a homogeneous or
47 heterogeneous environment; the implementation will perform type conversions if nec-
48 essary. 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 14.5.2. The data conversion rules for communication also apply to these conversions (see Section 3.3.2). The data on the storage medium is always in this canonical representation, and the data in memory is always in the local process's native representation.

This data representation has several advantages. First, all processes reading the file in a heterogeneous MPI environment will automatically have the data converted to their respective native representations. Second, the file can be exported from one MPI environment and imported into any other MPI environment with the guarantee that the second environment will be able to read all the data in the file.

The disadvantage of this data representation is that data precision and I/O performance may be lost in datatype conversions.

Advice to implementors. When implementing read and write operations on top of MPI message-passing, the message data should be converted to and from the "external32" representation in the client, and sent as type MPI_BYTE. This will avoid possible double datatype conversions and the associated further loss of precision and performance. (*End of advice to implementors.*)

14.5.1 Datatypes for File Interoperability

If the file data representation is other than "native", care must be taken in constructing etypes and filetypes. Any of the datatype constructor functions may be used; however, for those functions that accept displacements in bytes, the displacements must be specified in terms of their values in the file for the file data representation being used. MPI will interpret these byte displacements as is; no scaling will be done. The function MPI_FILE_GET_TYPE_EXTENT can be used to calculate the extents of datatypes in the file. For etypes and filetypes that are portable datatypes (see Section 2.4), MPI will scale any displacements in the datatypes to match the file data representation. Datatypes passed as arguments to read/write routines specify the data layout in memory; therefore, they must always be constructed using displacements corresponding to displacements in memory.

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

1 layout on the file as they would define in the memory of the calling process. If the
 2 `etype` and `filetype` are portable datatypes, then the data layout defined in the file is
 3 the same as would be defined in the calling process memory, up to a scaling factor.
 4 The routine `MPI_FILE_GET_TYPE_EXTENT` can be used to calculate this scaling
 5 factor. Thus, two equivalent, portable datatypes will define the same data layout
 6 in the file, even in a heterogeneous environment with "internal", "external32", or user
 7 defined data representations. Otherwise, the `etype` and `filetype` must be constructed
 8 so that their typemap and extent are the same on any architecture. This can be
 9 achieved if they have an explicit upper bound and lower bound (defined using
 10 `MPI_TYPE_CREATE_RESIZED`). This condition must also be fulfilled by any datatype
 11 that is used in the construction of the `etype` and `filetype`, if this datatype is replicated
 12 contiguously, either explicitly, by a call to `MPI_TYPE_CONTIGUOUS`, or implicitly,
 13 by a blocklength argument that is greater than one. If an `etype` or `filetype` is not
 14 portable, and has a typemap or extent that is architecture dependent, then the data
 15 layout specified by it on a file is implementation dependent.

16 File data representations other than "native" may be different from corresponding
 17 data representations in memory. Therefore, for these file data representations, it is
 18 important not to use hardwired byte offsets for file positioning, including the initial
 19 displacement that specifies the view. When a portable datatype (see Section 2.4) is
 20 used in a data access operation, any holes in the datatype are scaled to match the data
 21 representation. However, note that this technique only works when all the processes
 22 that created the file view build their `etypes` from the same predefined datatypes. For
 23 example, if one process uses an `etype` built from `MPI_INT` and another uses an `etype`
 24 built from `MPI_FLOAT`, the resulting views may be nonportable because the relative
 25 sizes of these types may differ from one data representation to another. (*End of advice*
 26 *to users.*)

27
 28
 29
 30 `MPI_FILE_GET_TYPE_EXTENT(fh, datatype, extent)`

31	IN	<code>fh</code>	file handle (handle)
32			
33	IN	<code>datatype</code>	datatype (handle)
34	OUT	<code>extent</code>	datatype extent (integer)

35 C binding

36
 37 `int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype,`
 38 `MPI_Aint *extent)`

39
 40 `int MPI_File_get_type_extent_c(MPI_File fh, MPI_Datatype datatype,`
 41 `MPI_Count *extent)`

42 Fortran 2008 binding

43 `MPI_File_get_type_extent(fh, datatype, extent, ierror)`
 44 `TYPE(MPI_File), INTENT(IN) :: fh`
 45 `TYPE(MPI_Datatype), INTENT(IN) :: datatype`
 46 `INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: extent`
 47 `INTEGER, OPTIONAL, INTENT(OUT) :: ierror`
 48


```

MPI_File_get_type_extent(fh, datatype, extent, ierror) !(_c)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: extent
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran 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 14.5.3), MPI uses the `dtype_file_extent_fn` callback to calculate the extent.

If the `datatype` extent cannot be represented in `extent`, it is set to `MPI_UNDEFINED`.

Advice to implementors. In the case of user-defined data representations, the extent of a derived datatype can be calculated by first determining the extents of the predefined datatypes in this derived datatype using `dtype_file_extent_fn` (see Section 14.5.3). (*End of advice to implementors.*)

14.5.2 External Data Representation: "external32"

All MPI implementations are required to support the data representation defined in this section. Support of optional datatypes (e.g., `MPI_INTEGER2`) is not required.

All floating point values are in big-endian IEEE format [42] of the appropriate size. Floating point values are represented by one of three IEEE formats. These are the IEEE "Single (binary32)," "Double (binary64)," and "Double Extended (binary128)" formats, requiring 4, 8, and 16 bytes of storage, respectively. For the IEEE "Double Extended (binary128)" formats, MPI specifies a Format Width of 16 bytes, with 15 exponent bits, bias = +16383, 112 fraction bits, and an encoding analogous to the "Double (binary64)" format. All integral values are in two's complement big-endian format. Big-endian means most significant byte at lowest address byte. For C `_Bool`, Fortran `LOGICAL`, and C++ `bool`, 0 implies false and nonzero implies true. C `float` `_Complex`, `double` `_Complex`, and `long double` `_Complex`, Fortran `COMPLEX` and `DOUBLE COMPLEX`, and other complex types are represented by a pair of floating point format values for the real and imaginary components. Characters are in ISO 8859-1 format [43]. Wide characters (of type `MPI_WCHAR`) are in Unicode format [68].

All signed numerals (e.g., `MPI_INT`, `MPI_REAL`) have the sign bit at the most significant bit. `MPI_COMPLEX` and `MPI_DOUBLE_COMPLEX` have the sign bit of the real and imaginary parts at the most significant bit of each part.

According to IEEE specifications [42], the "NaN" (not a number) is system dependent. It should not be interpreted within MPI as anything other than "NaN."

Advice to implementors. The MPI treatment of "NaN" is similar to the approach used in XDR [65]. (*End of advice to implementors.*)

All data is byte aligned, regardless of type. All data items are stored contiguously in the file (if the file view is contiguous).

1 *Advice to implementors.* All bytes of LOGICAL and bool must be checked to determine
2 the value. (*End of advice to implementors.*)

3
4 *Advice to users.* The type MPI_PACKED is treated as bytes and is not converted.
5 The user should be aware that MPI_PACK has the option of placing a header in the
6 beginning of the pack buffer. (*End of advice to users.*)

7
8 The sizes of the predefined datatypes returned from MPI_TYPE_CREATE_F90_REAL,
9 MPI_TYPE_CREATE_F90_COMPLEX, and MPI_TYPE_CREATE_F90_INTEGER are defined
10 in Section 19.1.9, page 777.

11
12 *Advice to implementors.* When converting a larger size integer to a smaller size
13 integer, only the least significant bytes are moved. Care must be taken to preserve
14 the sign bit value. This allows no conversion errors if the data range is within the
15 range of the smaller size integer. (*End of advice to implementors.*)

16 Table 14.2, 14.3, and 14.4 specify the sizes of predefined, optional, and C++ datatypes
17 in "external32" format, respectively.

19 14.5.3 User-Defined Data Representations

20 There are two situations that cannot be handled by the required representations:

- 21 1. a user wants to write a file in a representation unknown to the implementation, and
- 22 2. a user wants to read a file written in a representation unknown to the implementation.

23
24
25 User-defined data representations allow the user to insert a third party converter into
26 the I/O stream to do the data representation conversion.

27
28
29 MPI_REGISTER_DATAREP(datarep, read_conversion_fn, write_conversion_fn,
30 dtype_file_extent_fn, extra_state)

31	IN	datarep	data representation identifier (string)
32	IN	read_conversion_fn	function invoked to convert from file representation 33 to native representation (function)
34	IN	write_conversion_fn	function invoked to convert from native 35 representation to file representation (function)
36	IN	dtype_file_extent_fn	function invoked to get the extent of a datatype as 37 represented in the file (function)
38	IN	extra_state	extra state

42 C binding

43 int MPI_Register_datarep(const char *datarep,
44 MPI_Datarep_conversion_function *read_conversion_fn,
45 MPI_Datarep_conversion_function *write_conversion_fn,
46 MPI_Datarep_extent_function *dtype_file_extent_fn,
47 void *extra_state)

Table 14.2: "external32" sizes of predefined datatypes

Predefined Type	Length
MPI_PACKED	1
MPI_BYTE	1
MPI_CHAR	1
MPI_UNSIGNED_CHAR	1
MPI_SIGNED_CHAR	1
MPI_WCHAR	2
MPI_SHORT	2
MPI_UNSIGNED_SHORT	2
MPI_INT	4
MPI_LONG	4
MPI_UNSIGNED	4
MPI_UNSIGNED_LONG	4
MPI_LONG_LONG_INT	8
MPI_UNSIGNED_LONG_LONG	8
MPI_FLOAT	4
MPI_DOUBLE	8
MPI_LONG_DOUBLE	16
MPI_C_BOOL	1
MPI_INT8_T	1
MPI_INT16_T	2
MPI_INT32_T	4
MPI_INT64_T	8
MPI_UINT8_T	1
MPI_UINT16_T	2
MPI_UINT32_T	4
MPI_UINT64_T	8
MPI_AINT	8
MPI_COUNT	8
MPI_OFFSET	8
MPI_C_COMPLEX	2*4
MPI_C_FLOAT_COMPLEX	2*4
MPI_C_DOUBLE_COMPLEX	2*8
MPI_C_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
MPI_CXX_BOOL	1
MPI_CXX_FLOAT_COMPLEX	2*4
MPI_CXX_DOUBLE_COMPLEX	2*8
MPI_CXX_LONG_DOUBLE_COMPLEX	2*16

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

Table 14.3: "external32" sizes of optional datatypes

Predefined Type	Length
MPI_INTEGER1	1
MPI_INTEGER2	2
MPI_INTEGER4	4
MPI_INTEGER8	8
MPI_INTEGER16	16
MPI_REAL2	2
MPI_REAL4	4
MPI_REAL8	8
MPI_REAL16	16
MPI_COMPLEX4	2*2
MPI_COMPLEX8	2*4
MPI_COMPLEX16	2*8
MPI_COMPLEX32	2*16

Table 14.4: "external32" sizes of C++ datatypes

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

```

int MPI_Register_datarep_c(const char *datarep,
    MPI_Datarep_conversion_function_c *read_conversion_fn,
    MPI_Datarep_conversion_function_c *write_conversion_fn,
    MPI_Datarep_extent_function *dtype_file_extent_fn,
    void *extra_state)

```

Fortran 2008 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,
        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

```

```

MPI_Register_datarep_c(datarep, read_conversion_fn, write_conversion_fn,
    dtype_file_extent_fn, extra_state, ierror) !(_c)
    CHARACTER(LEN=*), INTENT(IN) :: datarep
    PROCEDURE(MPI_Datarep_conversion_function_c) :: read_conversion_fn,
        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

```

Fortran 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 14.7). The length of a data representation string is limited to the value of `MPI_MAX_DATAREP_STRING`. `MPI_MAX_DATAREP_STRING` must have a value of at least 64. No routines are provided to delete data representations and free the associated resources; it is not expected that an application will generate them in significant numbers.

Extent Callback

```

typedef int MPI_Datarep_extent_function(MPI_Datatype datatype,
    MPI_Aint *extent, void *extra_state);

```

```

1 ABSTRACT INTERFACE
2   SUBROUTINE MPI_Datarep_extent_function(datatype, extent, extra_state, ierror)
3     TYPE(MPI_Datatype) :: datatype
4     INTEGER(KIND=MPI_ADDRESS_KIND) :: extent, extra_state
5     INTEGER :: ierror

```

```

6 SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)
7   INTEGER DATATYPE, IERROR
8   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.

Rationale. This callback does not have a large count variant because it is anticipated that large counts will not be required to represent the extent output value. (*End of rationale.*)

`MPI_Datarep_conversion_function` also supports large count types in separate additional MPI procedures in C (suffixed with the “_c”) and multiple abstract interfaces in Fortran when using `USE mpi_f08`.

If the extent cannot be represented in `extent`, the callback function shall set `extent` to `MPI_UNDEFINED`. The MPI implementation will then raise an error of class `MPI_ERR_VALUE_TOO_LARGE`.

Datarep Conversion Functions

```

27 typedef int MPI_Datarep_conversion_function(void *userbuf,
28     MPI_Datatype datatype, int count, void *filebuf,
29     MPI_Offset position, void *extra_state);
30
31 typedef int MPI_Datarep_conversion_function_c(void *userbuf,
32     MPI_Datatype datatype, MPI_Count count, void *filebuf,
33     MPI_Offset position, void *extra_state);

```

```

34 ABSTRACT INTERFACE
35   SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count, filebuf,
36     position, extra_state, ierror)
37     USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
38     TYPE(C_PTR), VALUE :: userbuf, filebuf
39     TYPE(MPI_Datatype) :: datatype
40     INTEGER :: count, ierror
41     INTEGER(KIND=MPI_OFFSET_KIND) :: position
42     INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state

```

```

43 ABSTRACT INTERFACE
44   SUBROUTINE MPI_Datarep_conversion_function_c(userbuf, datatype, count,
45     filebuf, position, extra_state, ierror) !(_c)
46     USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
47     TYPE(C_PTR), VALUE :: userbuf, filebuf

```

```

TYPE(MPI_Datatype) :: datatype
INTEGER(KIND=MPI_COUNT_KIND) :: count
INTEGER(KIND=MPI_OFFSET_KIND) :: position
INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
INTEGER :: ierror

SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF,
    POSITION, EXTRA_STATE, IERROR)
<TYPE> USERBUF(*), FILEBUF(*)
INTEGER DATATYPE, COUNT, 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.

1 *Rationale.* Passing the conversion function a position and one datatype for the
2 transfer allows the conversion function to decode the datatype only once and cache an
3 internal representation of it on the datatype. Then on subsequent calls, the conversion
4 function can use the position to quickly find its place in the datatype and continue
5 storing converted data where it left off at the end of the previous call. (*End of*
6 *rationale.*)

7
8 *Advice to users.* Although the conversion function may usefully cache an internal
9 representation on the datatype, it should not cache any state information specific to
10 an ongoing conversion operation, since it is possible for the same datatype to be used
11 concurrently in multiple conversion operations. (*End of advice to users.*)

12
13 The function `write_conversion_fn` must convert from native representation to file data
14 representation. Before calling this routine, MPI allocates `filebuf` of a size large enough to
15 hold `count` contiguous data items. The type of each data item matches the corresponding
16 entry for the predefined datatype in the type signature of `datatype`. The function must copy
17 `count` data items from `userbuf` in the distribution described by `datatype`, to a contiguous
18 distribution in `filebuf`, converting each data item from native representation to file repre-
19 sentation. If the size of `datatype` is less than the size of `count` data items, the conversion
20 function must treat `datatype` as being contiguously tiled over the `userbuf`.

21 The function must begin copying at the location in `userbuf` specified by `position` into
22 the (tiled) `datatype`. `datatype` will be equivalent to the datatype that the user passed to the
23 write function. The function is passed, in `extra_state`, the argument that was passed to the
24 `MPI_REGISTER_DATAREP` call.

25 The predefined constant `MPI_CONVERSION_FN_NULL` may be used as either
26 `write_conversion_fn` or `read_conversion_fn` in bindings of `MPI_REGISTER_DATAREP` with-
27 out large counts in these conversion callbacks, whereas the constant
28 `MPI_CONVERSION_FN_NULL_C` can be used in the large count version (i.e.,
29 `MPI_Register_datarep_c`). In either of these cases, MPI will not attempt to invoke
30 `write_conversion_fn` or `read_conversion_fn`, respectively, but will perform the requested data
31 access using the native data representation.

32 An MPI implementation must ensure that all data accessed is converted, either by
33 using a `filebuf` large enough to hold all the requested data items or else by making repeated
34 calls to the conversion function with the same `datatype` argument and appropriate values
35 for `position`.

36 An implementation will only invoke the callback routines in this section
37 (`read_conversion_fn`, `write_conversion_fn`, and `dtype_file_extent_fn`) when one of the read or
38 write routines in Section 14.4, or `MPI_FILE_GET_TYPE_EXTENT` is called by the user.
39 `dtype_file_extent_fn` will only be passed predefined datatypes employed by the user. The
40 conversion functions will only be passed datatypes equivalent to those that the user has
41 passed to one of the routines noted above.

42 The conversion functions must be reentrant. User defined data representations are
43 restricted to use byte alignment for all types. Furthermore, it is erroneous for the conversion
44 functions to call any collective routines or to free `datatype`.

45 The conversion functions should return an error code. If the returned error code has
46 a value other than `MPI_SUCCESS`, the implementation will raise an error in the class
47 `MPI_ERR_CONVERSION`.

48

14.5.4 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 14.5.2, that are supported by all implementations participating in the I/O. The predefined type used to write a data item must also be used to read a data item.
- In the case of Fortran 90 programs, the programs participating in the data accesses obtain compatible datatypes using MPI routines that specify precision and/or range (Section 19.1.9).
- For any given data item, the programs participating in the data accesses use compatible predefined types to write and read the data item.

User-defined data representations may be used to provide an implementation compatibility with another implementation's "native" or "internal" representation.

Advice to users. Section 19.1.9 defines routines that support the use of matching datatypes in heterogeneous environments and contains examples illustrating their use. (*End of advice to users.*)

14.6 Consistency and Semantics

14.6.1 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

1 from a single collective open (e.g., $fh_{1a} \in FH_1$ and $fh_{1b} \in FH_1$), and two file handles from
 2 different collective opens (e.g., $fh_1 \in FH_1$ and $fh_2 \in FH_2$).

3 For the purpose of consistency semantics, a matched pair (Section 14.4.5) of split col-
 4 lective data access operations (e.g., `MPI_FILE_READ_ALL_BEGIN` and
 5 `MPI_FILE_READ_ALL_END`) compose a single data access operation. Similarly, a non-
 6 blocking data access routine (e.g., `MPI_FILE_IREAD`) and the routine that completes the
 7 request (e.g., `MPI_WAIT`) also compose a single data access operation. For all cases below,
 8 these data access operations are subject to the same constraints as blocking data access
 9 operations.

11 *Advice to users.* For an `MPI_FILE_IREAD` and `MPI_WAIT` pair, the operation begins
 12 when `MPI_FILE_IREAD` is called and ends when `MPI_WAIT` returns. (*End of advice*
 13 *to users.*)

15 Assume that A_1 and A_2 are two data access operations. Let D_1 (D_2) be the set of
 16 absolute byte displacements of every byte accessed in A_1 (A_2). The two data accesses
 17 **overlap** if $D_1 \cap D_2 \neq \emptyset$. The two data accesses **conflict** if they overlap and at least one is
 18 a write access.

19 Let SEQ_{fh} be a sequence of file operations on a single file handle, bracketed by
 20 `MPI_FILE_SYNCs` on that file handle. (Both opening and closing a file implicitly perform
 21 an `MPI_FILE_SYNC`.) SEQ_{fh} is a “write sequence” if any of the data access operations in
 22 the sequence are writes or if any of the file manipulation operations in the sequence change
 23 the state of the file (e.g., `MPI_FILE_SET_SIZE` or `MPI_FILE_PREALLOCATE`). Given two
 24 sequences, SEQ_1 and SEQ_2 , we say they are not **concurrent** if one sequence is guaranteed
 25 to completely precede the other (temporally).

26 The requirements for guaranteeing sequential consistency among all accesses to a par-
 27 ticular file are divided into the three cases given below. If any of these requirements are
 28 not met, then the value of all data in that file is implementation dependent.

30 **Case 1:** $fh_1 \in FH_1$. All operations on fh_1 are sequentially consistent if atomic mode is
 31 set. If nonatomic mode is set, then all operations on fh_1 are sequentially consistent if they
 32 are either nonconcurrent, nonconflicting, or both.

34 **Case 2:** $fh_{1a} \in FH_1$ and $fh_{1b} \in FH_1$. Assume A_1 is a data access operation using fh_{1a} ,
 35 and A_2 is a data access operation using fh_{1b} . If for any access A_1 , there is no access A_2
 36 that conflicts with A_1 , then MPI guarantees sequential consistency.

37 However, unlike POSIX semantics, the default MPI semantics for conflicting accesses
 38 do not guarantee sequential consistency. If A_1 and A_2 conflict, sequential consistency can
 39 be guaranteed by either enabling atomic mode via the `MPI_FILE_SET_ATOMICALITY` routine,
 40 or meeting the condition described in Case 3 below.

42 **Case 3:** $fh_1 \in FH_1$ and $fh_2 \in FH_2$. Consider access to a single file using file handles from
 43 distinct collective opens. In order to guarantee sequential consistency, `MPI_FILE_SYNC`
 44 must be used (both opening and closing a file implicitly perform an `MPI_FILE_SYNC`).

45 Sequential consistency is guaranteed among accesses to a single file if for any write
 46 sequence SEQ_1 to the file, there is no sequence SEQ_2 to the file that is *concurrent* with
 47 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 14.6.11 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)
```

Fortran 2008 binding

```
MPI_File_set_atomicity(fh, flag, ierror)
  TYPE(MPI_File), INTENT(IN) :: fh
  LOGICAL, INTENT(IN) :: flag
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran 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)
```

Fortran 2008 binding

```

MPI_File_get_atomicity(fh, flag, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran 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)

```

Fortran 2008 binding

```

MPI_File_sync(fh, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran 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.

14.6.2 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.

14.6.3 Progress

The *progress* rules of MPI are both a promise to users and a set of constraints on implementors. In cases where the progress rules restrict possible implementation choices more than the interface specification alone, the progress rules take precedence.

All blocking routines must complete in finite time unless an exceptional condition (such as resource exhaustion) causes an error.

Nonblocking data access routines inherit the following progress rule from nonblocking point-to-point communication: a nonblocking write is equivalent to a nonblocking send for which a receive is eventually posted, and a nonblocking read is equivalent to a nonblocking receive for which a send is eventually posted.

Finally, an implementation is free to delay progress of collective routines until all processes in the group associated with the collective call have invoked the routine. Once all processes in the group have invoked the routine, the progress rule of the equivalent noncollective routine must be followed.

14.6.4 Collective File Operations

Collective file operations are subject to the same restrictions as collective communication operations. For a complete discussion, please refer to the semantics set forth in Section 6.14.

Collective file operations are collective over a duplicate of the communicator used to open the file—this duplicate communicator is implicitly specified via the file handle argument. Different processes can pass different values for other arguments of a collective routine unless specified otherwise.

14.6.5 Nonblocking Collective File Operations

Nonblocking collective file operations are defined only for data access routines with explicit offsets and individual file pointers but not with shared file pointers.

Nonblocking collective file operations are subject to the same restrictions as blocking collective I/O operations. All processes belonging to the group of the communicator that was used to open the file must call collective I/O operations (blocking and nonblocking) in the same order. This is consistent with the ordering rules for collective operations in threaded environments. For a complete discussion, please refer to the semantics set forth in Section 6.14.

Nonblocking collective I/O operations do not match with blocking collective I/O operations. Multiple nonblocking collective I/O operations can be outstanding on a single file

1 handle. High quality MPI implementations should be able to support a large number of
2 pending nonblocking I/O operations.

3 All nonblocking collective I/O calls are local and return immediately, irrespective of the
4 status of other processes. The call initiates the operation that may progress independently
5 of any communication, computation, or I/O. The call returns a request handle, which must
6 be passed to a completion call. Input buffers should not be modified and output buffers
7 should not be accessed before the completion call returns. The same *progress* rules described
8 for nonblocking collective operations apply for nonblocking collective I/O operations. For
9 a complete discussion, please refer to the semantics set forth in Section 6.12.

11 14.6.6 Type Matching

12 The type matching rules for I/O mimic the type matching rules for communication with one
13 exception: if `etype` is `MPI_BYTE`, then this matches any `datatype` in a data access operation.
14 In general, the `etype` of data items written must match the `etype` used to read the items,
15 and for each data access operation, the current `etype` must also match the type declaration
16 of the data access buffer.

17
18 *Advice to users.* In most cases, use of `MPI_BYTE` as a wild card will defeat the
19 file interoperability features of MPI. File interoperability can only perform automatic
20 conversion between heterogeneous data representations when the exact datatypes ac-
21 cessed are explicitly specified. (*End of advice to users.*)

23 14.6.7 Miscellaneous Clarifications

24
25 Once an I/O routine completes, it is safe to free any opaque objects passed as arguments
26 to that routine. For example, the `comm` and `info` used in an `MPI_FILE_OPEN`, or the `etype`
27 and `filetype` used in an `MPI_FILE_SET_VIEW`, can be freed without affecting access to the
28 file. Note that for nonblocking routines and split collective operations, the operation must
29 be completed before it is safe to reuse data buffers passed as arguments.

30 As in communication, datatypes must be committed before they can be used in file
31 manipulation or data access operations. For example, the `etype` and `filetype` must be com-
32 mitted before calling `MPI_FILE_SET_VIEW`, and the `datatype` must be committed before
33 calling `MPI_FILE_READ` or `MPI_FILE_WRITE`.

35 14.6.8 MPI_Offset Type

36
37 `MPI_Offset` is an integer type of size sufficient to represent the size (in bytes) of the largest
38 file supported by MPI. Displacements and offsets are always specified as values of type
39 `MPI_Offset`.

40 In Fortran, the corresponding integer is an integer with kind parameter
41 `MPI_OFFSET_KIND`, which is defined in the `mpi_f08` module, the `mpi` module and the `mpif.h`
42 include file.

43 In Fortran 77 environments that do not support `KIND` parameters, `MPI_Offset` arguments
44 should be declared as an `INTEGER` of suitable size. The language interoperability implications
45 for `MPI_Offset` are similar to those for addresses (see Section 19.3).

14.6.9 Logical vs. Physical File Layout

MPI specifies how the data should be laid out in a virtual file structure (the view), not 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 14.2.8).

14.6.10 File Size

The size of a file may be increased by writing to the file after the current end of file. The size may also be changed by calling MPI *size changing* routines, such as `MPI_FILE_SET_SIZE`. A call to a size changing routine does not necessarily change the file size. For example, calling `MPI_FILE_PREALLOCATE` with a size less than the current size does not change the size.

Consider a set of bytes that has been written to a file since the most recent call to a size changing routine, or since `MPI_FILE_OPEN` if no such routine has been called. Let the *high byte* be the byte in that set with the largest displacement. The file size is the larger of

- One plus the displacement of the high byte.
- The size immediately after the size changing routine, or `MPI_FILE_OPEN`, returned.

When applying consistency semantics, calls to `MPI_FILE_SET_SIZE` and `MPI_FILE_PREALLOCATE` are considered writes to the file (which conflict with operations that access bytes at displacements between the old and new file sizes), and `MPI_FILE_GET_SIZE` is considered a read of the file (which overlaps with all accesses to the file).

Advice to users. Any sequence of operations containing the collective routines `MPI_FILE_SET_SIZE` and `MPI_FILE_PREALLOCATE` is a write sequence. As such, sequential consistency in nonatomic mode is not guaranteed unless the conditions in Section 14.6.1 are satisfied. (*End of advice to users.*)

File pointer update semantics (i.e., file pointers are updated by the amount accessed) are only guaranteed if file size changes are sequentially consistent.

Advice to users. Consider the following example. Given two operations made by separate processes to a file containing 100 bytes: an `MPI_FILE_READ` of 10 bytes and an `MPI_FILE_SET_SIZE` to 0 bytes. If the user does not enforce sequential consistency between these two operations, the file pointer may be updated by the amount requested (10 bytes) even if the amount accessed is zero bytes. (*End of advice to users.*)

14.6.11 Examples

The examples in this section illustrate the application of the MPI consistency and semantics guarantees. These address

- 1 • conflicting accesses on file handles obtained from a single collective open, and
- 2
- 3 • all accesses on file handles obtained from two separate collective opens.

4 The simplest way to achieve consistency for conflicting accesses is to obtain sequential
 5 consistency by setting atomic mode. For the code below, process 1 will read either 0 or 10
 6 integers. If the latter, every element of `b` will be 5. If nonatomic mode is set, the results of
 7 the read are undefined.

```

8  /* Process 0 */
9
10 int i, a[10];
11 int TRUE = 1;
12
13 for (i=0; i<10; i++)
14     a[i] = 5;
15
16 MPI_File_open(MPI_COMM_WORLD, "workfile",
17               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0);
18 MPI_File_set_view(fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
19 MPI_File_set_atomicity(fh0, TRUE);
20 MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status);
21 /* MPI_Barrier(MPI_COMM_WORLD); */
22
23 /* Process 1 */
24
25 int b[10];
26 int TRUE = 1;
27 MPI_File_open(MPI_COMM_WORLD, "workfile",
28               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1);
29 MPI_File_set_view(fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
30 MPI_File_set_atomicity(fh1, TRUE);
31 /* MPI_Barrier(MPI_COMM_WORLD); */
32 MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status);
  
```

33 A user may guarantee that the write on process 0 precedes the read on process 1 by imposing
 34 temporal order with, for example, calls to `MPI_BARRIER`.

35 *Advice to users.* Routines other than `MPI_BARRIER` may be used to impose temporal
 36 order. In the example above, process 0 could use `MPI_SEND` to send a 0 byte message,
 37 received by process 1 using `MPI_RECV`. (*End of advice to users.*)

38 Alternatively, a user can impose consistency with nonatomic mode set:

```

39 /* Process 0 */
40
41 int i, a[10];
42 for (i=0; i<10; i++)
43     a[i] = 5;
44
45 MPI_File_open(MPI_COMM_WORLD, "workfile",
46               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0);
47 MPI_File_set_view(fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
48 MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status);
49 MPI_File_sync(fh0);
50 MPI_Barrier(MPI_COMM_WORLD);
51 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 that 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

1 “sync-barrier-sync” construct above can be replaced by a single “sync.” The results of
 2 using such code with an implementation for which `MPI_FILE_SYNC` is not temporally
 3 synchronizing is undefined. (*End of advice to users.*)
 4

5 Asynchronous I/O

6 The behavior of asynchronous I/O operations is determined by applying the rules specified
 7 above for synchronous I/O operations.
 8

9 The following examples all access a preexisting file “myfile.” Word 10 in myfile initially
 10 contains the integer 2. Each example writes and reads word 10.

11 First consider the following code fragment:

```
12 int a = 4, b, TRUE=1;
13 MPI_File_open(MPI_COMM_WORLD, "myfile",
14               MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
15 MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
16 /* MPI_File_set_atomicity(fh, TRUE); Use this to set atomic mode. */
17 MPI_File_iread_at(fh, 10, &a, 1, MPI_INT, &reqs[0]);
18 MPI_File_iwrite_at(fh, 10, &b, 1, MPI_INT, &reqs[1]);
19 MPI_Waitall(2, reqs, statuses);
```

20 For asynchronous data access operations, MPI specifies that the access occurs at any time
 21 between the call to the asynchronous data access routine and the return from the corre-
 22 sponding request complete routine. Thus, executing either the read before the write, or the
 23 write before the read is consistent with program order. If atomic mode is set, then MPI
 24 guarantees sequential consistency, and the program will read either 2 or 4 into b. If atomic
 25 mode is not set, then sequential consistency is not guaranteed and the program may read
 26 something other than 2 or 4 due to the conflicting data access.

27 Similarly, the following code fragment does not order file accesses:

```
28 int a = 4, b;
29 MPI_File_open(MPI_COMM_WORLD, "myfile",
30               MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
31 MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
32 /* MPI_File_set_atomicity(fh, TRUE); Use this to set atomic mode. */
33 MPI_File_iread_at(fh, 10, &a, 1, MPI_INT, &reqs[0]);
34 MPI_File_iwrite_at(fh, 10, &b, 1, MPI_INT, &reqs[1]);
35 MPI_Wait(&reqs[0], &status);
36 MPI_Wait(&reqs[1], &status);
```

37 If atomic mode is set, either 2 or 4 will be read into b. Again, MPI does not guarantee
 38 sequential consistency in nonatomic mode.

39 On the other hand, the following code fragment:

```
40 int a = 4, b;
41 MPI_File_open(MPI_COMM_WORLD, "myfile",
42               MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
43 MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
44 MPI_File_iread_at(fh, 10, &a, 1, MPI_INT, &reqs[0]);
45 MPI_Wait(&reqs[0], &status);
46 MPI_File_iwrite_at(fh, 10, &b, 1, MPI_INT, &reqs[1]);
47 MPI_Wait(&reqs[1], &status);
```

48 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_iread_all(fh,...);
MPI_File_iwrite_all(fh,...);
MPI_Waitall(...);

```

In addition, as mentioned in Section 14.6.5, 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 14.4.5).

14.7 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 9.3.

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 file 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` when using the World Model. In I/O, there is no analogous “root” file handle from which default properties can be inherited. Rather than invent a new global file handle, the default file error handler is manipulated as if it were attached to `MPI_FILE_NULL`. (*End of rationale.*)

14.8 I/O Error Classes

The implementation dependent error codes returned by the I/O routines can be converted into the error classes defined in Table 14.5.

In addition, calls to routines in this chapter may raise errors in other MPI classes, such as `MPI_ERR_TYPE`.

14.9 Examples

14.9.1 Double Buffering with Split Collective I/O

This example shows how to overlap computation and output. The computation is performed by the function `compute_buffer()`.

```

/*=====
37  *
38  * Function:          double_buffer
39  *
40  * Synopsis:
41  *   void double_buffer(
42  *       MPI_File fh,                ** IN
43  *       MPI_Datatype buftype,      ** IN
44  *       int bufcount                ** IN
45  *   )
46  *
47  * Description:
48  *   Performs the steps to overlap computation with a collective write
49  *   by using a double-buffering technique.
50  *
51  *=====

```

Table 14.5: I/O error classes		1
MPI_ERR_FILE	Invalid file handle	2
MPI_ERR_NOT_SAME	Collective argument not identical on all processes, or collective routines called in a different order by different processes	3
MPI_ERR_AMODE	Error related to the <code>amode</code> passed to <code>MPI_FILE_OPEN</code>	4
MPI_ERR_UNSUPPORTED_DATAREP	Unsupported <code>datarep</code> passed to <code>MPI_FILE_SET_VIEW</code>	5
MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on a file that supports sequential access only	6
MPI_ERR_NO_SUCH_FILE	File does not exist	7
MPI_ERR_FILE_EXISTS	File exists	8
MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)	9
MPI_ERR_ACCESS	Permission denied	10
MPI_ERR_NO_SPACE	Not enough space	11
MPI_ERR_QUOTA	Quota exceeded	12
MPI_ERR_READ_ONLY	Read-only file or file system	13
MPI_ERR_FILE_IN_USE	File operation could not be completed, as the file is currently open by some process	14
MPI_ERR_DUP_DATAREP	Conversion functions could not be registered because a data representation identifier that was already defined was passed to <code>MPI_REGISTER_DATAREP</code>	15
MPI_ERR_CONVERSION	An error occurred in a user supplied data conversion function.	16
MPI_ERR_IO	Other I/O error	17

```

1  * Parameters:
2  *     fh                previously opened MPI file handle
3  *     buftype           MPI datatype for memory layout
4  *                       (Assumes a compatible view has been set on fh)
5  *     bufcount          # buftype elements to transfer
6  *-----*/
7  /* this macro switches which buffer "x" is pointing to */
8  #define TOGGLE_PTR(x) (((x)==(buffer1)) ? (x=buffer2) : (x=buffer1))
9
10 void double_buffer(MPI_File fh, MPI_Datatype buftype, int bufcount)
11 {
12     MPI_Status status;          /* status for MPI calls */
13     float *buffer1, *buffer2;  /* buffers to hold results */
14     float *compute_buf_ptr;    /* destination buffer */
15                                 /* for computing */
16     float *write_buf_ptr;      /* source for writing */
17     int done;                  /* determines when to quit */
18
19     /* buffer initialization */
20     buffer1 = (float *)
21         malloc(bufcount*sizeof(float));
22     buffer2 = (float *)
23         malloc(bufcount*sizeof(float));
24     compute_buf_ptr = buffer1;  /* initially point to buffer1 */
25     write_buf_ptr = buffer1;   /* initially point to buffer1 */
26
27     /* DOUBLE-BUFFER prolog:
28     *   compute buffer1; then initiate writing buffer1 to disk
29     */
30     compute_buffer(compute_buf_ptr, bufcount, &done);
31     MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
32
33     /* DOUBLE-BUFFER steady state:
34     *   Overlap writing old results from buffer pointed to by write_buf_ptr
35     *   with computing new results into buffer pointed to by compute_buf_ptr.
36     *
37     *   There is always one write-buffer and one compute-buffer in use
38     *   during steady state.
39     */
40     while (!done) {
41         TOGGLE_PTR(compute_buf_ptr);
42         compute_buffer(compute_buf_ptr, bufcount, &done);
43         MPI_File_write_all_end(fh, write_buf_ptr, &status);
44         TOGGLE_PTR(write_buf_ptr);
45         MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
46     }
47
48     /* DOUBLE-BUFFER epilog:
49     *   wait for final write to complete.
50     */
51     MPI_File_write_all_end(fh, write_buf_ptr, &status);
52
53     /* buffer cleanup */
54     free(buffer1);
55     free(buffer2);

```

}

14.9.2 Subarray Filetype Constructor

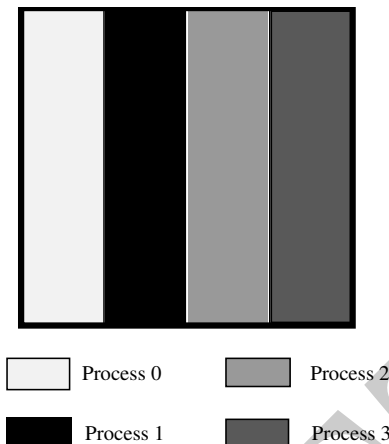


Figure 14.4: Example array file layout

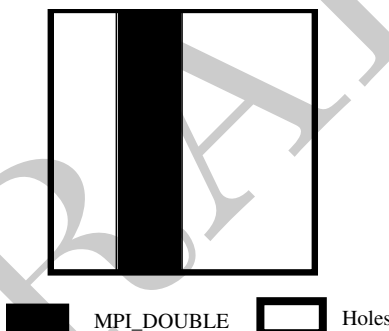


Figure 14.5: Example local array filetype for process 1

Assume we are writing out a 100×100 2D array of double precision floating point numbers that is distributed among 4 processes such that each process has a block of 25 columns (e.g., process 0 has columns 0–24, process 1 has columns 25–49, etc.; see Figure 14.4). To create the filetypes for each process one could use the following C program (see Section 5.1.3):

```

double subarray[100][25];
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,

```

```
1 MPI_DOUBLE, &filetype);
```

```
2
```

Or, equivalently in Fortran:

```
3
```

```
4 double precision subarray(100,25)
```

```
5 integer filetype, rank, ierror
```

```
6 integer sizes(2), subsizes(2), starts(2)
```

```
7
```

```
8 call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
```

```
9 sizes(1) = 100
```

```
10 sizes(2) = 100
```

```
11 subsizes(1) = 100
```

```
12 subsizes(2) = 25
```

```
13 starts(1) = 0
```

```
14 starts(2) = rank*subsizes(2)
```

```
15
```

```
16 call MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts, &
```

```
17 MPI_ORDER_FORTRAN, MPI_DOUBLE_PRECISION, &
```

```
18 filetype, ierror)
```

```
19
```

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 14.5 shows the filetype created for process 1.

```
20
```

```
21
```

```
22
```

```
23
```

```
24
```

```
25
```

```
26
```

```
27
```

```
28
```

```
29
```

```
30
```

```
31
```

```
32
```

```
33
```

```
34
```

```
35
```

```
36
```

```
37
```

```
38
```

```
39
```

```
40
```

```
41
```

```
42
```

```
43
```

```
44
```

```
45
```

```
46
```

```
47
```

```
48
```


Chapter 15

Tool Support

15.1 Introduction

This chapter discusses interfaces that allow debuggers, performance analyzers, and other tools to extract information about the behavior of MPI processes. Specifically, this chapter defines both the MPI profiling interface (Section 15.2), which supports the transparent interception and inspection of MPI calls, and the MPI tool information interface (Section 15.3), which supports the inspection and manipulation of MPI control and performance variables, as well as the registration of callbacks for MPI library events. The interfaces described in this chapter are all defined in the context of an MPI process, i.e., are callable from the same code that invokes other MPI functions.

15.2 Profiling Interface

15.2.1 Requirements

To meet the requirements for the MPI profiling interface, an implementation of the MPI functions *must*

1. provide a mechanism through which all of the MPI defined functions, except those allowed as macros (See Section 2.6.4), may be accessed with a name shift. This requires, in C and Fortran, an alternate entry point name, with the prefix `PMPI_` for each MPI function in each provided language binding and language support method. For routines implemented as macros, it is still required that the `PMPI_` version be supplied and work as expected, but it is not possible to replace at link time the `MPI_` version with a user-defined version.

For Fortran, the different support methods cause several specific procedure names. Therefore, several profiling routines (with these specific procedure names) are needed for each Fortran MPI routine, as described in Section 19.1.5.

2. ensure that those MPI functions that are not replaced may still be linked into an executable image without causing name clashes.
3. document the implementation of different language bindings of the MPI interface if they are layered on top of each other, so that the profiler developer knows whether to implement the profile interface for each binding, or to economize by implementing it only for the lowest level routines.
4. where the implementation of different language bindings is done through a layered approach (e.g., the Fortran binding is a set of “wrapper” functions that call the C

1 implementation), ensure that these wrapper functions are separable from the rest of
2 the library.

3 This separability is necessary to allow a separate profiling library to be correctly
4 implemented, since (at least with Unix linker semantics) the profiling library must
5 contain these wrapper functions if it is to perform as expected. This requirement
6 allows the person who builds the profiling library to extract these functions from the
7 original MPI library and add them into the profiling library without bringing along
8 any other unnecessary code.

- 9
10 5. provide a no-op routine `MPI_PCONTROL` in the MPI library.

11 15.2.2 Discussion

12 The objective of the MPI profiling interface is to ensure that it is relatively easy for authors
13 of profiling (and other similar) tools to interface their codes to MPI implementations on
14 different machines.

15 Since MPI is a machine independent standard with many different implementations,
16 it is unreasonable to expect that the authors of profiling tools for MPI will have access to
17 the source code that implements MPI on any particular machine. It is therefore necessary
18 to provide a mechanism by which the implementors of such tools can collect whatever
19 performance information they wish *without* access to the underlying implementation.

20 We believe that having such an interface is important if MPI is to be attractive to end
21 users, since the availability of many different tools will be a significant factor in attracting
22 users to the MPI standard.

23 The profiling interface is just that, an interface. It says *nothing* about the way in which
24 it is used. There is therefore no attempt to lay down what information is collected through
25 the interface, or how the collected information is saved, filtered, or displayed.

26 While the initial impetus for the development of this interface arose from the desire to
27 permit the implementation of profiling tools, it is clear that an interface like that specified
28 may also prove useful for other purposes, such as “internetworking” multiple MPI imple-
29 mentations. Since all that is defined is an interface, there is no objection to it being used
30 wherever it is useful.

31 As the issues being addressed here are intimately tied up with the way in which ex-
32 ecutable images are built, which may differ greatly on different machines, the examples
33 given below should be treated solely as one way of implementing the objective of the MPI
34 profiling interface. The actual requirements made of an implementation are those detailed
35 in the Requirements section above, the whole of the rest of this section is only present as
36 justification and discussion of the logic for those requirements.

37 The examples below show one way in which an implementation could be constructed to
38 meet the requirements on a Unix system (there are doubtless others that would be equally
39 valid).

40 15.2.3 Logic of the Design

41 Provided that an MPI implementation meets the requirements above, it is possible for
42 the implementor of the profiling system to intercept the MPI calls that are made by the
43 user program. The profiling system implementor can then collect any required information
44 before calling the underlying MPI implementation (through its name shifted entry points)
45 to achieve the desired effects.

15.2.4 Miscellaneous Control of Profiling

There is a clear requirement for the user code to be able to control the profiler dynamically at run time. This capability is normally used for (at least) the purposes of

- Enabling and disabling profiling depending on the state of the calculation.
- Flushing trace buffers at noncritical points in the calculation.
- Adding user events to a trace file.

These requirements are met by use of `MPI_PCONTROL`.

`MPI_PCONTROL(level, ...)`

IN level Profiling level (integer)

C binding

`int MPI_Pcontrol(const int level, ...)`

Fortran 2008 binding

`MPI_Pcontrol(level)`
`INTEGER, INTENT(IN) :: level`

Fortran binding

`MPI_PCONTROL(LEVEL)`
`INTEGER LEVEL`

MPI libraries themselves make no use of this routine, and simply return immediately to the user code. However the presence of calls to this routine allows a profiling package to be explicitly called by the user.

Since MPI has no control of the implementation of the profiling code, we are unable to specify precisely the semantics that will be provided by calls to `MPI_PCONTROL`. This vagueness extends to the number of arguments to the function, and their datatypes.

However to provide some level of portability of user codes to different profiling libraries, we request the following meanings for certain values of `level`.

- `level=0` Profiling is disabled.
- `level=1` Profiling is enabled at a normal default level of detail.
- `level=2` Profile buffers are flushed, which may be a no-op in some profilers.
- All other values of `level` have profile library defined effects and additional arguments.

We also request that the default state after MPI has been initialized is for profiling to be enabled at the normal default level. (i.e., as if `MPI_PCONTROL` had just been called with the argument 1). This allows users to link with a profiling library and to obtain profile output without having to modify their source code at all.

The provision of `MPI_PCONTROL` as a no-op in the standard MPI library supports the collection of more detailed profiling information with source code that can still link against the standard MPI library.

Example 15.1. A wrapper to accumulate the total amount of data sent by the `MPI_SEND` function, along with the total elapsed time spent in the function.

```

1  static int totalBytes = 0;
2  static double totalTime = 0.0;
3
4  int MPI_Send(const void* buffer, int count, MPI_Datatype datatype,
5              int dest, int tag, MPI_Comm comm)
6  {
7      double tstart = MPI_Wtime();      /* Pass on all arguments */
8      int size;
9      int result = PMPI_Send(buffer, count, datatype, dest, tag, comm);
10
11     totalTime += MPI_Wtime() - tstart; /* Compute time */
12
13     MPI_Type_size(datatype, &size);    /* and size */
14     totalBytes += count*size;
15
16     return result;
17 }
18
19

```

15.2.5 MPI Library Implementation

If the MPI library is implemented in C on a Unix system, then there are various options, including the two presented here, for supporting the name-shift requirement. The choice between these two options depends partly on whether the linker and compiler support weak symbols.

If the compiler and linker support weak external symbols (e.g., Solaris 2.x, other System V.4 machines), then only a single library is required as the following example shows:

Example 15.2. Library implementation using weak symbols.

```

30 #pragma weak MPI_Example = PMPI_Example
31
32 int PMPI_Example(/* appropriate args */)
33 {
34     /* Useful content */
35 }
36

```

The effect of this `#pragma` is to define the external symbol `MPI_Example` as a weak definition. This means that the linker will not complain if there is another definition of the symbol (for instance in the profiling library); however if no other definition exists, then the linker will use the weak definition.

In the absence of weak symbols then one possible solution would be to use the C macro preprocessor as the following example shows:

Example 15.3. Library implementation using C pre-processor macros.

```

45 #ifdef PROFILELIB
46     #ifdef __STDC__
47         #define FUNCTION(name) P##name
48     #else

```

```

1 #     define FUNCTION(name) P/**/name
2 #     endif
3 #else
4 #     define FUNCTION(name) name
5 #endif

```

Each of the user visible functions in the library would then be declared thus

```

6
7 int FUNCTION(MPI_Example)(/* appropriate args */)
8 {
9     /* Useful content */
10 }
11

```

The same source file can then be compiled to produce both versions of the library, depending on the state of the PROFILELIB macro symbol.

It is required that the standard MPI library be built in such a way that the inclusion of MPI functions can be achieved one at a time. This is a somewhat unpleasant requirement, since it may mean that each external function has to be compiled from a separate file. However this is necessary so that the author of the profiling library need only define those MPI functions that need to be intercepted, references to any others being fulfilled by the normal MPI library.

Example 15.4.

The following example shows a potential link step when using the profiling interface.

```

12 % cc ... -lmyprof -lpmpi -lmpi
13

```

Here libmyprof.a contains the profiler functions that intercept some of the MPI functions, libpmpi.a contains the “name shifted” MPI functions, and libmpi.a contains the normal definitions of the MPI functions.

15.2.6 Complications

Multiple Counting

Since parts of the MPI library may themselves be implemented using more basic MPI functions (e.g., a portable implementation of the collective operations implemented using point-to-point communications), there is potential for profiling functions to be called from within an MPI function that was called from a profiling function. This could lead to “double counting” of the time spent in the inner routine. Since this effect could actually be useful under some circumstances (e.g., it might allow one to answer the question “How much time is spent in the point-to-point routines when they are called from collective functions?”), we have decided not to enforce any restrictions on the author of the MPI library that would overcome this. Therefore the author of the profiling library should be aware of this problem, and guard against it. In a single-threaded world this is easily achieved through use of a static variable in the profiling code that remembers if you are already inside a profiling routine. It becomes more complex in a multithreaded environment (as does the meaning of the times recorded).

Linker Oddities

The Unix linker traditionally operates in one pass: the effect of this is that functions from libraries are only included in the image if they are needed at the time the library is scanned. When combined with weak symbols, or multiple definitions of the same function, this can cause odd (and unexpected) effects.

Consider, for instance, an implementation of MPI in which the Fortran binding is achieved by using wrapper functions on top of the C implementation. The author of the profile library then assumes that it is reasonable only to provide profile functions for the C binding, since Fortran will eventually call these, and the cost of the wrappers is assumed to be small. However, if the wrapper functions are not in the profiling library, then none of the profiled entry points will be undefined when the profiling library is called. Therefore none of the profiling code will be included in the image. When the standard MPI library is scanned, the Fortran wrappers will be resolved, and will also pull in the base versions of the MPI functions. The overall effect is that the code will link successfully, but will not be profiled.

To overcome this we must ensure that the Fortran wrapper functions are included in the profiling version of the library. We ensure that this is possible by requiring that these be separable from the rest of the base MPI library. This allows them to be copied out of the base library and into the profiling one using a tool such as `ar`.

Fortran Support Methods

The different Fortran support methods and possible options for the support of subarrays (depending on whether the compiler can support `TYPE(*)`, `DIMENSION(..)` choice buffers) imply different specific procedure names for the same Fortran MPI routine. The rules and implications for the profiling interface are described in Section 19.1.5.

15.2.7 Multiple Levels of Interception

The scheme given here does not directly support the nesting of profiling functions, since it provides only a single alternative name for each MPI function. Consideration was given to an implementation that would allow multiple levels of call interception, however we were unable to construct an implementation of this that did not have the following disadvantages

- assuming a particular implementation language, and
- imposing a run time cost even when no profiling was taking place.

Since one of the objectives of MPI is to permit efficient, low latency implementations, and it is not the business of a standard to require a particular implementation language, we decided to accept the scheme outlined above.

Note, however, that it is possible to use the scheme above to implement a multi-level system, since the function called by the user may call many different profiling functions before calling the underlying MPI function. This capability has been demonstrated in the P^NMPI tool infrastructure [58].

15.3 The MPI Tool Information Interface

MPI implementations often use internal variables to control their behavior and performance and rely on internal events for their implementation. Understanding and manipulating these variables and tracking these events can provide a more efficient execution environment or improve performance for many applications. This section describes the MPI tool information interface, which provides a mechanism for MPI implementors to expose variables, each of which represents a particular property, setting, or performance measurement from within the MPI implementation, as well as expose events that can be tracked by tools. The interface is split into three parts: the first part provides information about, and supports the setting of, control variables through which the MPI implementation tunes its configuration. The second part provides access to performance variables that can provide insight into internal performance information of the MPI implementation. The third part enables tools to query available events within an MPI implementation and register callbacks for them.

To avoid restrictions on the MPI implementation, the MPI tool information interface allows the implementation to specify which control variables, performance variables, and events exist. Additionally, the user of the MPI tool information interface can obtain meta-data about each available variable or event, such as its datatype, and a textual description. The MPI tool information interface provides the necessary routines to find all variables and events that exist in a particular MPI implementation; to query their properties; to retrieve descriptions about their meaning; to access and, if appropriate, to alter their values; and (in case of events) set callbacks triggered by them.

Variables, events, and categories across connected MPI processes with equivalent names are required to have the same meaning (see the definition of “equivalent” as related to strings in Section 15.3.3). Furthermore, enumerations with equivalent names across connected MPI processes are required to have the same meaning, but are allowed to comprise different enumeration items. Enumeration items that have equivalent names across connected MPI processes in enumerations with the same meaning must also have the same meaning. In order for variables and categories to have the same meaning, routines in the tools information interface that return details for those variables and categories have requirements on what parameters must be identical. These requirements are specified in their respective sections.

Rationale. The intent of requiring the same meaning for entities with equivalent names is to enforce consistency across connected MPI processes. For example, variables describing the number of packets sent on different types of network devices should have different names to reflect their potentially different meanings. (*End of rationale.*)

The MPI tool information interface can be used independently from the MPI communication functionality. In particular, the routines of this interface can be called before MPI is initialized and after MPI is finalized. In order to support this behavior cleanly, the MPI tool information interface uses separate initialization and finalization routines. All identifiers used in the MPI tool information interface have the prefix `MPI_T_`.

On success, all MPI tool information interface routines return `MPI_SUCCESS`, otherwise they return an appropriate and unique return code indicating the reason why the call was not successfully completed. Details on return codes can be found in Section 15.3.10. However, unsuccessful calls to the MPI tool information interface are not fatal and do not impact the execution of subsequent MPI routines.

Table 15.1: MPI tool information interface verbosity levels

MPI_T_VERBOSITY_USER_BASIC	Basic information of interest to users
MPI_T_VERBOSITY_USER_DETAIL	Detailed information of interest to users
MPI_T_VERBOSITY_USER_ALL	All remaining information of interest to users
MPI_T_VERBOSITY_TUNER_BASIC	Basic information required for tuning
MPI_T_VERBOSITY_TUNER_DETAIL	Detailed information required for tuning
MPI_T_VERBOSITY_TUNER_ALL	All remaining information required for tuning
MPI_T_VERBOSITY_MPIDEV_BASIC	Basic information for MPI implementors
MPI_T_VERBOSITY_MPIDEV_DETAIL	Detailed information for MPI implementors
MPI_T_VERBOSITY_MPIDEV_ALL	All remaining information for MPI implementors

Since the MPI tool information interface primarily focuses on tools and support libraries, MPI implementations are only required to provide C bindings for functions and constants introduced in this section. Except where otherwise noted, all conventions and principles governing the C bindings of the MPI API also apply to the MPI tool information interface, which is available by including the `mpi.h` header file. All routines in this interface have local semantics.

Advice to users. The number and type of control variables, performance variables, and events can vary between MPI implementations, platforms and different builds of the same implementation on the same platform as well as between runs. Hence, any application relying on a particular variable will not be portable. Further, there is no guarantee that the number of variables and variable indices are the same across connected MPI processes.

This interface is primarily intended for performance monitoring tools, support tools, and libraries controlling the application's environment. When maximum portability is desired, application programmers should either avoid using the MPI tool information interface or avoid being dependent on the existence of a particular control or performance variable or of a particular event. (*End of advice to users.*)

15.3.1 Verbosity Levels

The MPI tool information interface provides access to internal configuration and performance information through a set of control and performance variables defined by the MPI implementation. Since some implementations may export a large number of variables, variables are classified by a verbosity level that categorizes both their intended audience (end users, performance tuners or MPI implementors) and a relative measure of level of detail (basic, detailed or all). These verbosity levels are described by a single integer. Table 15.1 lists the constants for all possible verbosity levels. The values of the constants are monotonic in the order listed in the table; i.e., `MPI_T_VERBOSITY_USER_BASIC` < `MPI_T_VERBOSITY_USER_DETAIL` < ... < `MPI_T_VERBOSITY_MPIDEV_ALL`.

15.3.2 Binding MPI Tool Information Interface Variables to MPI Objects

Each MPI tool information interface variable provides access to a particular control setting or performance property of the MPI implementation. A variable may refer to a specific

MPI object such as a communicator, datatype, or one-sided communication window, or the variable may refer more generally to the MPI environment of the process. Except for the last case, the variable must be bound to exactly one MPI object before it can be used. Table 15.2 lists all MPI object types to which an MPI tool information interface variable can be bound, together with the matching constant that MPI tool information interface routines return to identify the object type. It is erroneous to bind a control variable, performance variable, or event to a handle that would not be valid to use as an input argument to another MPI call (excluding calls to the MPI Tool Information Interface) at the same point of execution.

Table 15.2: Constants to identify associations of variables

Constant	MPI object
MPI_T_BIND_NO_OBJECT	N/A; applies globally to entire MPI process
MPI_T_BIND_MPI_COMM	MPI communicators
MPI_T_BIND_MPI_DATATYPE	MPI datatypes
MPI_T_BIND_MPI_ERRHANDLER	MPI error handlers
MPI_T_BIND_MPI_FILE	MPI file handles
MPI_T_BIND_MPI_GROUP	MPI groups
MPI_T_BIND_MPI_OP	MPI reduction operators
MPI_T_BIND_MPI_REQUEST	MPI requests
MPI_T_BIND_MPI_WIN	MPI windows for one-sided communication
MPI_T_BIND_MPI_MESSAGE	MPI message object
MPI_T_BIND_MPI_INFO	MPI info object
MPI_T_BIND_MPI_SESSION	MPI session object

Rationale. Some variables have meanings tied to a specific MPI object. Examples include the number of send or receive operations that use a particular datatype, the number of times a particular error handler has been called, or the communication protocol and “eager limit” used for a particular communicator. Creating a new MPI tool information interface variable for each MPI object would cause the number of variables to grow without bound, since they cannot be reused to avoid naming conflicts. By associating MPI tool information interface variables with a specific MPI object, the MPI implementation only must specify and maintain a single variable, which can then be applied to as many MPI objects of the respective type as created during the program’s execution. (*End of rationale.*)

15.3.3 Convention for Returning Strings

Several MPI tool information interface functions return one or more strings. These functions have two arguments for each string to be returned: an OUT parameter that identifies a pointer to the buffer in which the string will be returned, and an INOUT parameter to pass the length of the buffer. The user is responsible for the memory allocation of the buffer and must pass the size of the buffer (n) as the length argument. Let n be the length value specified to the function. On return, the function writes at most $n - 1$ of the string’s characters into the buffer, followed by a null terminator. If the returned string’s length is greater than or equal to n , the string will be truncated to $n - 1$ characters. In this case, the length of the string plus one (for the terminating null character) is returned in the length

argument. If the user passes the null pointer as the buffer argument or passes 0 as the length argument, the function does not return the string and only returns the length of the string plus one in the length argument. If the user passes the null pointer as the length argument, the buffer argument is ignored and nothing is returned.

MPI implementations behave as if they have an internal character array that is copied to the output character array supplied by the user. Such output strings are only defined to be equivalent if their notional source-internal character arrays are identical (up to and including the null terminator), even if the output string is truncated due to a small input length parameter n .

15.3.4 Initialization and Finalization

The MPI tool information interface requires a separate set of initialization and finalization routines.

`MPI_T_INIT_THREAD(required, provided)`

IN	required	desired level of thread support (integer)
OUT	provided	provided level of thread support (integer)

C binding

`int MPI_T_init_thread(int required, int *provided)`

All programs or tools that use the MPI tool information interface must initialize the MPI tool information interface in the processes that will use the interface before calling any other of its routines. A user can initialize the MPI tool information interface by calling `MPI_T_INIT_THREAD`, which can be called multiple times. In addition, this routine initializes the thread environment for all routines in the MPI tool information interface. Calling this routine when the MPI information interface is already initialized has no effect beyond increasing the reference count of how often the interface has been initialized. The argument `required` is used to specify the desired level of thread support. The possible values and their semantics are identical to the ones that can be used with `MPI_INIT_THREAD` listed in Section 11.6. The call returns in `provided` information about the actual level of thread support that will be provided by the MPI implementation for calls to MPI tool information interface routines. It can be one of the four values listed in Section 11.6.

The MPI specification does not require all MPI processes to exist before MPI is initialized. If the MPI tool information interface is used before initialization of MPI, the user is responsible for ensuring that the MPI tool information interface is initialized on all processes it is used in. Processes created by the MPI implementation during initialization inherit the status of the MPI tool information interface (whether it is initialized or not as well as all active sessions and handles) from the process from which they are created.

Processes created at runtime as a result of calls to MPI's dynamic process management require their own initialization before they can use the MPI tool information interface.

Advice to users. If `MPI_T_INIT_THREAD` is called before `MPI_INIT_THREAD`, the requested and provided thread level for `MPI_T_INIT_THREAD` may influence the behavior and return value of `MPI_INIT_THREAD`. The same is true for the reverse order. Likewise, when using the Sessions Model (Section 11.3), the requested and

provided thread level for `MPI_T_INIT_THREAD` may influence the behavior and return values of `MPI_SESSION_INIT` (see Section 11.3), with the same being true for the reverse order. (*End of advice to users.*)

Advice to implementors. MPI implementations should strive to make as many control or performance variables available before MPI initialization (instead of adding them during initialization) to allow tools the most flexibility. In particular, control variables should be available before MPI initialization if their value cannot be changed after MPI initialization. (*End of advice to implementors.*)

MPI_T_FINALIZE()

C binding

```
int MPI_T_finalize(void)
```

This routine finalizes the use of the MPI tool information interface and may be called as often as the corresponding `MPI_T_INIT_THREAD` routine up to the current point of execution. Calling it more times returns a corresponding error code. As long as the number of calls to `MPI_T_FINALIZE` is smaller than the number of calls to `MPI_T_INIT_THREAD` up to the current point of execution, the MPI tool information interface remains initialized and calls to its routines are permissible. Further, additional calls to `MPI_T_INIT_THREAD` after one or more calls to `MPI_T_FINALIZE` are permissible.

Once `MPI_T_FINALIZE` is called the same number of times as the routine `MPI_T_INIT_THREAD` up to the current point of execution, the MPI tool information interface is no longer initialized. The user can reinitialize the interface by a subsequent call to `MPI_T_INIT_THREAD`.

At the end of the program execution, unless `MPI_ABORT` is called, an application must have called `MPI_T_INIT_THREAD` and `MPI_T_FINALIZE` an equal number of times.

15.3.5 Datatype System

All variables managed through the MPI tool information interface represent their values through typed buffers of a given length and type using an MPI datatype (similar to regular send/receive buffers). Since the initialization of the MPI tool information interface is separate from the initialization of MPI, MPI tool information interface routines can be called before MPI initialization. Consequently, these routines can also use MPI datatypes before MPI initialization. Therefore, within the context of the MPI tool information interface, it is permissible to use a subset of MPI datatypes as specified below before MPI initialization.

Rationale. The MPI tool information interface relies mainly on unsigned datatypes for integer values since most variables are expected to represent counters or resource sizes. `MPI_INT` is provided for additional flexibility and is expected to be used mainly for control variables and enumeration types (see below).

Providing all basic datatypes, in particular providing all signed and unsigned variants of integer types, would lead to a larger number of types, which tools need to interpret. This would cause unnecessary complexity in the implementation of tools based on the MPI tool information interface. (*End of rationale.*)

Table 15.3: MPI datatypes that can be used by the MPI tool information interface

```

MPI_INT
MPI_INT32_T
MPI_INT64_T
MPI_UNSIGNED
MPI_UNSIGNED_LONG
MPI_UNSIGNED_LONG_LONG
MPI_UINT32_T
MPI_UINT64_T
MPI_COUNT
MPI_CHAR
MPI_DOUBLE

```

The MPI tool information interface only relies on a subset of the basic MPI datatypes and does not use any derived MPI datatypes. Table 15.3 lists all MPI datatypes that can be returned by the MPI tool information interface to represent its variables.

The use of the datatype `MPI_CHAR` in the MPI tool information interface implies a null-terminated character array, i.e., a string in the C language. If a variable has type `MPI_CHAR`, the value of the count parameter returned by `MPI_T_CVAR_HANDLE_ALLOC` and `MPI_T_PVAR_HANDLE_ALLOC` must be large enough to include any valid value, including its terminating null character. The contents of returned `MPI_CHAR` arrays are only defined from index 0 through the location of the first null character.

Rationale. The MPI tool information interface requires a significantly simpler type system than MPI itself. Therefore, only its required subset must be present before MPI initialization and MPI implementations do not need to initialize the complete MPI datatype system. (*End of rationale.*)

For variables of type `MPI_INT`, an MPI implementation can provide additional information by associating names with a fixed number of values. We refer to this information in the following as an enumeration. In this case, the respective calls that provide additional metadata for each control or performance variable, i.e., `MPI_T_CVAR_GET_INFO` (Section 15.3.6), `MPI_T_PVAR_GET_INFO` (Section 15.3.7), and `MPI_T_EVENT_GET_INFO` (Section 15.3.8), return a handle of type `MPI_T_enum` that can be passed to the following functions to extract additional information. Thus, the MPI implementation can describe variables with a fixed set of values that each represents a particular state. Each enumeration type can have N different values, with a fixed N that can be queried using `MPI_T_ENUM_GET_INFO`.

```
MPI_T_ENUM_GET_INFO(enumtype, num, name, name_len)
```

IN	<code>enumtype</code>	enumeration to be queried (handle)
OUT	<code>num</code>	number of discrete values represented by this enumeration (integer)
OUT	<code>name</code>	buffer to return the string containing the name of the enumeration item (string)

INOUT name_len length of the string and/or buffer for name (integer)

C binding

```
int MPI_T_enum_get_info(MPI_T_enum enumtype, int *num, char *name,
                       int *name_len)
```

If `enumtype` is a valid enumeration, this routine returns the number of items represented by this enumeration type as well as its name. N must be greater than 0, i.e., the enumeration must represent at least one value.

The arguments `name` and `name_len` are used to return the name of the enumeration as described in Section 15.3.3.

The routine is required to return a name of at least length one. This name must be unique with respect to all other names for enumerations that the MPI implementation uses.

Names associated with individual values in each enumeration `enumtype` can be queried using `MPI_T_ENUM_GET_ITEM`.

```
MPI_T_ENUM_GET_ITEM(enumtype, index, value, name, name_len)
```

IN	enumtype	enumeration to be queried (handle)
IN	index	number of the value to be queried in this enumeration (integer)
OUT	value	variable value (integer)
OUT	name	buffer to return the string containing the name of the enumeration item (string)
INOUT	name_len	length of the string and/or buffer for name (integer)

C binding

```
int MPI_T_enum_get_item(MPI_T_enum enumtype, int index, int *value, char *name,
                       int *name_len)
```

The arguments `name` and `name_len` are used to return the name of the enumeration item as described in Section 15.3.3.

If completed successfully, the routine returns the name/value pair that describes the enumeration at the specified index. The call is further required to return a name of at least length one. This name must be unique with respect to all other names of items for the same enumeration.

15.3.6 Control Variables

The routines described in this section of the MPI tool information interface specification focus on the ability to list, query, and possibly set control variables exposed by the MPI implementation. These variables can typically be used by the user to fine tune properties and configuration settings of the MPI implementation. On many systems, such variables can be set using environment variables, although other configuration mechanisms may be available, such as configuration files or central configuration registries. A typical example that is available in several existing MPI implementations is the ability to specify an “eager limit,” i.e., an upper bound on the size of messages sent or received using an eager protocol.

Control Variable Query Functions

An MPI implementation exports a set of N control variables through the MPI tool information interface. If N is zero, then the MPI implementation does not export any control variables, otherwise the provided control variables are indexed from 0 to $N - 1$. This index number is used in subsequent calls to identify the individual variables.

An MPI implementation is allowed to increase the number of control variables during the execution of an MPI application when new variables become available through dynamic loading. However, MPI implementations are not allowed to change the index of a control variable or to delete a variable once it has been added to the set. When a variable becomes inactive, e.g., through dynamic unloading, accessing its value should return a corresponding error code.

Advice to users. While the MPI tool information interface guarantees that indices or variable properties do not change during a particular run of an MPI program, it does not provide a similar guarantee between runs. (*End of advice to users.*)

The following function can be used to query the number of control variables, `num_cvar`:

```
MPI_T_CVAR_GET_NUM(num_cvar)
```

OUT `num_cvar` returns number of control variables (integer)

C binding

```
int MPI_T_cvar_get_num(int *num_cvar)
```

The function `MPI_T_CVAR_GET_INFO` provides access to additional information for each variable.

```
MPI_T_CVAR_GET_INFO(cvar_index, name, name_len, verbosity, datatype, enumtype, desc,
                    desc_len, bind, scope)
```

IN	<code>cvar_index</code>	index of the control variable to be queried, value between 0 and <code>num_cvar - 1</code> (integer)
OUT	<code>name</code>	buffer to return the string containing the name of the control variable (string)
INOUT	<code>name_len</code>	length of the string and/or buffer for <code>name</code> (integer)
OUT	<code>verbosity</code>	verbosity level of this variable (integer)
OUT	<code>datatype</code>	MPI datatype of the information stored in the control variable (handle)
OUT	<code>enumtype</code>	optional descriptor for enumeration information (handle)
OUT	<code>desc</code>	buffer to return the string containing a description of the control variable (string)
INOUT	<code>desc_len</code>	length of the string and/or buffer for <code>desc</code> (integer)
OUT	<code>bind</code>	type of MPI object to which this variable must be bound (integer)

Table 15.4: Scopes for control variables

Scope Constant	Description
MPI_T_SCOPE_CONSTANT	read-only, value is constant
MPI_T_SCOPE_READONLY	read-only, cannot be written, but can change
MPI_T_SCOPE_LOCAL	may be writeable, writing only affects the local MPI process
MPI_T_SCOPE_GROUP	may be writeable, must be set to consistent values across a group of connected MPI processes
MPI_T_SCOPE_GROUP_EQ	may be writeable, must be set to the same value across a group of connected MPI processes
MPI_T_SCOPE_ALL	may be writeable, must be set to consistent values across all connected MPI processes
MPI_T_SCOPE_ALL_EQ	may be writeable, must be set to the same value across all connected MPI processes

changeable; it is not a guarantee that it can be changed at any time. (*End of advice to users.*)

MPI_T_CVAR_GET_INDEX(name, cvar_index)

IN	name	name of the control variable (string)
OUT	cvar_index	index of the control variable (integer)

C binding

```
int MPI_T_cvar_get_index(const char *name, int *cvar_index)
```

MPI_T_CVAR_GET_INDEX is a function for retrieving the index of a control variable given a known variable name. The name parameter is provided by the caller, and cvar_index is returned by the MPI implementation. The name parameter is a string terminated with a null character.

This routine returns MPI_SUCCESS on success and returns MPI_T_ERR_INVALID_NAME if name does not match the name of any control variable provided by the implementation at the time of the call.

Rationale. This routine is provided to enable fast retrieval of control variables by a tool, assuming it knows the name of the variable for which it is looking. The number of variables exposed by the implementation can change over time, so it is not possible for the tool to simply iterate over the list of variables once at initialization. Although using MPI implementation specific variable names is not portable across MPI implementations, tool developers may choose to take this route for lower overhead at runtime because the tool will not have to iterate over the entire set of variables to find a specific one. (*End of rationale.*)

Example 15.5. Querying and printing the names of all available control variables.

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char *argv[]) {
    int i, err, num, namelen, bind, verbose, scope;
    int threadsupport;
    char name[100];

    MPI_Datatype datatype;

    err=MPI_T_init_thread(MPI_THREAD_SINGLE, &threadsupport);
    if (err!=MPI_SUCCESS)
        return err;

    err=MPI_T_cvar_get_num(&num);
    if (err!=MPI_SUCCESS)
        return err;

    for (i=0; i<num; i++) {
        namelen=100;
        err=MPI_T_cvar_get_info(i, name, &namelen,
                               &verbose, &datatype, NULL,
                               NULL, NULL, /*no description */
                               &bind, &scope);
        if (err!=MPI_SUCCESS && err!=MPI_T_ERR_INVALID_INDEX)
            return err;
        printf("Var %i: %s\n", i, name);
    }

    err=MPI_T_finalize();
    if (err!=MPI_SUCCESS)
        return 1;
    else
        return 0;
}

```

Handle Allocation and Deallocation

Before reading or writing the value of a variable, a user must first allocate a handle of type `MPI_T_cvar_handle` for the variable by binding it to an MPI object (see also Section 15.3.2).

Rationale. Handles used in the MPI tool information interface are distinct from handles used in the remaining parts of the MPI standard because they must be usable before MPI is initialized and after MPI is finalized. Further, accessing handles, in particular for performance variables, can be time critical and having a separate handle space enables optimizations. (*End of rationale.*)

```

1 MPI_T_CVAR_HANDLE_ALLOC(cvar_index, obj_handle, handle, count)
2     IN      cvar_index      index of control variable for which handle is to be
3                               allocated (index)
4
5     IN      obj_handle      reference to a handle of the MPI object to which this
6                               variable is supposed to be bound (pointer)
7
8     OUT     handle          allocated handle (handle)
9
10    OUT     count           number of elements used to represent this variable
11                               (integer)

```

C binding

```

12 int MPI_T_cvar_handle_alloc(int cvar_index, void *obj_handle,
13                             MPI_T_cvar_handle *handle, int *count)
14

```

15 This routine binds the control variable specified by the argument `index` to an MPI object.
16 The object is passed in the argument `obj_handle` as an address to a local variable that stores
17 the object's handle. The argument `obj_handle` is ignored if the `MPI_T_CVAR_GET_INFO`
18 call for this control variable returned `MPI_T_BIND_NO_OBJECT` in the argument `bind`. The
19 handle allocated to reference the variable is returned in the argument `handle`. Upon success-
20 ful return, `count` contains the number of elements (of the datatype returned by a previous
21 `MPI_T_CVAR_GET_INFO` call) used to represent this variable.

22
23 *Advice to users.* The `count` can be different based on the MPI object to which the
24 control variable was bound. For example, variables bound to communicators could
25 have a count that matches the size of the communicator.

26 It is not portable to pass references to predefined MPI object handles, such as
27 `MPI_COMM_WORLD` to this routine, since their implementation depends on the MPI
28 library. Instead, such object handles should be stored in a local variable and the
29 address of this local variable should be passed into `MPI_T_CVAR_HANDLE_ALLOC`.
30 (*End of advice to users.*)

31
32 The value of `cvar_index` should be in the range from 0 to `num_cvar - 1`, where `num_cvar`
33 is the number of available control variables as determined from a prior call to
34 `MPI_T_CVAR_GET_NUM`. The type of the MPI object it references must be consistent with
35 the type returned in the `bind` argument in a prior call to `MPI_T_CVAR_GET_INFO`.
36

```

37
38 MPI_T_CVAR_HANDLE_FREE(handle)
39     INOUT   handle          handle to be freed (handle)
40

```

C binding

```

41
42 int MPI_T_cvar_handle_free(MPI_T_cvar_handle *handle)
43

```

44 When a handle is no longer needed, a user of the MPI tool information interface should
45 call `MPI_T_CVAR_HANDLE_FREE` to free the handle and the associated resources in the
46 MPI implementation. On a successful return, MPI sets the handle to
47 `MPI_T_CVAR_HANDLE_NULL`.
48

Control Variable Access Functions

MPI_T_CVAR_READ(handle, buf)

IN	handle	handle to the control variable to be read (handle)
OUT	buf	initial address of storage location for variable value (choice)

C binding

```
int MPI_T_cvar_read(MPI_T_cvar_handle handle, void *buf)
```

This routine queries the value of a control variable identified by the argument `handle` and stores the result in the buffer identified by the parameter `buf`. The user must ensure that the buffer is of the appropriate size to hold the entire value of the control variable (based on the returned datatype and count from prior corresponding calls to `MPI_T_CVAR_GET_INFO` and `MPI_T_CVAR_HANDLE_ALLOC`, respectively).

MPI_T_CVAR_WRITE(handle, buf)

INOUT	handle	handle to the control variable to be written (handle)
IN	buf	initial address of storage location for variable value (choice)

C binding

```
int MPI_T_cvar_write(MPI_T_cvar_handle handle, const void *buf)
```

This routine sets the value of the control variable identified by the argument `handle` to the data stored in the buffer identified by the parameter `buf`. The user must ensure that the buffer is of the appropriate size to hold the entire value of the control variable (based on the returned datatype and count from prior corresponding calls to `MPI_T_CVAR_GET_INFO` and `MPI_T_CVAR_HANDLE_ALLOC`, respectively).

If the variable has a global scope (as returned by a prior corresponding `MPI_T_CVAR_GET_INFO` call), any write call to this variable must be issued by the user in all connected (as defined in Section 11.10.4) MPI processes. If the variable has group scope, any write call to this variable must be issued by the user in all MPI processes in the group, which must be described by the MPI implementation in the description by the `MPI_T_CVAR_GET_INFO`.

In both cases, the user must ensure that the writes in all participating MPI processes are consistent. If the scope is either `MPI_T_SCOPE_ALL_EQ` or `MPI_T_SCOPE_GROUP_EQ` this means that the variable in all connected MPI processes or MPI processes of the group, respectively, must be set to the same value.

If it is not possible to change the variable at the time the call is made, the function returns either `MPI_T_ERR_CVAR_SET_NOT_NOW`, if there may be a later time at which the variable could be set, or `MPI_T_ERR_CVAR_SET_NEVER`, if the variable cannot be set for the remainder of the application's execution.

Example 15.6. Reading the value of a control variable.

```

1  int getValue_int_comm(int index, MPI_Comm comm, int *val) {
2
3      int err, count;
4      MPI_T_cvar_handle handle;
5
6      /* This example assumes that the variable index */
7      /* can be bound to a communicator */
8
9      err=MPI_T_cvar_handle_alloc(index, &comm, &handle, &count);
10     if (err!=MPI_SUCCESS)
11         return err;
12
13     /* The following assumes that the variable is */
14     /* represented by a single integer */
15
16     err=MPI_T_cvar_read(handle, val);
17     if (err!=MPI_SUCCESS)
18         return err;
19
20     err=MPI_T_cvar_handle_free(&handle);
21     return err;
22 }

```

15.3.7 Performance Variables

The following section focuses on the ability to list and to query performance variables provided by the MPI implementation. Performance variables provide insight into MPI implementation-specific internals and can represent information such as the state of the MPI implementation (e.g., waiting blocked, receiving, not active), aggregated timing data for submodules, or queue sizes and lengths.

Rationale. The interface for performance variables is separate from the interface for control variables, since performance variables have different requirements and parameters. By keeping them separate, the interface provides cleaner semantics and allows for more performance optimization opportunities. (*End of rationale.*)

Some performance variables and classes refer to **events**. In general, such events describe state transitions within software or hardware related to the performance of an MPI application. The events offered through the callback-driven event-notification interface described in Section 15.3.8 also refer to such state transitions; however, the set of state transitions referred to by performance variables and events as described in Section 15.3.8 may not be identical.

Performance Variable Classes

Each performance variable is associated with a class that describes its basic semantics, possible datatypes, basic behavior, its starting value, whether it can overflow, and when and how an MPI implementation can change the variable's value. The starting value is the value that is assigned to the variable the first time that it is used or whenever it is reset.

Advice to users. If a performance variable belongs to a class that can overflow, it is up to the user to protect against this overflow, e.g., by frequently reading and resetting the variable value. (*End of advice to users.*)

Advice to implementors. MPI implementations should use large enough datatypes for each performance variable to avoid overflows under normal circumstances. (*End of advice to implementors.*)

The classes are defined by the following constants:

MPI_T_PVAR_CLASS_STATE: A performance variable in this class represents a set of discrete states. Variables of this class are represented by MPI_INT and can be set by the MPI implementation at any time. Variables of this type should be described further using an enumeration, as discussed in Section 15.3.5. The starting value is the current state of the implementation at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

MPI_T_PVAR_CLASS_LEVEL: A performance variable in this class represents a value that describes the utilization level of a resource. The value of a variable of this class can change at any time to match the current utilization level of the resource. Values returned from variables in this class are nonnegative and represented by one of the following datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG, MPI_DOUBLE. The starting value is the current utilization level of the resource at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

MPI_T_PVAR_CLASS_SIZE: A performance variable in this class represents a value that is the size of a resource. Values returned from variables in this class are nonnegative and represented by one of the following datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG, MPI_DOUBLE. The starting value is the current size of the resource at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

MPI_T_PVAR_CLASS_PERCENTAGE: The value of a performance variable in this class represents the percentage utilization of a finite resource. The value of a variable of this class can change at any time to match the current utilization level of the resource. It will be returned as an MPI_DOUBLE datatype. The value must always be between 0.0 (resource not used at all) and 1.0 (resource completely used). The starting value is the current percentage utilization level of the resource at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

MPI_T_PVAR_CLASS_HIGHWATERMARK: A performance variable in this class represents a value that describes the high watermark utilization of a resource. The value of a variable of this class is nonnegative and grows monotonically from the initialization or reset of the variable. It can be represented by one of the following datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG, MPI_DOUBLE. The starting value is the current utilization level of the resource at the time that the variable is started or reset. MPI implementations must ensure that variables of this class cannot overflow.

1 **MPI_T_PVAR_CLASS_LOWWATERMARK:** A performance variable in this class represents
2 a value that describes the low watermark utilization of a resource. The value of a
3 variable of this class is nonnegative and decreases monotonically from the initialization
4 or reset of the variable. It can be represented by one of the following datatypes:
5 MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG,
6 MPI_DOUBLE. The starting value is the current utilization level of the resource at the
7 time that the variable is started or reset. MPI implementations must ensure that
8 variables of this class cannot overflow.

9
10 **MPI_T_PVAR_CLASS_COUNTER:** A performance variable in this class counts the number
11 of occurrences of a specific event (e.g., the number of memory allocations within an
12 MPI library). The value of a variable of this class increases monotonically from the
13 initialization or reset of the performance variable by one for each specific event that
14 is observed. Values must be nonnegative and represented by one of the following
15 datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG. The
16 starting value for variables of this class is 0. Variables of this class can overflow.

17 **MPI_T_PVAR_CLASS_AGGREGATE:** The value of a performance variable in this class is an
18 an aggregated value that represents a sum of arguments processed during a specific
19 event (e.g., the amount of memory allocated by all memory allocations). This class is
20 similar to the counter class, but instead of counting individual events, the value can
21 be incremented by arbitrary amounts. The value of a variable of this class increases
22 monotonically from the initialization or reset of the performance variable. It must
23 be nonnegative and represented by one of the following datatypes: MPI_UNSIGNED,
24 MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG, MPI_DOUBLE. The starting value
25 for variables of this class is 0. Variables of this class can overflow.

26
27 **MPI_T_PVAR_CLASS_TIMER:** The value of a performance variable in this class represents
28 the aggregated time that the MPI implementation spends executing a particular event,
29 type of event, or section of the MPI library. This class has the same basic semantics
30 as MPI_T_PVAR_CLASS_AGGREGATE, but explicitly records a timing value. The value
31 of a variable of this class increases monotonically from the initialization or reset of the
32 performance variable. It must be nonnegative and represented by one of the following
33 datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG,
34 MPI_DOUBLE. The starting value for variables of this class is 0. If the type
35 MPI_DOUBLE is used, the units that represent time in this datatype must match the
36 units used by MPI_WTIME. Otherwise, the time units should be documented, e.g.,
37 in the description returned by MPI_T_PVAR_GET_INFO. Variables of this class can
38 overflow.

39
40 **MPI_T_PVAR_CLASS_GENERIC:** This class can be used to describe a variable that does
41 not fit into any of the other classes. For variables in this class, the starting value is
42 variable-specific and implementation-defined.

43 Performance Variable Query Functions

44
45 An MPI implementation exports a set of N performance variables through the MPI tool
46 information interface. If N is zero, then the MPI implementation does not export any
47 performance variables; otherwise the provided performance variables are indexed from 0 to
48 $N - 1$. This index number is used in subsequent calls to identify the individual variables.

An MPI implementation is allowed to increase the number of performance variables during the execution of an MPI application when new variables become available through dynamic loading. However, MPI implementations are not allowed to change the index of a performance variable or to delete a variable once it has been added to the set. When a variable becomes inactive, e.g., through dynamic unloading, accessing its value should return a corresponding error code.

The following function can be used to query the number of performance variables, `num_pvar`:

`MPI_T_PVAR_GET_NUM(num_pvar)`

OUT `num_pvar` returns number of performance variables (integer)

C binding

`int MPI_T_pvar_get_num(int *num_pvar)`

The function `MPI_T_PVAR_GET_INFO` provides access to additional information for each variable.

`MPI_T_PVAR_GET_INFO(pvar_index, name, name_len, verbosity, var_class, datatype, enumtype, desc, desc_len, bind, readonly, continuous, atomic)`

IN	<code>pvar_index</code>	index of the performance variable to be queried between 0 and <code>num_pvar - 1</code> (integer)
OUT	<code>name</code>	buffer to return the string containing the name of the performance variable (string)
INOUT	<code>name_len</code>	length of the string and/or buffer for <code>name</code> (integer)
OUT	<code>verbosity</code>	verbosity level of this variable (integer)
OUT	<code>var_class</code>	class of performance variable (integer)
OUT	<code>datatype</code>	MPI datatype of the information stored in the performance variable (handle)
OUT	<code>enumtype</code>	optional descriptor for enumeration information (handle)
OUT	<code>desc</code>	buffer to return the string containing a description of the performance variable (string)
INOUT	<code>desc_len</code>	length of the string and/or buffer for <code>desc</code> (integer)
OUT	<code>bind</code>	type of MPI object to which this variable must be bound (integer)
OUT	<code>readonly</code>	flag indicating whether the variable can be written/reset (integer)
OUT	<code>continuous</code>	flag indicating whether the variable can be started and stopped or is continuously active (integer)
OUT	<code>atomic</code>	flag indicating whether the variable can be atomically read and reset (integer)

C binding

```

1 int MPI_T_pvar_get_info(int pvar_index, char *name, int *name_len,
2     int *verbosity, int *var_class, MPI_Datatype *datatype,
3     MPI_T_enum *enumtype, char *desc, int *desc_len, int *bind,
4     int *readonly, int *continuous, int *atomic)
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

```

After a successful call to `MPI_T_PVAR_GET_INFO` for a particular variable, subsequent calls to this routine that query information about the same variable must return the same information. An MPI implementation is not allowed to alter any of the returned values.

If any OUT parameter to `MPI_T_PVAR_GET_INFO` is a NULL pointer, the implementation will ignore the parameter and not return a value for the parameter.

The arguments `name` and `name_len` are used to return the name of the performance variable as described in Section 15.3.3. If completed successfully, the routine is required to return a name of at least length one.

The argument `verbosity` returns the verbosity level of the variable (see Section 15.3.1).

The class of the performance variable is returned in the parameter `var_class`. The class must be one of the constants defined in Section 15.3.7.

The combination of the name and the class of the performance variable must be unique with respect to all other names for performance variables used by the MPI implementation.

Advice to implementors. Groups of variables that belong closely together, but have different classes, can have the same name. This choice is useful, e.g., to refer to multiple variables that describe a single resource (like the level, the total size, as well as high and low watermarks). (*End of advice to implementors.*)

The argument `datatype` returns the MPI datatype that is used to represent the performance variable.

If the variable is of type `MPI_INT`, MPI can optionally specify an enumeration for the values represented by this variable and return it in `enumtype`. In this case, MPI returns an enumeration identifier, which can then be used to gather more information as described in Section 15.3.5. Otherwise, `enumtype` is set to `MPI_T_ENUM_NULL`. If the datatype is not `MPI_INT` or the argument `enumtype` is the null pointer, no enumeration type is returned.

Returning a description is optional. If an MPI implementation does not return a description, the first character for `desc` must be set to the null character and `desc_len` must be set to one at the return from this function.

The parameter `bind` returns the type of the MPI object to which the variable must be bound or the value `MPI_T_BIND_NO_OBJECT` (see Section 15.3.2).

Upon return, the argument `readonly` is set to zero if the variable can be written or reset by the user. It is set to one if the variable can only be read.

Upon return, the argument `continuous` is set to zero if the variable can be started and stopped by the user, i.e., it is possible for the user to control if and when the value of a variable is updated. It is set to one if the variable is always active and cannot be controlled by the user.

Upon return, the argument `atomic` is set to zero if the variable cannot be read and reset atomically. Only variables for which the call sets `atomic` to one can be used in a call to `MPI_T_PVAR_READRESET`.

If a performance variable has an equivalent name and has the same class across connected MPI processes, the following OUT parameters must be identical: `verbosity`, `varclass`,

datatype, enumtype, bind, readonly, continuous, and atomic. The returned description must be equivalent.

MPI_T_PVAR_GET_INDEX(name, var_class, pvar_index)

IN	name	the name of the performance variable (string)
IN	var_class	the class of the performance variable (integer)
OUT	pvar_index	the index of the performance variable (integer)

C binding

int MPI_T_pvar_get_index(const char *name, int var_class, int *pvar_index)

MPI_T_PVAR_GET_INDEX is a function for retrieving the index of a performance variable given a known variable name and class. The `name` and `var_class` parameters are provided by the caller, and `pvar_index` is returned by the MPI implementation. The `name` parameter is a string terminated with a null character.

This routine returns MPI_SUCCESS on success and returns MPI_T_ERR_INVALID_NAME if `name` does not match the name of any performance variable of the specified `var_class` provided by the implementation at the time of the call.

Rationale. This routine is provided to enable fast retrieval of performance variables by a tool, assuming it knows the name of the variable for which it is looking. The number of variables exposed by the implementation can change over time, so it is not possible for the tool to simply iterate over the list of variables once at initialization. Although using MPI implementation specific variable names is not portable across MPI implementations, tool developers may choose to take this route for lower overhead at runtime because the tool will not have to iterate over the entire set of variables to find a specific one. (*End of rationale.*)

Performance Experiment Sessions

Within a single program, multiple components can use the MPI tool information interface. To avoid collisions with respect to accesses to performance variables, users of the MPI tool information interface must first create a performance experiment session. Subsequent calls that access performance variables can then be made within the context of this performance experiment session. Starting, stopping, reading, writing, or resetting a variable in one performance experiment session shall not influence whether a variable is started, stopped, read, written, or reset in another performance experiment session.

MPI_T_PVAR_SESSION_CREATE(pe_session)

OUT	pe_session	identifier of performance experiment session (handle)
-----	------------	---

C binding

int MPI_T_pvar_session_create(MPI_T_pvar_session *pe_session)

This call creates a new performance experiment session for accessing performance variables and returns a handle for this performance experiment session in the argument

1 pe_session of type MPI_T_pvar_session.

2
3
4 MPI_T_PVAR_SESSION_FREE(pe_session)

5 INOUT pe_session identifier of performance experiment session (handle)

6
7 **C binding**

8 int MPI_T_pvar_session_free(MPI_T_pvar_session *pe_session)

9
10 This call frees an existing performance experiment session. Calls to the MPI tool
11 information interface can no longer be made within the context of a performance experiment
12 session after it is freed. On a successful return, MPI sets the performance experiment session
13 identifier to MPI_T_PVAR_SESSION_NULL.

14
15 **Handle Allocation and Deallocation**

16 Before using a performance variable, a user must first allocate a handle of type
17 MPI_T_pvar_handle for the variable by binding it to an MPI object (see also Section 15.3.2).

18
19
20 MPI_T_PVAR_HANDLE_ALLOC(pe_session, pvar_index, obj_handle, handle, count)

21 INOUT pe_session identifier of performance experiment session (handle)

22 IN pvar_index index of performance variable for which handle is to
23 be allocated (integer)

24
25 IN obj_handle reference to a handle of the MPI object to which this
26 variable is supposed to be bound (pointer)

27 OUT handle allocated handle (handle)

28 OUT count number of elements used to represent this variable
29 (integer)

30
31
32 **C binding**

33 int MPI_T_pvar_handle_alloc(MPI_T_pvar_session pe_session, int pvar_index,
34 void *obj_handle, MPI_T_pvar_handle *handle, int *count)

35
36 This routine binds the performance variable specified by the argument index to an MPI
37 object in the performance experiment session identified by the parameter pe_session. The
38 object is passed in the argument obj_handle as an address to a local variable that stores
39 the object's handle. The argument obj_handle is ignored if the MPI_T_PVAR_GET_INFO
40 call for this performance variable returned MPI_T_BIND_NO_OBJECT in the argument bind.
41 The handle allocated to reference the variable is returned in the argument handle. Upon
42 successful return, count contains the number of elements (of the datatype returned by a
43 previous MPI_T_PVAR_GET_INFO call) used to represent this variable.

44 *Advice to users.* The count can be different based on the MPI object to which the
45 performance variable was bound. For example, variables bound to communicators
46 could have a count that matches the size of the communicator.
47
48

It is not portable to pass references to predefined MPI object handles, such as MPI_COMM_WORLD, to this routine, since their implementation depends on the MPI library. Instead, such an object handle should be stored in a local variable and the address of this local variable should be passed into MPI_T_PVAR_HANDLE_ALLOC. (*End of advice to users.*)

The value of index should be in the range from 0 to num_pvar – 1, where num_pvar is the number of available performance variables as determined from a prior call to MPI_T_PVAR_GET_NUM. The type of the MPI object it references must be consistent with the type returned in the bind argument in a prior call to MPI_T_PVAR_GET_INFO.

For all routines in the rest of this section that take both handle and pe_session as IN or INOUT arguments, if the handle argument passed in is not associated with the pe_session argument, MPI_T_ERR_INVALID_HANDLE is returned.

MPI_T_PVAR_HANDLE_FREE(pe_session, handle)

INOUT	pe_session	identifier of performance experiment session (handle)
INOUT	handle	handle to be freed (handle)

C binding

```
int MPI_T_pvar_handle_free(MPI_T_pvar_session pe_session,
                          MPI_T_pvar_handle *handle)
```

When a handle is no longer needed, a user of the MPI tool information interface should call MPI_T_PVAR_HANDLE_FREE to free the handle in the performance experiment session identified by the parameter pe_session and the associated resources in the MPI implementation. On a successful return, MPI sets the handle to MPI_T_PVAR_HANDLE_NULL.

Starting and Stopping of Performance Variables

Performance variables that have the continuous flag set during the query procedure are continuously updated once a handle has been allocated. Such variables may be queried at any time, but they cannot be started or stopped by the user. All other variables are in a stopped state after their handle has been allocated; their values are not updated until they have been started by the user.

MPI_T_PVAR_START(pe_session, handle)

IN	pe_session	identifier of performance experiment session (handle)
INOUT	handle	handle of a performance variable (handle)

C binding

```
int MPI_T_pvar_start(MPI_T_pvar_session pe_session, MPI_T_pvar_handle handle)
```

This functions starts the performance variable with the handle identified by the parameter handle in the performance experiment session identified by the parameter pe_session.

If the constant MPI_T_PVAR_ALL_HANDLES is passed in handle, the MPI implementation attempts to start all variables within the performance experiment session iden-

1 tified by the parameter `pe_session` for which handles have been allocated. In this case,
 2 the routine returns `MPI_SUCCESS` if all variables are started successfully (even if there are
 3 no noncontinuous variables to be started), otherwise `MPI_T_ERR_PVAR_NO_STARTSTOP` is
 4 returned. Continuous variables and variables that are already started are ignored when
 5 `MPI_T_PVAR_ALL_HANDLES` is specified.

6
 7
 8 `MPI_T_PVAR_STOP(pe_session, handle)`

9 IN `pe_session` identifier of performance experiment session (handle)

10 INOUT `handle` handle of a performance variable (handle)

12 **C binding**

13 `int MPI_T_pvar_stop(MPI_T_pvar_session pe_session, MPI_T_pvar_handle handle)`

15 This function stops the performance variable with the handle identified by the param-
 16 eter `handle` in the performance experiment session identified by the parameter `pe_session`.

17 If the constant `MPI_T_PVAR_ALL_HANDLES` is passed in `handle`, the MPI implementa-
 18 tion attempts to stop all variables within the performance experiment session identified
 19 by the parameter `pe_session` for which handles have been allocated. In this case, the rou-
 20 tine returns `MPI_SUCCESS` if all variables are stopped successfully (even if there are no
 21 noncontinuous variables to be stopped), otherwise `MPI_T_ERR_PVAR_NO_STARTSTOP` is re-
 22 turned. Continuous variables and variables that are already stopped are ignored when
 23 `MPI_T_PVAR_ALL_HANDLES` is specified.

25 Performance Variable Access Functions

26
 27
 28 `MPI_T_PVAR_READ(pe_session, handle, buf)`

29 IN `pe_session` identifier of performance experiment session (handle)

30 IN `handle` handle of a performance variable (handle)

31 OUT `buf` initial address of storage location for variable value
 32 (choice)

35 **C binding**

36 `int MPI_T_pvar_read(MPI_T_pvar_session pe_session, MPI_T_pvar_handle handle,`
 37 `void *buf)`

39 The `MPI_T_PVAR_READ` call queries the value of the performance variable with the
 40 handle `handle` in the performance experiment session identified by the parameter `pe_session`
 41 and stores the result in the buffer identified by the parameter `buf`. The user is responsible to
 42 ensure that the buffer is of the appropriate size to hold the entire value of the performance
 43 variable (based on the datatype and count returned by the corresponding previous calls to
 44 `MPI_T_PVAR_GET_INFO` and `MPI_T_PVAR_HANDLE_ALLOC`, respectively).

45 The constant `MPI_T_PVAR_ALL_HANDLES` cannot be used as an argument for the func-
 46 tion `MPI_T_PVAR_READ`.

MPI_T_PVAR_WRITE(pe_session, handle, buf)				1
IN	pe_session	identifier of performance experiment session (handle)		2
INOUT	handle	handle of a performance variable (handle)		3
IN	buf	initial address of storage location for variable value (choice)		4
				5
				6
				7

C binding

```
int MPI_T_pvar_write(MPI_T_pvar_session pe_session, MPI_T_pvar_handle handle,
                    const void *buf)
```

The MPI_T_PVAR_WRITE call attempts to write the value of the performance variable with the handle identified by the parameter `handle` in the performance experiment session identified by the parameter `pe_session`. The value to be written is passed in the buffer identified by the parameter `buf`. The user must ensure that the buffer is of the appropriate size to hold the entire value of the performance variable (based on the datatype and count returned by the corresponding previous calls to MPI_T_PVAR_GET_INFO and MPI_T_PVAR_HANDLE_ALLOC, respectively).

If it is not possible to change the variable, the function returns MPI_T_ERR_PVAR_NO_WRITE.

The constant MPI_T_PVAR_ALL_HANDLES cannot be used as an argument for the function MPI_T_PVAR_WRITE.

MPI_T_PVAR_RESET(pe_session, handle)				24
IN	pe_session	identifier of performance experiment session (handle)		25
INOUT	handle	handle of a performance variable (handle)		26
				27
				28

C binding

```
int MPI_T_pvar_reset(MPI_T_pvar_session pe_session, MPI_T_pvar_handle handle)
```

The MPI_T_PVAR_RESET call sets the performance variable with the handle identified by the parameter `handle` to its starting value specified in Section 15.3.7. If it is not possible to change the variable, the function returns MPI_T_ERR_PVAR_NO_WRITE.

If the constant MPI_T_PVAR_ALL_HANDLES is passed in `handle`, the MPI implementation attempts to reset all variables within the performance experiment session identified by the parameter `pe_session` for which handles have been allocated. In this case, the routine returns MPI_SUCCESS if all variables are reset successfully (even if there are no valid handles or all are read-only), otherwise MPI_T_ERR_PVAR_NO_WRITE is returned. Read-only variables are ignored when MPI_T_PVAR_ALL_HANDLES is specified.

MPI_T_PVAR_READRESET(pe_session, handle, buf)				42
IN	pe_session	identifier of performance experiment session (handle)		43
INOUT	handle	handle of a performance variable (handle)		44
OUT	buf	initial address of storage location for variable value (choice)		45
				46
				47
				48

C binding

```
int MPI_T_pvar_readreset(MPI_T_pvar_session pe_session,
                        MPI_T_pvar_handle handle, void *buf)
```

This call atomically combines the functionality of `MPI_T_PVAR_READ` and `MPI_T_PVAR_RESET` with the same semantics as if these two calls were called separately. If the variable cannot be read and reset atomically, this routine returns `MPI_T_ERR_PVAR_NO_ATOMIC`.

The constant `MPI_T_PVAR_ALL_HANDLES` cannot be used as an argument for the function `MPI_T_PVAR_READRESET`.

Advice to implementors. Sampling-based tools rely on the ability to call the MPI tool information interface, in particular routines to start, stop, read, write, and reset performance variables, from any program context, including asynchronous contexts such as signal handlers. MPI implementations should strive, if possible in their particular environment, to enable these usage scenarios for all or a subset of the routines mentioned above. If implementing only a subset, the read, write, and reset routines are typically the most critical for sampling based tools. An MPI implementation should clearly document any restrictions on the program contexts in which the MPI tool information interface can be used. Restrictions might include guaranteeing usage outside of all signals or outside a specific set of signals. Any restrictions could be documented, for example, through the description returned by `MPI_T_PVAR_GET_INFO`. (*End of advice to implementors.*)

Rationale. All routines to read, to write or to reset performance variables require the performance experiment session argument. This requirement keeps the interface consistent and allows the use of `MPI_T_PVAR_ALL_HANDLES` where appropriate. Further, this opens up additional performance optimizations for the implementation of handles. (*End of rationale.*)

Example 15.7. Detecting Receives with long unexpected message queues.

The following example shows a sample tool to identify receive operations that occur during times with long message queues. This example assumes that the MPI implementation exports a variable with the name “`MPI_T_UMQ_LENGTH`” to represent the current length of the unexpected message queue. The tool is implemented as a PMPI tool using the MPI profiling interface.

The tool consists of three parts: (1) the initialization (by intercepting the call to `MPI_INIT`), (2) the test for long unexpected message queues (by intercepting calls to `MPI_RECV`), and (3) the clean-up phase (by intercepting the call to `MPI_FINALIZE`). To capture all receives, the example would have to be extended to have similar wrappers for all receive operations.

Part 1—Initialization: During initialization, the tool searches for the variable and, once the right index is found, allocates a performance experiment session and a handle for the variable with the found index, and starts the performance variable.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <mpi.h>
```

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
/* Global variables for the tool */
static MPI_T_pvar_session pe_session;
static MPI_T_pvar_handle handle;

int MPI_Init(int *argc, char ***argv ) {
    int err, num, i, index, namelen, verbosity;
    int var_class, bind, threadsup;
    int readonly, continuous, atomic, count;
    char name[18];

    MPI_Comm comm;
    MPI_Datatype datatype;
    MPI_T_enum enumtype;

    err=PMPI_Init(argc, argv);
    if (err!=MPI_SUCCESS)
        return err;

    err=PMPI_T_init_thread(MPI_THREAD_SINGLE, &threadsup);
    if (err!=MPI_SUCCESS)
        return err;

    err=PMPI_T_pvar_get_num(&num);
    if (err!=MPI_SUCCESS)
        return err;

    index=-1;
    i=0;
    while ((i<num) && (index<0) && (err==MPI_SUCCESS)) {
        /* Pass a buffer that is at least one character longer than */
        /* the name of the variable being searched for to avoid */
        /* finding variables that have a name that has a prefix */
        /* equal to the name of the variable being searched. */
        namelen=18;
        err=PMPI_T_pvar_get_info(i, name, &namelen, &verbosity,
                                &var_class, &datatype, &enumtype,
                                NULL, NULL, &bind,&readonly,
                                &continuous, &atomic);
        if (strcmp(name,"MPI_T_UMQ_LENGTH")==0) index=i;
        i++;
    }
    if (err!=MPI_SUCCESS)
        return err;

    /* this could be handled in a more flexible way for a generic tool */
    assert(index>=0);
    assert(var_class==MPI_T_PVAR_CLASS_LEVEL);
    assert(datatype==MPI_INT);
    assert(bind==MPI_T_BIND_MPI_COMM);

    /* Create a session */
    err=PMPI_T_pvar_session_create(&pe_session);
    if (err!=MPI_SUCCESS) return err;

    /* Get a handle and bind to MPI_COMM_WORLD */
    comm=MPI_COMM_WORLD;

```

```

1  err=PMPI_T_pvar_handle_alloc(pe_session, index, &comm, &handle,
2  &count);
3  if (err!=MPI_SUCCESS) return err;
4
5  /* this could be handled in a more flexible way for a generic tool */
6  assert(count==1);
7
8  /* Start variable */
9  err=PMPI_T_pvar_start(pe_session, handle);
10 if (err!=MPI_SUCCESS) return err;
11
12 return MPI_SUCCESS;
13 }

```

Part 2—Testing the Queue Lengths During Receives: During every receive operation, the tool reads the unexpected queue length through the matching performance variable and compares it against a predefined threshold.

```

14 #define THRESHOLD 5
15
16 int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
17             int tag, MPI_Comm comm, MPI_Status *status)
18 {
19     int value, err;
20
21     if (comm==MPI_COMM_WORLD) {
22         err=PMPI_T_pvar_read(pe_session, handle, &value);
23         if ((err==MPI_SUCCESS) && (value>THRESHOLD))
24         {
25             /* tool identified receive called with long UMQ */
26             /* execute tool functionality, */
27             /* e.g., gather and print call stack */
28         }
29     }
30
31     return PMPI_Recv(buf, count, datatype, source, tag, comm, status);
32 }

```

Part 3—Termination: In the wrapper for MPI_FINALIZE, the MPI tool information interface is finalized.

```

33 int MPI_Finalize(void)
34 {
35     int err;
36
37     err=PMPI_T_pvar_handle_free(pe_session, &handle);
38     err=PMPI_T_pvar_session_free(&pe_session);
39     err=PMPI_T_finalize();
40     return PMPI_Finalize();
41 }

```


15.3.8 Events

During the execution of an MPI application, the MPI implementation can raise *events* of a specific type to inform the user of a state change in the implementation. **Event types** describe specific state changes within the MPI implementation. In comparison to aggregate performance variables, events provide per-instance information on such state changes. The MPI implementation is said to **raise an event** when it invokes a callback function previously registered for the corresponding event type by the user. Each callback invocation for a specific event instance has a timestamp associated with it, which can be queried by the user, describing the time when the event was observed by the implementation. This decouples the observation of the state change from the communication of this information to the user. A timestamp in this context is a count of clock ticks elapsed since some time in the past and represented as a variable of type `MPI_Count`.

Event Sources

As a means to manage multiple state changes to be observed concurrently by different parts of the software and hardware system, the event interface of the MPI Tool Information Interface uses the concept of *sources*. A source in this context is a concept describing the logical entity raising the event. A source may or may not directly represent a concrete part of the software or hardware system. This concept is used primarily to describe partial ordering of events across different components where total ordering cannot necessarily be determined or is too costly to enforce.

The following function can be used to query the number of event sources, *num_sources*:

`MPI_T_SOURCE_GET_NUM(num_sources)`

OUT `num_sources` returns number of event sources (integer)

C binding

`int MPI_T_source_get_num(int *num_sources)`

The number of available event sources can be queried with a call to `MPI_T_SOURCE_GET_NUM`. An MPI implementation is allowed to increase the number of sources during the execution of an MPI process. However, MPI implementations are not allowed to change the index of an event source or to delete an event source once it has been made visible to the user (e.g., if new event sources become available via dynamic loading of additional components in the MPI implementation).

`MPI_T_SOURCE_GET_INFO(source_index, name, name_len, desc, desc_len, ordering, ticks_per_second, max_ticks, info)`

IN `source_index` index of the source to be queried between 0 and `num_sources - 1` (integer)

OUT `name` buffer to return the string containing the name of the source (string)

INOUT `name_len` length of the string and/or buffer for name (integer)

1	OUT	desc	buffer to return the string containing the description of the source (string)
2			
3	INOUT	desc_len	length of the string and/or buffer for desc (integer)
4			
5	OUT	ordering	flag indicating chronological ordering guarantees given by the source (integer)
6			
7	OUT	ticks_per_second	the number of ticks per second for the timer of this source (integer)
8			
9	OUT	max_ticks	the maximum count of ticks reported by this source before overflow occurs (integer)
10			
11			
12	OUT	info	optional info object (handle)

C binding

```

15 int MPI_T_source_get_info(int source_index, char *name, int *name_len,
16                          char *desc, int *desc_len, MPI_T_source_order *ordering,
17                          MPI_Count *ticks_per_second, MPI_Count *max_ticks,
18                          MPI_Info *info)

```

19 A call to `MPI_T_SOURCE_GET_INFO` returns additional information on the source
20 identified by the `source_index` argument.

21 The arguments `name` and `name_len` are used to return the name of the source as de-
22 scribed in Section 15.3.3.

23 The arguments `desc` and `desc_len` are used to return the description of the source as
24 described in Section 15.3.3.

25 The `ordering` argument returns whether event callbacks of this source will be invoked
26 in chronological order, i.e., the timestamps reported by `MPI_T_EVENT_GET_TIMESTAMP`
27 of subsequent events of the same source are monotonically increasing. The value of `ordering`
28 can be `MPI_T_SOURCE_ORDERED` or `MPI_T_SOURCE_UNORDERED`.

29 The `ticks_per_seconds` argument returns the number of ticks elapsed in one second for
30 the timer used for the specific source.

31 The `max_ticks` argument returns the largest number of ticks reported by this source as
32 a timestamp before the value overflows.

34 *Advice to users.* As the size of `MPI_Count` is defined in relation to the types `MPI_Aint`
35 and `MPI_Offset`, the effective size of `MPI_Count` may lead to overflows of the timestamp
36 values reported. Users can use the argument `max_ticks` to mitigate resulting problems.
37 (*End of advice to users.*)

39 MPI can optionally return an info object containing the default hints set for this source.
40 If the argument to `info` provided by the user is the `NULL` pointer, this argument is ignored,
41 otherwise an MPI implementation is required to return all hints that are supported by
42 the implementation for this source and have default values specified; any user-supplied
43 hints that were not ignored by the implementation; and any additional hints that were
44 set by the implementation. If no such hints exist, a handle to a newly created info object
45 is returned that contains no key/value pair. The user is responsible for freeing `info` via
46 `MPI_INFO_FREE`.

Table 15.5: Hierarchy of safety requirement levels for event callback routines

Safety Requirement
MPI_T_CB_REQUIRE_NONE
MPI_T_CB_REQUIRE_MPI_RESTRICTED
MPI_T_CB_REQUIRE_THREAD_SAFE
MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE

MPI_T_SOURCE_GET_TIMESTAMP(source_index, timestamp)

IN	source_index	index of the source (integer)
OUT	timestamp	current timestamp from specified source (integer)

C binding

```
int MPI_T_source_get_timestamp(int source_index, MPI_Count *timestamp)
```

To enable proper query of a reference timestamp for a specific source, a user can obtain a current timestamp using `MPI_T_SOURCE_GET_TIMESTAMP`. The argument `source_index` identifies the index of the source to query. The call returns `MPI_SUCCESS` and a current timestamp in the argument `timestamp` if the source supports ad-hoc generation of timestamps. The call returns `MPI_T_ERR_INVALID_INDEX` if the index does not identify a valid source. The call returns `MPI_T_ERR_NOT_SUPPORTED` if the source does not support the ad-hoc generation of timestamps.

Callback Safety Requirements

The actions a user is allowed to perform inside a callback function may vary with its execution context. As the user has no control over the execution context of specific callback function invocations, MPI provides a way to communicate this information using callback safety levels.

Table 15.5 provides the hierarchy of callback safety requirements levels within user-defined callback functions. The MPI implementation provides the safety requirement as an argument to the callback when it is invoked.

The level of `MPI_T_CB_REQUIRE_NONE` is the lowest level and does not impose any restrictions on the callback function.

The level of `MPI_T_CB_REQUIRE_MPI_RESTRICTED` restricts the set of MPI functions that can be called from inside the callback to all functions with the prefix `MPI_T` as well as `MPI_WTICK` and `MPI_WTIME`.

Advice to users. While some MPI functions are safe to be called inside a callback function used in the MPI tool information interface—which may in some implementations be issued from asynchronous contexts such as signal handlers—this does not imply that those MPI functions are generally safe to be called in asynchronous contexts such as signal handlers. (*End of advice to users.*)

The level of `MPI_T_CB_REQUIRE_THREAD_SAFE` includes all the limitations of `MPI_T_CB_REQUIRE_MPI_RESTRICTED` and additionally requires the callback to be reentrant

Table 15.6: List of MPI functions that when called from within a callback function may not return MPI_T_ERR_NOT_ACCESSIBLE

```

MPI_T_EVENT_COPY
MPI_T_EVENT_GET_SOURCE
MPI_T_EVENT_GET_TIMESTAMP
MPI_T_EVENT_READ
MPI_T_PVAR_READ
MPI_T_PVAR_READRESET
MPI_T_PVAR_RESET
MPI_T_PVAR_START
MPI_T_PVAR_STOP
MPI_T_PVAR_WRITE
MPI_T_SOURCE_GET_TIMESTAMP

```

and thread-safe. This means the callback must allow its execution to be interrupted by or happen concurrently with any other callback including itself.

The level of MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE includes all the limitations of MPI_T_CB_REQUIRE_THREAD_SAFE and additionally requires the callback to meet the safety requirements needed to support invocations from asynchronous contexts, such as signal handlers.

Advice to users. It is always safe to assume the highest restrictions for a callback invocation (i.e., MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE). By evaluating the specific requirements at runtime, a tool may obtain more freedom of action within the callback. (*End of advice to users.*)

Advice to implementors. A high-quality implementation will strive to set callback safety requirements to the most permissive level for a given callback invocation. (*End of advice to implementors.*)

All functions with the prefix MPI_T, except those listed in Table 15.6, may return the error code MPI_T_ERR_NOT_ACCESSIBLE to indicate that the user may not access this function at this time. The functions (and their respective PMPI versions) listed in Table 15.6 are exceptions to this rule and shall not return MPI_T_ERR_NOT_ACCESSIBLE.

Rationale. A call may be implemented in a way that is not safe for all execution contexts of a callback function, e.g., inside a signal handler. An MPI implementation therefore needs a way to communicate its inability to perform a certain action due to the execution context of a callback invocation. (*End of rationale.*)

Advice to implementors. A high-quality implementation shall not return MPI_T_ERR_NOT_ACCESSIBLE except where absolutely necessary. (*End of advice to implementors.*)

Advice to users. Users intercepting calls into the MPI tool information interface using the PMPI interface must ensure that the safety requirements for the calling context are met. This means that users may have to implement the wrapper with the highest safety level used by the MPI implementation. (*End of advice to users.*)

Event Type Query Functions

An MPI implementation exports a set of N event types through the MPI tool information interface. If N is zero, then the MPI implementation does not export any event types; otherwise, the provided event types are indexed from 0 to $N - 1$. This index number is used in subsequent calls to identify a specific event type.

An MPI implementation is allowed to increase the number of event types during the execution of an MPI process. However, MPI implementations are not allowed to change the index of an event type or to delete an event type once it has been made visible to the user (e.g., if new event types become available via dynamic loading of additional components in the MPI implementation).

The following function can be used to query the number of event types, *num_events*:

`MPI_T_EVENT_GET_NUM(num_events)`

OUT *num_events* returns number of event types (integer)

C binding

`int MPI_T_event_get_num(int *num_events)`

The function `MPI_T_EVENT_GET_INFO` provides access to additional information about a specific event type.

`MPI_T_EVENT_GET_INFO(event_index, name, name_len, verbosity, array_of_datatypes, array_of_displacements, num_elements, enumtype, info, desc, desc_len, bind)`

IN	<i>event_index</i>	index of the event type to be queried between 0 and <i>num_events</i> - 1 (integer)
OUT	<i>name</i>	buffer to return the string containing the name of the event type (string)
INOUT	<i>name_len</i>	length of the string and/or buffer for <i>name</i> (integer)
OUT	<i>verbosity</i>	verbosity level of this event type (integer)
OUT	<i>array_of_datatypes</i>	array of MPI basic datatypes used to encode the event data (array of handles)
OUT	<i>array_of_displacements</i>	array of byte displacements of the elements in the event buffer (array of non-negative integers)
INOUT	<i>num_elements</i>	length of <i>array_of_datatypes</i> and <i>array_of_displacements</i> arrays (non-negative integer)
OUT	<i>enumtype</i>	optional descriptor for enumeration information (handle)
OUT	<i>info</i>	optional info object (handle)
OUT	<i>desc</i>	buffer to return the string containing a description of the event type (string)
INOUT	<i>desc_len</i>	length of the string and/or buffer for <i>desc</i> (integer)

```

1      OUT      bind                type of MPI object to which an event of this type
2                                      must be bound (integer)
3

```

C binding

```

5  int MPI_T_event_get_info(int event_index, char *name, int *name_len,
6                          int *verbosity, MPI_Datatype array_of_datatypes[],
7                          MPI_Aint array_of_displacements[], int *num_elements,
8                          MPI_T_enum *enumtype, MPI_Info *info, char *desc, int *desc_len,
9                          int *bind)
10

```

11 After a successful call to `MPI_T_EVENT_GET_INFO` for a particular event type, sub-
12 sequent calls to this routine that query information about the same event type must return
13 the same information. If any `INOUT` or `OUT` argument to `MPI_T_EVENT_GET_INFO` is a
14 `NULL` pointer, the implementation will ignore the argument and not return a value for the
15 specific argument.

16 The arguments `name` and `name_len` are used to return the name of the event type as
17 described in Section 15.3.3. If completed successfully, the routine is required to return a
18 name of at least length one. The name of the event type must be unique with respect to
19 all other names for event types used by the MPI implementation.

20 The argument `verbosity` returns the verbosity level of the event type (see Section 15.3.1).

21 The argument `array_of_datatypes` returns an array of MPI datatype handles that de-
22 scribe the elements returned for an instance of the event type with index `event_index`. The
23 event data can either be queried element by element with `MPI_T_EVENT_READ` or copied
24 into a contiguous event buffer with `MPI_T_EVENT_COPY`. For the latter case, the argu-
25 ment `array_of_displacements` returns an array of byte displacements in the event buffer in
26 ascending order starting with zero.

27 The user is responsible for the memory allocation for the `array_of_datatypes` and
28 `array_of_displacements` arrays. The number of elements in each array is supplied by the user
29 in `num_elements`. If the number of elements used by the event type is larger than the value
30 of `num_elements` provided by the user, the number of datatype handles and displacements
31 returned in the corresponding arrays is truncated to the value of `num_elements` passed in
32 by the user. If the user passes the `NULL` pointer for `array_of_datatypes` or
33 `array_of_displacements`, the respective arguments are ignored. Unless the user passes the
34 `NULL` pointer for `num_elements`, the function returns the number of elements required for
35 this event type. If the user passes the `NULL` pointer for `num_elements`, the arguments
36 `num_elements`, `array_of_datatypes`, and `array_of_displacements` are ignored.

37 MPI can optionally return an enumeration identifier in the `enumtype` argument, de-
38 scribing the individual elements in the `array_of_datatypes` argument. Otherwise, `enumtype`
39 is set to `MPI_T_ENUM_NULL`. If the argument to `enumtype` provided by the user is the `NULL`
40 pointer, no enumeration type is returned.

41 MPI can optionally return an info object containing the default hints set for a regis-
42 tration handle for this event type. If the argument to `info` provided by the user is the `NULL`
43 pointer, this argument is ignored, otherwise an MPI implementation is required to return
44 all hints that are supported by the implementation for a registration handle for this event
45 type and have default values specified; any user-supplied hints that were not ignored by the
46 implementation; and any additional hints that were set by the implementation. If no such
47 hints exist, a handle to a newly created info object is returned that contains no key/value
48 pair. The user is responsible for freeing `info` via `MPI_INFO_FREE`.

The arguments `desc` and `desc_len` are used to return the description of the event type as described in Section 15.3.3. Returning a description is optional. If an MPI implementation does not return a description, the first character for `desc` must be set to the null character and `desc_len` must be set to one at the return from this function.

The parameter `bind` returns the type of the MPI object to which the event type must be bound or the value `MPI_T_BIND_NO_OBJECT` (see Section 15.3.2).

If an event type has an equivalent name across connected MPI processes, the following OUT parameters must be identical: `verbosity`, `array_of_datatypes`, `num_elements`, `enumtype`, and `bind`. The returned description must be equivalent. As the argument `array_of_displacements` is process dependent, it may differ across connected MPI processes.

This routine returns `MPI_SUCCESS` on success and returns `MPI_T_ERR_INVALID_INDEX` if `event_index` does not match a valid event type index provided by the implementation at the time of the call.

`MPI_T_EVENT_GET_INDEX(name, event_index)`

IN	<code>name</code>	name of the event type (string)
OUT	<code>event_index</code>	index of the event type (integer)

C binding

`int MPI_T_event_get_index(const char *name, int *event_index)`

`MPI_T_EVENT_GET_INDEX` returns the index of an event type identified by a known event type name. The name parameter is provided by the caller, and `event_index` is returned by the MPI implementation. The name parameter is a string terminated with a null character.

This routine returns `MPI_SUCCESS` on success and returns `MPI_T_ERR_INVALID_NAME` if `name` does not match the name of any event type provided by the implementation at the time of the call.

Rationale. This routine is provided to enable fast retrieval of an event index by a tool, assuming it knows the name of the event type for which it is looking. The number of event types exposed by the implementation can change over time, so it is not possible for the tool to simply iterate over the list of event types once at initialization. Although using MPI implementation specific event type names is not portable across MPI implementations, tool developers may choose to take this route for lower overhead at runtime because the tool will not have to iterate over the entire set of event types to find a specific one. (*End of rationale.*)

Handle Allocation and Deallocation

Before the MPI implementation calls a callback function on the occurrence of a specific event, the user needs to register a callback function to be called for that event type and obtain a handle of type `MPI_T_event_registration`.

```

1 MPI_T_EVENT_HANDLE_ALLOC(event_index, obj_handle, info, event_registration)
2   IN      event_index      index of event type for which the registration handle
3                               is to be allocated (integer)
4
5   IN      obj_handle       reference to a handle of the MPI object to which this
6                               event is supposed to be bound (pointer)
7
8   IN      info             info object (handle)
9
10  OUT     event_registration event registration (handle)

```

C binding

```

11 int MPI_T_event_handle_alloc(int event_index, void *obj_handle, MPI_Info info,
12                             MPI_T_event_registration *event_registration)
13

```

14 MPI_T_EVENT_HANDLE_ALLOC creates a **registration handle** for the event type
15 identified by `event_index`. Furthermore, if required by the event type, the registration
16 handle is bound to the object referred to by the argument `obj_handle`. The argument
17 `obj_handle` is ignored if the `MPI_T_EVENT_GET_INFO` call for this event type returned
18 `MPI_T_BIND_NO_OBJECT` in the argument `bind`. The user can pass hints for the handle al-
19 location to the MPI implementation via the `info` argument. The allocated event-registration
20 handle is returned in the argument `event_registration`.

```

21
22
23 MPI_T_EVENT_HANDLE_SET_INFO(event_registration, info)

```

```

24   INOUT   event_registration event registration (handle)
25
26   IN      info             info object (handle)

```

C binding

```

27
28 int MPI_T_event_handle_set_info(MPI_T_event_registration event_registration,
29                               MPI_Info info)
30

```

31 MPI_T_EVENT_HANDLE_SET_INFO updates the hints of the event-registration handle
32 associated with `event_registration` using the hints provided in `info`. A call to this procedure
33 has no effect on previously set or defaulted hints that are not specified by `info`. It also has
34 no effect on previously set or defaulted hints that are specified by `info`, but are ignored by
35 the MPI implementation in this call to `MPI_T_EVENT_HANDLE_SET_INFO`.

36
37 *Advice to users.* Some info items that an implementation can use when it creates
38 an event-registration handle cannot easily be changed once the registration handle
39 is created. Thus, an implementation may ignore hints issued in this call that it
40 would have accepted in a handle allocation call. An implementation may also be
41 unable to update certain info hints in a call to `MPI_T_EVENT_HANDLE_SET_INFO`.
42 `MPI_T_EVENT_HANDLE_GET_INFO` can be used to determine whether info changes
43 were ignored by the implementation. (*End of advice to users.*)

MPI_T_EVENT_HANDLE_GET_INFO(event_registration, info_used)			1
IN	event_registration	event registration (handle)	2
OUT	info_used	info object (handle)	3
			4
			5

C binding

```
int MPI_T_event_handle_get_info(MPI_T_event_registration event_registration,
                               MPI_Info *info_used)

```

MPI_T_EVENT_HANDLE_GET_INFO returns a new info object containing the hints of the event-registration handle associated with `event_registration`. The current setting of all hints related to this registration handle 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`.

MPI_T_EVENT_REGISTER_CALLBACK(event_registration, cb_safety, info, user_data,			19
event_cb_function)			20
INOUT	event_registration	event registration (handle)	21
IN	cb_safety	maximum callback safety level (integer)	22
IN	info	info object (handle)	23
IN	user_data	pointer to a user-controlled buffer	24
IN	event_cb_function	pointer to user-defined callback function (function)	25
			26
			27
			28

C binding

```
int MPI_T_event_register_callback(MPI_T_event_registration event_registration,
                                 MPI_T_cb_safety cb_safety, MPI_Info info, void *user_data,
                                 MPI_T_event_cb_function event_cb_function)

```

MPI_T_EVENT_REGISTER_CALLBACK associates a user-defined function pointed to by `event_cb_function` with an allocated event-registration handle. The maximum callback safety level supported by the callback function is passed in the argument `cb_safety`. The safety levels are defined in Table 15.5. A user can register multiple callback functions for a given event-registration handle, potentially specifying one for each callback safety level. Registering a callback function for a specific callback safety level overwrites any previously-registered callback function pointer and info object associated with the event registration for the specific callback safety level. If `event_cb_function` is the NULL pointer, an existing association of a callback function for that callback safety level is removed.

When an event is triggered, the implementation will select from all registered callbacks the callback with the lowest safety level valid in the context in which the callback is invoked. In situations where the required callback safety level exceeds the highest level for which a callback function is registered for a given registration handle, the event instance is dropped.

At callback invocation time, the implementation passes the pointer to a user-defined memory region specified during callback registration with the argument `user_data`.

1 The user can pass hints for the registration of the specified callback function to the
 2 MPI implementation via the info argument.

3
 4 *Advice to users.* As event instances can be raised as soon as the registration handle
 5 is associated with the first callback function, the callback function with the highest
 6 callback safety guarantees should be registered before any further registrations for
 7 lower callback safety guarantees, to avoid dropped events due to insufficient callback
 8 safety guarantees. (*End of advice to users.*)

9
 10 The callback function passed to MPI_T_EVENT_REGISTER_CALLBACK in the argu-
 11 ment event_cb_function needs to have the following type:

```
12 typedef void MPI_T_event_cb_function(MPI_T_event_instance event_instance,  
13                                     MPI_T_event_registration event_registration,  
14                                     MPI_T_cb_safety cb_safety, void *user_data);
```

15 The argument event_instance corresponds to a handle for the opaque event-instance
 16 object of type MPI_T_event_instance. This handle is only valid inside the corresponding
 17 invocation of the function to which it is passed. The argument event_registration corresponds
 18 to the event-registration handle returned by MPI_T_EVENT_HANDLE_ALLOC for the user
 19 function to the same event type and bound object combination. The handle can be used to
 20 identify the specific event registration information, such as event type and bound object, or
 21 even to deallocate the handle from within the callback invocation. The argument cb_safety
 22 describes the safety requirements the callback function must fulfill in the current invocation.
 23 The argument user_data is the pointer to user-allocated memory that was passed to the MPI
 24 implementation during callback registration.

```
25  
26  
27 MPI_T_EVENT_CALLBACK_SET_INFO(event_registration, cb_safety, info)
```

28	INOUT	event_registration	event registration (handle)
29	IN	cb_safety	callback safety level (integer)
30			
31	IN	info	info object (handle)

32 33 C binding

```
34 int MPI_T_event_callback_set_info(MPI_T_event_registration event_registration,  
35                                 MPI_T_cb_safety cb_safety, MPI_Info info)
```

36 MPI_T_EVENT_CALLBACK_SET_INFO updates the hints of the callback function reg-
 37 istered for the callback safety level specified by cb_safety of the event-registration handle
 38 associated with event_registration using the hints provided in info. A call to this procedure
 39 has no effect on previously set or defaulted hints that are not specified by info. It also has
 40 no effect on previously set or defaulted hints that are specified by info, but are ignored by
 41 the MPI implementation in this call to MPI_T_EVENT_CALLBACK_SET_INFO.

```
42  
43  
44 MPI_T_EVENT_CALLBACK_GET_INFO(event_registration, cb_safety, info_used)
```

45	IN	event_registration	event registration (handle)
46			
47	IN	cb_safety	callback safety level (integer)
48	OUT	info_used	info object (handle)

C binding

```
int MPI_T_event_callback_get_info(MPI_T_event_registration event_registration,
                                MPI_T_cb_safety cb_safety, MPI_Info *info_used)
```

MPI_T_EVENT_CALLBACK_GET_INFO returns a new info object containing the hints of the callback function registered for the callback safety level specified by `cb_safety` of the event-registration handle associated with `event_registration`. The current set of all hints related to this callback safety level of the event-registration handle 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`.

To stop the MPI implementation from raising events for a specific registration, a user needs to free the corresponding event-registration handle.

```
MPI_T_EVENT_HANDLE_FREE(event_registration, user_data, free_cb_function)
```

INOUT	<code>event_registration</code>	event registration (handle)
IN	<code>user_data</code>	pointer to a user-controlled buffer
IN	<code>free_cb_function</code>	pointer to user-defined callback function (function)

C binding

```
int MPI_T_event_handle_free(MPI_T_event_registration event_registration,
                            void *user_data, MPI_T_event_free_cb_function free_cb_function)
```

MPI_T_EVENT_HANDLE_FREE returns `MPI_SUCCESS` when deallocation of the handle was initiated successfully and returns `MPI_T_ERR_INVALID_HANDLE` if `event_registration` does not match a valid allocated event-registration handle at the time of the call. The callback function `free_cb_function` is called by the MPI implementation, when it is able to guarantee that no further event instances for the corresponding event-registration handle will be raised. If the pointer to `free_cb_function` is the `NULL` pointer, no user function is invoked after successful deallocation of the event registration handle. The pointer to user-controlled memory provided in the `user_data` argument will be passed to the function provided in the `free_cb_function` on invocation.

Advice to users. A free-callback function associated with a registration handle should always be prepared to postpone any pending actions, should the provided callback safety requirements exceed those required by the pending actions. (*End of advice to users.*)

The callback function passed to `MPI_T_EVENT_HANDLE_FREE` in the argument `free_cb_function` needs to have the following type:

```
typedef void MPI_T_event_free_cb_function(
    MPI_T_event_registration event_registration,
    MPI_T_cb_safety cb_safety, void *user_data);
```

Handling Dropped Events

Events may occur at times when the MPI implementation cannot invoke the user function corresponding to a matching event handle. An implementation is allowed to buffer such events and delay the callback invocation. If an event occurs at times when the corresponding callback function cannot be called and the corresponding data cannot be buffered, or no callback function meeting the required callback safety level is registered, the event data may be dropped. To discover such data loss, the user can set a handler function for a specific event-registration handle.

```
MPI_T_EVENT_SET_DROPPED_HANDLER(event_registration, dropped_cb_function)
```

INOUT	event_registration	valid event registration (handle)
IN	dropped_cb_function	pointer to user-defined callback function (function)

C binding

```
int MPI_T_event_set_dropped_handler(
    MPI_T_event_registration event_registration,
    MPI_T_event_dropped_cb_function dropped_cb_function)
```

`MPI_T_EVENT_SET_DROPPED_HANDLER` registers the function `dropped_cb_function` to be called by the MPI implementation when event information is dropped for the registration handle specified in `event_registration`. Subsequent calls to `MPI_T_EVENT_SET_DROPPED_HANDLER` with the same registration handle will replace previously-registered callback functions for that registration handle. If the pointer to `dropped_cb_function` is the `NULL` pointer, no data loss is recorded or reported until a new valid callback function is registered.

Advice to users. The invocation of the dropped handler callback function may not necessarily occur close to the time the event was actually lost. (*End of advice to users.*)

The callback function passed to `MPI_T_EVENT_SET_DROPPED_HANDLER` in the argument `dropped_cb_function` needs to have the following type:

```
typedef void MPI_T_event_dropped_cb_function(MPI_Count count,
    MPI_T_event_registration event_registration, int source_index,
    MPI_T_cb_safety cb_safety, void *user_data);
```

The argument `event_registration` corresponds to the event registration handle to which the dropped data corresponds. The argument `count` provides a best effort estimation of the number of invocations to a registered event callback corresponding to `event_registration` that were not executed since the registration of the dropped-callback handler or the last invocation of a registered dropped-callback handler. The `source_index` provides the index of the source that dropped the corresponding event information. The argument `cb_safety` describes the safety requirements the callback function must fulfill in the current invocation. The possible values for `cb_safety` are described in Table 15.5. The argument `user_data` is the pointer to user-allocated memory that was passed to the MPI implementation during callback registration. If no event callback is registered for safety requirement levels that an implementation uses to invoke the dropped handler callback function for a specific event, the corresponding dropped handler callback function will not be invoked.

Advice to users. A callback function for dropped events associated with a registration handle should always be prepared to postpone any pending actions, should the provided callback safety requirements exceed those required by the pending actions. (*End of advice to users.*)

Advice to implementors. A high-quality implementation should strive to find a good balance between timely notification, completeness of information, and the freedom of action for a tool when invoking the callback function for dropped events associated with a registration handle. (*End of advice to implementors.*)

If dropped event notifications have been observed for a specific source since the last event notification of that source, the corresponding dropped handler callback function must be called before other events are raised for that source. This means in a sequence of five events E1 to E5 from the same source, where E3 and E4 were dropped, any handler function set through `MPI_T_EVENT_SET_DROPPED_HANDLER` for event-registration handles associated with E3 or E4 must be called before E5 is raised.

Reading Event Data

In event callbacks, the parameter `event_instance` provides access to the per-instance event data, i.e., the data encoded by the specific event type for this instance. The user can obtain event data as well as event meta data, such as a time stamp and the source, by providing this handle to the respective query functions. The event-instance handle is invalid beyond the scope of the current invocation of the callback function to which it is provided.

The callback function argument `event_registration` identifies the registration handle that was used to register the callback function.

The callback function argument `cb_safety` indicates the requirements for the specific callback invocation. The value is one of the safety requirements levels described in Table 15.5. The argument `user_data` passes the pointer provided by the user during callback registration back to the function call.

Advice to users. Depending on the registered event and usage of MPI by the application, a callback function may be invoked with high frequency. Users should therefore strive to minimize the amount of work done inside callback functions. Furthermore, the time spent in a callback function may influence the capability of an implementation to buffer events; long execution times may lead to an increased number of dropped events. (*End of advice to users.*)

MPI provides the following function calls to access data of a specific event instance and its corresponding meta data (such as its time and source).

`MPI_T_EVENT_READ(event_instance, element_index, buffer)`

IN	<code>event_instance</code>	event-instance handle provided to the callback function (handle)
IN	<code>element_index</code>	index into the array of datatypes of the item to be queried (integer)
OUT	<code>buffer</code>	pointer to a memory location to store the item data (choice)

C binding

```
int MPI_T_event_read(MPI_T_event_instance event_instance, int element_index,
                    void *buffer)
```

MPI_T_EVENT_READ allows users to copy one element of the event data to a user-specified buffer at a time.

The `event_instance` argument identifies the event instance to query. It is erroneous to provide any other event-instance handle to the call than the one passed by the MPI implementation to the callback function in which the data is read. The `buffer` argument must point to a memory location the MPI implementation can copy the element of the event data to identified by `element_index`.

```
MPI_T_EVENT_COPY(event_instance, buffer)
```

IN	<code>event_instance</code>	event instance provided to the callback function (handle)
OUT	<code>buffer</code>	user-allocated buffer for event data (choice)

C binding

```
int MPI_T_event_copy(MPI_T_event_instance event_instance, void *buffer)
```

MPI_T_EVENT_COPY copies the event data as a whole into the user-provided `buffer`. The user must assure that the buffer is of at least the size of the extent of the event type, which can be computed from the type and displacement information returned by the corresponding call to `MPI_T_EVENT_GET_INFO`. The data may include padding bytes between individual elements of the event data in the buffer. A user can reconstruct the location and size of the data contained in the buffer through the information returned by `MPI_T_EVENT_GET_INFO`.

Advice to implementors. An implementation should strive to use an appropriately compact representation when copying event instance data to a user buffer via `MPI_T_EVENT_COPY` to reduce the amount of memory required for the user buffer. *(End of advice to implementors.)*

Reading Event Meta Data

Additional to the specific event data encoded by each event type, supplemental information available across all event types can be queried.

```
MPI_T_EVENT_GET_TIMESTAMP(event_instance, event_timestamp)
```

IN	<code>event_instance</code>	event instance provided to the callback function (handle)
OUT	<code>event_timestamp</code>	timestamp the event was observed (integer)

C binding

```
int MPI_T_event_get_timestamp(MPI_T_event_instance event_instance,
                             MPI_Count *event_timestamp)
```

MPI_T_EVENT_GET_TIMESTAMP returns the timestamp of when the event was initially observed by the implementation. The `event_instance` argument identifies the event instance to query. It is erroneous to provide any other handle to the call than the one passed by the MPI implementation to the callback function in which the timestamp is read.

Advice to users. An MPI implementation may postpone the call to the user's callback function. In this case, the call to MPI_T_EVENT_GET_TIMESTAMP may yield a timestamp in the past that is closer to the time the event was initially observed, as opposed to a timestamp captured during callback function invocation. (*End of advice to users.*)

Advice to implementors. A high-quality implementation will return a timestamp as close as possible to the earliest time the event was observed by the MPI implementation. (*End of advice to implementors.*)

An event may be raised from different components acting as event sources in the MPI implementation. A source in this context is an abstract concept that helps to define partial ordering of raised events, as each source provides its own ordering guarantees. A source describes the entity that raises the event, rather than the origin of the data.

To identify the source of an event instance, the user can query the index of the source within the corresponding event callback function invocation.

Advice to implementors. An excessive number of event sources may negatively impact performance of a tool due to per-source overhead in event handling. (*End of advice to implementors.*)

MPI_T_EVENT_GET_SOURCE(`event_instance`, `source_index`)

IN	<code>event_instance</code>	event instance provided to the callback function (handle)
OUT	<code>source_index</code>	index identifying the source (integer)

C binding

```
int MPI_T_event_get_source(MPI_T_event_instance event_instance,
                          int *source_index)
```

The `event_instance` argument identifies the event instance to query. It is erroneous to provide any other event-instance handle to the call than the one passed by the MPI implementation to the callback function in which the source is queried.

The `source_index` argument returns the index of the source of the event instance. It can be used to query more information on the source using MPI_T_SOURCE_GET_INFO.

Rationale. Event callback function invocations are associated with a source to enable chronological processing of events on the tool side, when required, while retaining low overhead on the side of the MPI implementation. (*End of rationale.*)

15.3.9 Variable Categorization

MPI implementations can optionally group performance and control variables into categories to express logical relationships between various variables. For example, an MPI implementation could group all control and performance variables that refer to message transfers in the MPI implementation and thereby distinguish them from variables that refer to local resources such as memory allocations or other interactions with the operating system.

Categories can also contain other categories to form a hierarchical grouping. Categories can never include themselves, either directly or transitively within other included categories. Expanding on the example above, this allows MPI to refine the grouping of variables referring to message transfers into variables to control and to monitor message queues, message matching activities and communication protocols. Each of these groups of variables would be represented by a separate category and these categories would then be listed in a single category representing variables for message transfers.

The category information may be queried in a fashion similar to the mechanism for querying variable information. The MPI implementation exports a set of N categories via the MPI tool information interface. If $N = 0$, then the MPI implementation does not export any categories, otherwise the provided categories are indexed from 0 to $N - 1$. This index number is used in subsequent calls to functions of the MPI tool information interface to identify the individual categories.

An MPI implementation is permitted to increase the number of categories during the execution of an MPI program when new categories become available through dynamic loading. However, MPI implementations are not allowed to change the index of a category or delete it once it has been added to the set.

Similarly, MPI implementations are allowed to add variables to categories, but they are not allowed to remove variables from categories or change the order in which they are returned.

Category Query Functions

The following function can be used to query the number of categories, `num_cat`.

```
MPI_T_CATEGORY_GET_NUM(num_cat)
```

OUT	<code>num_cat</code>	current number of categories (integer)
-----	----------------------	--

C binding

```
int MPI_T_category_get_num(int *num_cat)
```

Individual category information can then be queried by calling the following function:

```
MPI_T_CATEGORY_GET_INFO(cat_index, name, name_len, desc, desc_len, num_cvars,
                        num_pvars, num_categories)
```

IN	<code>cat_index</code>	index of the category to be queried (integer)
OUT	<code>name</code>	buffer to return the string containing the name of the category (string)
INOUT	<code>name_len</code>	length of the string and/or buffer for name (integer)

OUT	desc	buffer to return the string containing the description of the category (string)	1
			2
INOUT	desc_len	length of the string and/or buffer for desc (integer)	3
			4
OUT	num_cvars	number of control variables in the category (integer)	5
OUT	num_pvars	number of performance variables in the category (integer)	6
			7
OUT	num_categories	number of categories contained in the category (integer)	8
			9
			10

C binding

```
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)
```

The arguments `name` and `name_len` are used to return the name of the category as described in Section 15.3.3.

The routine is required to return a name of at least length one. This name must be unique with respect to all other names for categories used by the MPI implementation.

If any OUT parameter to `MPI_T_CATEGORY_GET_INFO` is the NULL pointer, the implementation will ignore the parameter and not return a value for the parameter.

The arguments `desc` and `desc_len` are used to return the description of the category as described in Section 15.3.3.

Returning a description is optional. If an MPI implementation decides not to return a description, the first character for `desc` must be set to the null character and `desc_len` must be set to one at the return of this call.

The function returns the number of control variables, performance variables and other categories contained in the queried category in the arguments `num_cvars`, `num_pvars`, and `num_categories`, respectively.

If the name of a category is equivalent across connected MPI processes, then the returned description must be equivalent.

```
MPI_T_CATEGORY_GET_NUM_EVENTS(cat_index, num_events)
```

IN	cat_index	index of the category to be queried (integer)	35
OUT	num_events	number of event types in the category (integer)	36

C binding

```
int MPI_T_category_get_num_events(int cat_index, int *num_events)
```

`MPI_T_CATEGORY_GET_NUM_EVENTS` returns the number of event types contained in the queried category.

```
MPI_T_CATEGORY_GET_INDEX(name, cat_index)
```

IN	name	the name of the category (string)	46
OUT	cat_index	the index of the category (integer)	47

C binding

```
int MPI_T_category_get_index(const char *name, int *cat_index)
```

`MPI_T_CATEGORY_GET_INDEX` is a function for retrieving the index of a category given a known category name. The `name` parameter is provided by the caller, and `cat_index` is returned by the MPI implementation. The `name` parameter is a string terminated with a null character.

This routine returns `MPI_SUCCESS` on success and returns `MPI_T_ERR_INVALID_NAME` if `name` does not match the name of any category provided by the implementation at the time of the call.

Rationale. This routine is provided to enable fast retrieval of a category index by a tool, assuming it knows the name of the category for which it is looking. The number of categories exposed by the implementation can change over time, so it is not possible for the tool to simply iterate over the list of categories once at initialization. Although using MPI implementation specific category names is not portable across MPI implementations, tool developers may choose to take this route for lower overhead at runtime because the tool will not have to iterate over the entire set of categories to find a specific one. (*End of rationale.*)

Category Member Query Functions

```
MPI_T_CATEGORY_GET_CVARS(cat_index, len, indices)
```

IN	cat_index	index of the category to be queried, in the range from 0 to <code>num_cat - 1</code> (integer)
IN	len	the length of the indices array (integer)
OUT	indices	an integer array of size <code>len</code> , indicating control variable indices (array of integers)

C binding

```
int MPI_T_category_get_cvars(int cat_index, int len, int indices[])
```

`MPI_T_CATEGORY_GET_CVARS` can be used to query which control variables are contained in a particular category. A category contains zero or more control variables.

```
MPI_T_CATEGORY_GET_PVARS(cat_index, len, indices)
```

IN	cat_index	index of the category to be queried, in the range from 0 to <code>num_cat - 1</code> (integer)
IN	len	the length of the indices array (integer)
OUT	indices	an integer array of size <code>len</code> , indicating performance variable indices (array of integers)

C binding

```
int MPI_T_category_get_pvars(int cat_index, int len, int indices[])
```

MPI_T_CATEGORY_GET_PVARS can be used to query which performance variables are contained in a particular category. A category contains zero or more performance variables.

MPI_T_CATEGORY_GET_EVENTS(cat_index, len, indices)

IN	cat_index	index of the category to be queried, in the range from 0 to num_cat - 1 (integer)
IN	len	the length of the indices array (integer)
OUT	indices	an integer array of size len, indicating event type indices (array of integers)

C binding

```
int MPI_T_category_get_events(int cat_index, int len, int indices[])
```

MPI_T_CATEGORY_GET_EVENTS can be used to query which event types are contained in a particular category. A category contains zero or more event types.

MPI_T_CATEGORY_GET_CATEGORIES(cat_index, len, indices)

IN	cat_index	index of the category to be queried, in the range from 0 to num_cat - 1 (integer)
IN	len	the length of the indices array (integer)
OUT	indices	an integer array of size len, indicating category indices (array of integers)

C binding

```
int MPI_T_category_get_categories(int cat_index, int len, int indices[])
```

MPI_T_CATEGORY_GET_CATEGORIES can be used to query which other categories are contained in a particular category. A category contains zero or more other categories.

As mentioned above, MPI implementations can grow the number of categories as well as the number of variables or other categories within a category. In order to allow users of the MPI tool information interface to check quickly whether new categories have been added or new variables or categories have been added to a category, MPI maintains an update number that is monotonically increasing during the execution and is returned by the following function:

MPI_T_CATEGORY_CHANGED(update_number)

OUT	update_number	update number (integer)
-----	---------------	-------------------------

C binding

```
int MPI_T_category_changed(int *update_number)
```

If two calls to this routine return the same update number, it is guaranteed that the category information has not changed between the two calls. If the update number retrieved

1 from the second call is higher, then some categories have been added or expanded.

2 The index values returned in indices by `MPI_T_CATEGORY_GET_CVARS`,
3 `MPI_T_CATEGORY_GET_PVARS`, `MPI_T_CATEGORY_GET_EVENTS`, and
4 `MPI_T_CATEGORY_GET_CATEGORIES` can be used as input to
5 `MPI_T_CVAR_GET_INFO`, `MPI_T_PVAR_GET_INFO`, `MPI_T_EVENT_GET_INFO`, and
6 `MPI_T_CATEGORY_GET_INFO`, respectively.

7 The user is responsible for allocating the arrays passed into the functions
8 `MPI_T_CATEGORY_GET_CVARS`, `MPI_T_CATEGORY_GET_PVARS`,
9 `MPI_T_CATEGORY_GET_EVENTS`, and `MPI_T_CATEGORY_GET_CATEGORIES`. Starting
10 from array index 0, each function writes up to `len` elements into the array. If the category
11 contains more than `len` elements, the function returns an arbitrary subset of size `len`. Oth-
12 erwise, the entire set of elements is returned in the beginning entries of the array, and any
13 remaining array entries are not modified.

14 15.3.10 Return Codes for the MPI Tool Information Interface

15 All functions defined as part of the MPI tool information interface return an integer error
16 code (see Table 15.7) to indicate whether the function was completed successfully or was
17 aborted. In the latter case, the error code indicates the reason for not completing the
18 routine. Such errors neither impact the execution of the MPI process nor invoke MPI error
19 handlers. The MPI process continues executing regardless of the return code from the
20 call. The MPI implementation is not required to check all user-provided parameters; if a
21 user passes invalid parameter values to any routine the behavior of the implementation is
22 undefined.

23 All error codes with the prefix `MPI_T_` must be unique values and cannot overlap with
24 any other error codes or error classes returned by the MPI implementation. Further, they
25 shall be treated as MPI error classes as defined in Section 9.4 and follow the same rules and
26 restrictions. In particular, they must satisfy:

$$27 \quad 0 = \text{MPI_SUCCESS} < \text{MPI_T_ERR_XXX} \leq \text{MPI_ERR_LASTCODE}.$$

28 15.3.11 Profiling Interface

29 All requirements for the profiling interfaces, as described in Section 15.2, also apply to
30 the MPI tool information interface. All rules, guidelines, and recommendations from Sec-
31 tion 15.2 apply equally to calls defined as part of the MPI tool information interface.

Table 15.7: Return codes used in functions of the MPI tool information interface

Return Code	Description
Return Codes for All Functions in the MPI Tool Information Interface	
MPI_SUCCESS	Call completed successfully
MPI_T_ERR_INVALID	Invalid or bad parameter value(s)
MPI_T_ERR_MEMORY	Out of memory
MPI_T_ERR_NOT_INITIALIZED	Interface not initialized
MPI_T_ERR_CANNOT_INIT	Interface not in the state to be initialized
MPI_T_ERR_NOT_ACCESSIBLE	Requested functionality not accessible
Return Codes for Datatype Functions: MPI_T_ENUM_*	
MPI_T_ERR_INVALID_INDEX	The enumeration index is invalid
Return Codes for Variable, Category, and Event Query Functions: MPI_T_*_GET_*	
MPI_T_ERR_INVALID_INDEX	The variable or category index is invalid
MPI_T_ERR_INVALID_NAME	The variable or category name is invalid
Return Codes for Handle Functions: MPI_T_*_{ALLOC FREE}	
MPI_T_ERR_INVALID_INDEX	The variable index is invalid
MPI_T_ERR_INVALID_HANDLE	The handle is invalid
MPI_T_ERR_OUT_OF_HANDLES	No more handles available
Return Codes for Performance Experiment Session Functions: MPI_T_PVAR_SESSION_*	
MPI_T_ERR_OUT_OF_SESSIONS	No more sessions available
MPI_T_ERR_INVALID_SESSION	Session argument is not a valid session
Return Codes for Control Variable Access Functions: MPI_T_CVAR_{READ WRITE}	
MPI_T_ERR_CVAR_SET_NOT_NOW	Variable cannot be set at this moment
MPI_T_ERR_CVAR_SET_NEVER	Variable cannot be set until end of execution
MPI_T_ERR_INVALID_HANDLE	The handle is invalid
Return Codes for Performance Variable Access and Control: MPI_T_PVAR_{START STOP READ WRITE RESET READREST}	
MPI_T_ERR_INVALID_HANDLE	The handle is invalid
MPI_T_ERR_INVALID_SESSION	Performance experiment session argument is not valid
MPI_T_ERR_PVAR_NO_STARTSTOP	Variable cannot be started or stopped (for MPI_T_PVAR_START and MPI_T_PVAR_STOP)
MPI_T_ERR_PVAR_NO_WRITE	Variable cannot be written or reset (for MPI_T_PVAR_WRITE and MPI_T_PVAR_RESET)
MPI_T_ERR_PVAR_NO_ATOMIC	Variable cannot be read and written atomically (for MPI_T_PVAR_READRESET)
Return Codes for Source Functions: MPI_T_SOURCE_*	
MPI_T_ERR_INVALID_INDEX	The source index is invalid
MPI_T_ERR_NOT_SUPPORTED	Requested functionality not supported
Return Codes for Category Functions: MPI_T_CATEGORY_*	
MPI_T_ERR_INVALID_INDEX	The category index is invalid

DRAFT

Chapter 16

Deprecated Interfaces

16.1 Deprecated since MPI-2.0

- The following function is deprecated and is superseded by `MPI_COMM_CREATE_KEYVAL` in MPI-2.0. The language independent definition of the deprecated function is the same as that of the new function, except for the function name and a different behavior in the C/Fortran language interoperability, see Section 19.3.7. The language bindings are modified.

`MPI_KEYVAL_CREATE(copy_fn, delete_fn, keyval, extra_state)`

IN	<code>copy_fn</code>	Copy callback function for keyval
IN	<code>delete_fn</code>	Delete callback function for keyval
OUT	<code>keyval</code>	key value for future access (integer)
IN	<code>extra_state</code>	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)
```

For this routine, an interface within the `mpi_f08` module was never defined.

Fortran binding

```
MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)  
EXTERNAL COPY_FN, DELETE_FN  
INTEGER KEYVAL, EXTRA_STATE, IERROR
```

The `copy_fn` function is invoked when a communicator is duplicated by `MPI_COMM_DUP`. `copy_fn` should be of type `MPI_Copy_function`, which is defined as follows:

```
typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval,  
void *extra_state, void *attribute_val_in, void *attribute_val_out,  
int *flag);
```

A Fortran declaration for such a function is as follows:

For this routine, an interface within the `mpi_f08` module was never defined.

```

1  SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
2  ATTRIBUTE_VAL_OUT, FLAG, IERR)
3      INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
4      ATTRIBUTE_VAL_OUT, IERR
5      LOGICAL FLAG
6

```

7 `copy_fn` may be specified as `MPI_NULL_COPY_FN` or `MPI_DUP_FN` from either C
8 or Fortran; `MPI_NULL_COPY_FN` is a function that does nothing other than re-
9 turn `flag = 0` and `MPI_SUCCESS`. `MPI_DUP_FN` is a simple-minded copy function that
10 sets `flag = 1`, returns the value of `attribute_val_in` in `attribute_val_out`, and returns
11 `MPI_SUCCESS`. Note that `MPI_NULL_COPY_FN` and `MPI_DUP_FN` are also depre-
12 cated.

13 Analogous to `copy_fn` is a callback deletion function, defined as follows. The
14 `delete_fn` function is invoked when a communicator is deleted by `MPI_COMM_FREE`
15 or when a call is made explicitly to `MPI_ATTR_DELETE`. `delete_fn` should be of type
16 `MPI_Delete_function`, which is defined as follows:

```

17 typedef int MPI_Delete_function(MPI_Comm comm, int keyval,
18 void *attribute_val, void *extra_state);
19

```

20 A Fortran declaration for such a function is as follows:

21 For this routine, an interface within the `mpi_f08` module was never defined.

```

22
23
24 SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)
25     INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR
26

```

27 `delete_fn` may be specified as `MPI_NULL_DELETE_FN` from either C or Fortran;
28 `MPI_NULL_DELETE_FN` is a function that does nothing other than return
29 `MPI_SUCCESS`. Note that `MPI_NULL_DELETE_FN` is also deprecated.

- 30
31 • The following function is deprecated and is superseded by
32 `MPI_COMM_FREE_KEYVAL` in MPI-2.0. The language independent definition of the
33 deprecated function is the same as the new function, except for the function name.
34 The language bindings are modified.
35

```

36
37 MPI_KEYVAL_FREE(keyval)
38

```

```

39     INOUT    keyval                Frees the integer key value (integer)
40

```

41 C binding

```

42 int MPI_Keyval_free(int *keyval)
43

```

44 For this routine, an interface within the `mpi_f08` module was never defined.

45 Fortran binding

```

46
47 MPI_KEYVAL_FREE(KEYVAL, IERROR)
48

```


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 the new function, except for the function name. The language bindings are modified.

`MPI_ATTR_PUT(comm, keyval, attribute_val)`

INOUT	<code>comm</code>	communicator to which attribute will be attached (handle)
IN	<code>keyval</code>	key value, as returned by <code>MPI_KEYVAL_CREATE</code> (integer)
IN	<code>attribute_val</code>	attribute value

C binding

```
int MPI_Attr_put(MPI_Comm comm, int keyval, void *attribute_val)
```

For this routine, an interface within the `mpi_f08` module was never defined.

Fortran 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 the new function, except for the function name. The language bindings are modified.

`MPI_ATTR_GET(comm, keyval, attribute_val, flag)`

IN	<code>comm</code>	communicator to which attribute is attached (handle)
IN	<code>keyval</code>	key value (integer)
OUT	<code>attribute_val</code>	attribute value, unless <code>flag = false</code>
OUT	<code>flag</code>	true if an attribute value was extracted; false if no attribute is associated with the key

C binding

```
int MPI_Attr_get(MPI_Comm comm, int keyval, void *attribute_val, int *flag)
```

1 For this routine, an interface within the `mpi_f08` module was never defined.

3 Fortran binding

4 `MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)`
 5 `INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR`
 6 `LOGICAL FLAG`

- 8
9
10 • The following function is deprecated and is superseded by
 11 `MPI_COMM_DELETE_ATTR` in MPI-2.0. The language independent definition of the
 12 deprecated function is the same as the new function, except for the function name.
 13 The language bindings are modified.

14
15
16 `MPI_ATTR_DELETE(comm, keyval)`

17 `INOUT comm` communicator to which attribute is attached
 18 (handle)
 19 `IN keyval` The key value of the deleted attribute (integer)

22 C binding

23 `int MPI_Attr_delete(MPI_Comm comm, int keyval)`

24
25 For this routine, an interface within the `mpi_f08` module was never defined.

27 Fortran binding

28 `MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)`
 29 `INTEGER COMM, KEYVAL, IERROR`

32 16.2 Deprecated since MPI-2.2

- 33
34 • The entire set of C++ language bindings was deprecated as of MPI-2.2 and removed
 35 in MPI-3.0. See Chapter 17, [Removed Interfaces](#) for more information.
 36
 37
 38 • The following function typedefs have been deprecated and are superseded by new
 39 names. Other than the typedef names, the function signatures are exactly the same;
 40 the names were updated to match conventions of other function typedef names.

41 <u>Deprecated Name</u>	42 <u>New Name</u>
43 <code>MPI_Comm_errhandler_fn</code>	<code>MPI_Comm_errhandler_function</code>
44 <code>MPI_File_errhandler_fn</code>	<code>MPI_File_errhandler_function</code>
45 <code>MPI_Win_errhandler_fn</code>	<code>MPI_Win_errhandler_function</code>

16.3 Deprecated since MPI-4.0

- Cancelling a send request by calling `MPI_CANCEL` has been deprecated and may be removed in a future version of the MPI specification.
- The following function is deprecated and is superseded by the new `MPI_INFO_GET_STRING` call in MPI-4.0.

`MPI_INFO_GET`(info, key, valuelen, value, flag)

IN	info	info object (handle)
IN	key	key (string)
IN	valuelen	length of value associated with key (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)
```

Fortran 2008 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=valuelen), INTENT(OUT) :: value
  LOGICAL, INTENT(OUT) :: flag
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran 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.

The function `MPI_INFO_GET` is allowed to be called at any time, following the description for MPI functionality that is always available in Section 11.4.1.

- The following function is deprecated and is superseded by the new `MPI_INFO_GET_STRING` call in MPI-4.0.

`MPI_INFO_GET_VALUELEN`(info, key, valuelen, flag)

IN	info	info object (handle)
IN	key	key (string)
OUT	valuelen	length of value associated with key (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)
```

Fortran 2008 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
```

Fortran 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.

The function `MPI_INFO_GET_VALUELEN` is allowed to be called at any time, following the description for MPI functionality that is always available in Section 11.4.1.

- The following return code has been deprecated and is superseded by a new name in MPI-4.0.

Deprecated Name	Replacement Name
MPI_T_ERR_INVALID_ITEM	MPI_T_ERR_INVALID_INDEX

- The following Fortran subroutines are deprecated because the Fortran language `storage_size()` and `c_sizeof()` intrinsic functions provide similar functionality. Note that while `MPI_SIZEOF` and `c_sizeof()` return the size in bytes, `storage_size()` provides the size in bits.

`MPI_SIZEOF(x, size)`

IN	x	a Fortran variable of numeric intrinsic type (choice)
OUT	size	size of machine representation of that type (integer)

Fortran 2008 binding

```
MPI_Sizeof(x, size, ierror)
  TYPE(*), DIMENSION(..) :: x
  INTEGER, INTENT(OUT) :: size
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran 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.*)

DRAFT

Chapter 17

Removed Interfaces

17.1 Removed MPI-1 Bindings

17.1.1 Overview

The following MPI-1 bindings were deprecated as of MPI-2 and are removed in MPI-3. They may be provided by an implementation for backwards compatibility, but are not required. Removal of these bindings affects all language-specific definitions thereof. Only the language-neutral bindings are listed when possible.

17.1.2 Removed MPI-1 Functions

Table 17.1 shows the removed MPI-1 functions and their replacements.

Table 17.1: Removed MPI-1 functions and their replacements

Removed	MPI-2 Replacement
MPI_ADDRESS	MPI_GET_ADDRESS
MPI_ERRHANDLER_CREATE	MPI_COMM_CREATE_ERRHANDLER
MPI_ERRHANDLER_GET	MPI_COMM_GET_ERRHANDLER
MPI_ERRHANDLER_SET	MPI_COMM_SET_ERRHANDLER
MPI_TYPE_EXTENT	MPI_TYPE_GET_EXTENT
MPI_TYPE_HINDEXED	MPI_TYPE_CREATE_HINDEXED
MPI_TYPE_HVECTOR	MPI_TYPE_CREATE_HVECTOR
MPI_TYPE_LB	MPI_TYPE_GET_EXTENT
MPI_TYPE_STRUCT	MPI_TYPE_CREATE_STRUCT
MPI_TYPE_UB	MPI_TYPE_GET_EXTENT

17.1.3 Removed MPI-1 Datatypes

Table 17.2 shows the removed MPI-1 datatypes and their replacements.

17.1.4 Removed MPI-1 Constants

Table 17.3 shows the removed MPI-1 constants. There are no replacements.

17.1.5 Removed MPI-1 Callback Prototypes

Table 17.4 shows the removed MPI-1 callback prototypes and their replacements.

Table 17.2: Removed MPI-1 datatypes. The indicated routine may be used for changing the lower and upper bound respectively.

Removed	MPI-2 Replacement
MPI_LB	MPI_TYPE_CREATE_RESIZED
MPI_UB	MPI_TYPE_CREATE_RESIZED

Table 17.3: Removed MPI-1 constants

Removed MPI-1 Constants
C type: <code>const int</code> (or unnamed enum)
Fortran type: <code>INTEGER</code>
MPI_COMBINER_HINDEXED_INTEGER
MPI_COMBINER_HVECTOR_INTEGER
MPI_COMBINER_STRUCT_INTEGER

Table 17.4: Removed MPI-1 callback prototypes and their replacements

Removed	MPI-2 Replacement
<code>MPI_Handler_function</code>	<code>MPI_Comm_errhandler_function</code>

17.2 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 18

Semantic Changes and Warnings

This chapter lists semantic changes that have been introduced into the MPI Standard as well as warnings that could potentially impact program behavior. In addition to those listed here, Chapter 17 also lists changes and backward incompatibilities caused by removing interfaces. Unlike Chapter 17, the changes in this chapter did not go through a deprecation process.

18.1 Semantic Changes

This section describes semantics that have changed in a way that would potentially cause an MPI program to behave differently when using this version of the MPI Standard without changing the program's code.

18.1.1 Semantic Changes Starting in MPI-4.0

- `MPI_COMM_DUP` and `MPI_COMM_IDUP` no longer propagate info hints from the input communicator to the output communicator. This behavior can be achieved using `MPI_COMM_DUP_WITH_INFO` and `MPI_COMM_IDUP_WITH_INFO`.
- The default communicator where errors are raised when not involving a communicator, window, or file was changed from `MPI_COMM_WORLD` to `MPI_COMM_SELF`.

18.2 Additional Warnings

This section describes additional changes that could potentially cause a program that relies on the semantics described in a previous version of the MPI Standard to behave differently than with this version of MPI. The changes in this section are limited in scope and unlikely to impact most programs.

18.2.1 Warnings Starting in MPI-4.0

The limit for length of MPI identifiers was removed. Prior to MPI-4.0, MPI identifiers were limited to 30 characters (31 with the profiling interface). This limitation was initially introduced to avoid exceeding the limit on some compilation systems.

Rationale. For Fortran, this limit was already relaxed for the Fortran specific function names, see Section 19.1.5, and the Fortran language specification 2003 requires support for a minimum of 63 characters for internal and external identifiers. Starting with the ISO/IEC 9899:1999 C programming language standard, support for a minimum of 63

1 characters is required for internal identifiers, but only 31 characters are required to
2 be significant for external identifiers. At the time of the release of MPI-4.0, most or
3 nearly all compilers allow external identifiers longer than 31 characters. Therefore,
4 the restriction is removed. (*End of rationale.*)

5
6 *Advice to users.* This affects users only if they store MPI identifiers into fixed sized
7 strings. (*End of advice to users.*)
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

DRAFT

Chapter 19

Language Bindings

19.1 Support for Fortran

19.1.1 Overview

The Fortran MPI language bindings have been designed to be compatible with the Fortran 90 standard with additional features from Fortran 2003 and Fortran 2008 [45] with TS 29113 [46].

Rationale. Fortran 90 contains numerous features designed to make it a more “modern” language than Fortran 77. It seems natural that MPI should be able to take advantage of these new features with a set of bindings tailored to Fortran 90. In Fortran 2008 with TS 29113, the major new language features used are the ASYNCHRONOUS attribute to protect nonblocking MPI operations, and assumed-type and assumed-rank dummy arguments for choice buffer arguments. Further requirements for compiler support are listed in Section 19.1.7. (*End of rationale.*)

MPI defines three methods of Fortran support:

1. **USE mpi_f08:** This method is described in Section 19.1.2. It requires compile-time argument checking with unique MPI handle types and provides techniques to fully solve the optimization problems with nonblocking calls. This is the only Fortran support method that is consistent with the Fortran standard (Fortran 2008 with TS 29113 and later). This method is highly recommended for all MPI applications.
2. **USE mpi:** This method is described in Section 19.1.3 and requires compile-time argument checking. Handles are defined as INTEGER. This Fortran support method is inconsistent with the Fortran standard, and its use is therefore not recommended. It exists only for backwards compatibility.
3. **INCLUDE 'mpif.h':** This method is described in Section 19.1.4. The use of the include file `mpif.h` is strongly discouraged starting with MPI-3.0, because this method neither guarantees compile-time argument checking nor provides sufficient techniques to solve the optimization problems with nonblocking calls, and is therefore inconsistent with the Fortran standard. It exists only for backwards compatibility with legacy MPI applications.

MPI 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 19.1.6 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 that `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 [46]; 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 19.1.2 through 19.1.4 define the Fortran support methods. The Fortran interfaces of each MPI routine are shorthands. Section 19.1.5 defines the corresponding full interface specification together with the specific procedure names and implications for the profiling interface. Section 19.1.6 describes the implementation of the MPI routines for different versions of the Fortran standard. Section 19.1.7 summarizes major requirements for MPI implementations with Fortran support. Section 19.1.8 and Section 19.1.9 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_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 that are specified using `KIND` type parameters. Sections 19.1.10 through 19.1.19 give an overview and details on known problems when using Fortran together with MPI; Section 19.1.20 compares the Fortran problems with those in C.

19.1.2 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 19.1.6 describes restrictions if the compiler does not support all the needed features. Within all MPI function specifications, the first of the set of two Fortran routine interface specifications is provided by this module. This module must:

- Define all named MPI constants.

- Declare MPI functions that return a value. 1
- Provide explicit interfaces according to the Fortran routine interface specifications. 2
This module therefore guarantees compile-time argument checking for all arguments 3
that are not `TYPE(*)`, with the following exception: 4
5

Only one Fortran interface is defined for functions that are deprecated as of 6
MPI-3.0. This interface must be provided as an explicit interface according to 7
the rules defined for the `mpi` module, see Section 19.1.3. 8

Advice to users. It is strongly recommended that developers substitute 9
calls to deprecated routines when upgrading from `mpif.h` or the `mpi` module 10
to the `mpi_f08` module. (*End of advice to users.*) 11
12

- Define the derived type `MPI_Status`, and define all MPI handles with uniquely named 13
handle types (instead of `INTEGER` handles, as in the `mpi` module). This is reflected in 14
the first Fortran binding in each MPI function definition throughout this document 15
(except for the deprecated routines). 16
- Overload the operators `.EQ.` and `.NE.` to allow the comparison of these MPI handles 17
with `.EQ.`, `.NE.`, `==` and `/=`. 18
19
- Use the `ASYNCHRONOUS` attribute to protect the buffers of nonblocking operations, 20
and set the `LOGICAL` compile-time constant `MPI_ASYNC_PROTECTS_NONBLOCKING` 21
to `.TRUE.` if the underlying Fortran compiler supports the `ASYNCHRONOUS` attribute 22
for MPI communication (as part of TS 29113). See Section 19.1.6 for older compiler 23
versions. 24
25
- Set the `LOGICAL` compile-time constant `MPI_SUBARRAYS_SUPPORTED` to `.TRUE.` and 26
declare choice buffers using the Fortran 2008 TS 29113 features `assumed-type` and 27
`assumed-rank`, i.e., `TYPE(*)`, `DIMENSION(...)` in all nonblocking, split collective and 28
persistent communication routines, if the underlying Fortran compiler supports it. 29
With this, noncontiguous sub-arrays can be used as buffers in nonblocking routines. 30

Rationale. In all blocking routines, i.e., if the choice-buffer is not declared as 31
`ASYNCHRONOUS`, the TS 29113 feature is not needed for the support of noncontigu- 32
ous buffers because the compiler can pass the buffer by in-and-out-copy through 33
a contiguous scratch array. (*End of rationale.*) 34
35

- Set the `MPI_SUBARRAYS_SUPPORTED` compile-time constant to `.FALSE.` and declare 36
choice buffers with a compiler-dependent mechanism that overrides type checking if 37
the underlying Fortran compiler does not support the Fortran 2008 TS 29113 `assumed-` 38
`type` and `assumed-rank` notation. In this case, the use of noncontiguous sub-arrays as 39
buffers in nonblocking calls may be invalid. See Section 19.1.6 for details. 40
41
- Declare each argument with an `INTENT` of `IN`, `OUT`, or `INOUT` as defined in this standard. 42
43

Rationale. For these definitions in the `mpi_f08` bindings, in most cases, `INTENT(IN)` 44
is used if the C interface uses call-by-value. For all buffer arguments and for `OUT` and 45
`INOUT` dummy arguments that allow one of the nonordinary Fortran constants (see 46
`MPI_BOTTOM`, etc. in Section 2.5.4) as input, an `INTENT` is not specified. (*End of* 47
rationale.) 48

Advice to users. If a dummy argument is declared with `INTENT(OUT)`, then the Fortran standard stipulates that the actual argument becomes undefined upon invocation of the MPI routine, i.e., it may be overwritten by some other values, e.g. zeros; according to [45], 12.5.2.4 Ordinary dummy variables, Paragraph 17: “If a dummy argument has `INTENT(OUT)`, the actual argument becomes undefined at the time the association is established, except [...]”. For example, if the dummy argument is an assumed-size array and the actual argument is a strided array, the call may be implemented with copy-in and copy-out of the argument. In the case of `INTENT(OUT)` the copy-in may be suppressed by the optimization and the routine starts execution using an array of undefined values. If the routine stores fewer elements into the dummy argument than is provided in the actual argument, then the remaining locations are overwritten with these undefined values. See also both advices to implementors in Section 19.1.3. (*End of advice to users.*)

- Declare all ierror output arguments as `OPTIONAL`, except for user-defined callback functions (e.g., of type `MPI_Comm_copy_attr_function` or `COMM_COPY_ATTR_FUNCTION`) and predefined callbacks (e.g., `MPI_COMM_NULL_COPY_FN`).

Rationale. For user-defined callback functions (e.g., of type `MPI_Comm_copy_attr_function` or `COMM_COPY_ATTR_FUNCTION`) and their predefined callbacks (e.g., `MPI_COMM_NULL_COPY_FN`), the ierror argument is not optional. The MPI library must always call these routines with an actual ierror argument. Therefore, these user-defined functions need not check whether the MPI library calls these routines with or without an actual ierror output argument. (*End of rationale.*)

The MPI Fortran bindings in the `mpi_f08` module are designed based on the Fortran 2008 standard [45] together with the Technical Specification “TS 29113 Further Interoperability with C” [46] of the ISO/IEC JTC1/SC22/WG5 (Fortran) working group.

Rationale. The features in TS 29113 on further interoperability with C were decided on by ISO/IEC JTC1/SC22/WG5 and designed by PL22.3 (formerly J3) to support a higher level of integration between Fortran-specific features and C than was provided in the Fortran 2008 standard; part of this design is based on requirements from the MPI Forum to support MPI-3.0. According to [46], “an ISO/IEC TS is reviewed after three years in order to decide whether it will be confirmed for a further three years, revised to become an International Standard, or withdrawn. If the ISO/IEC TS is confirmed, it is reviewed again after a further three years, at which time it must either be transformed into an International Standard or be withdrawn.”

The TS 29113 contains the following language features that are needed for the MPI bindings in the `mpi_f08` module: assumed-type and assumed-rank. It is important that any possible actual argument can be used for such dummy arguments, e.g., scalars, arrays, assumed-shape arrays, assumed-size arrays, allocatable arrays, and with any element type, e.g., `REAL`, `CHARACTER*5`, `CHARACTER*(*)`, sequence derived types, or `BIND(C)` derived types. Especially for backward compatibility reasons, it is important that any possible actual argument in an implicit interface implementation of a choice buffer dummy argument (e.g., with `mpif.h` without argument-checking) can be used in an implementation with assumed-type and assumed-rank argument in an explicit interface (e.g., with the `mpi_f08` module).

A further feature useful for MPI is the extension of the semantics of the ASYNCHRONOUS attribute: In F2003 and F2008, this attribute could be used only to protect buffers of Fortran asynchronous I/O. With TS 29113, this attribute now also covers asynchronous communication occurring within library routines written in C.

The MPI Forum hereby wishes to acknowledge this important effort by the Fortran PL22.3 and WG5 committee. (*End of rationale.*)

19.1.3 Fortran Support Through the `mpi` Module

An MPI implementation providing a Fortran interface must provide a module named `mpi` that can be used in a Fortran program. Within all MPI function specifications, the second of the set of two Fortran routine interface specifications is provided by this module. This module must:

- Define all named MPI constants.
- Declare MPI functions that return a value.
- Provide explicit interfaces according to the Fortran routine interface specifications. This module therefore guarantees compile-time argument checking and allows positional and keyword-based argument lists. If an implementation is paired with a compiler that either does not support `TYPE(*)`, `DIMENSION(..)` from TS 29113, or is otherwise unable to ignore the types of choice buffers, then the implementation must provide explicit interfaces only for MPI routines with no choice buffer arguments. See Section 19.1.6 for more details.
- Define all MPI handles as type `INTEGER`.
- Define the derived type `MPI_Status` and all named handle types that are used in the `mpi_f08` module. For these named handle types, overload the operators `.EQ.` and `.NE.` to allow handle comparison via the `.EQ.`, `.NE.`, `==` and `/=` operators.

Rationale. They are needed only when the application converts old-style `INTEGER` handles into new-style handles with a named type. (*End of rationale.*)

- A high quality MPI implementation may enhance the interface by using the ASYNCHRONOUS attribute in the same way as in the `mpi_f08` module if it is supported by the underlying compiler.
- Set the LOGICAL compile-time constant `MPI_ASYNC_PROTECTS_NONBLOCKING` to `.TRUE.` if the ASYNCHRONOUS attribute is used in all nonblocking interfaces **and** the underlying Fortran compiler supports the ASYNCHRONOUS attribute for MPI communication (as part of TS 29113), otherwise to `.FALSE.`

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 the MPI standard 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 nonconforming. See Section 19.1.12 for more details. (*End of advice to users.*)

- A high quality MPI implementation may enhance the interface by using TYPE(*), DIMENSION(..) choice buffer dummy arguments instead of using nonstandardized extensions such as !\$PRAGMA IGNORE_TKR or a set of overloaded functions as described by M. Hennecke in [32], if the compiler supports this TS 29113 language feature. See Section 19.1.6 for further details.
- Set the LOGICAL compile-time constant MPI_SUBARRAYS_SUPPORTED to .TRUE. if all choice buffer arguments in all nonblocking, split collective and persistent communication routines are declared with TYPE(*), DIMENSION(..), otherwise set it to .FALSE.. When MPI_SUBARRAYS_SUPPORTED is defined as .TRUE., noncontiguous sub-arrays can be used as buffers in nonblocking routines.
- Set the MPI_SUBARRAYS_SUPPORTED compile-time constant to .FALSE. and declare choice buffers with a compiler-dependent mechanism that overrides type checking if the underlying Fortran compiler does not support the TS 29113 assumed-type and assumed-rank features. In this case, the use of noncontiguous sub-arrays in nonblocking calls may be disallowed. See Section 19.1.6 for details.

An MPI implementation may provide other features in the mpi module that enhance the usability of MPI while maintaining adherence to the standard. For example, it may provide INTENT information in these interface blocks.

Advice to implementors. The appropriate INTENT may be different from what is given in the MPI language-neutral bindings. Implementations must choose INTENT so that the function adheres to the MPI standard, e.g., by defining the INTENT as provided in the mpi_f08 bindings. (*End of advice to implementors.*)

Rationale. The intent given by the MPI generic interface is not precisely defined and does not in all cases correspond to the correct Fortran INTENT. For instance, receiving into a buffer specified by a datatype with absolute addresses may require associating MPI_BOTTOM with a dummy OUT argument. Moreover, “constants” such 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.*)

19.1.4 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. The `mpif.h` include file has not been deprecated in order to retain strong backward compatibility. Internally, `mpif.h` and the `mpi` module may be implemented so that essentially the same library implementation of the MPI routines can be used.

(*End of rationale.*)

19.1.5 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. When this section was originally introduced in MPI-3.0, the major goals for the three Fortran support methods were:

- 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 with TS 29113.

The design expected that all dummy arguments in the MPI Fortran interfaces are interoperable with C according to Fortran 2008 with 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`.

Table 19.1: Specific Fortran procedure names and related calling conventions. `MPI_ISEND` is used as an example. For routines without choice buffers, only 1A and 2A apply.

No.	Specific procedure name	Calling convention
1A	<code>MPI_Isend_f08</code>	Fortran interface and arguments, as in Annex A.4, except that in routines with a choice buffer dummy argument, this dummy argument is implemented with nonstandard extensions like <code>!\$PRAGMA IGNORE_TKR</code> , which provides a call-by-reference argument without type, kind, and dimension checking.
1B	<code>MPI_Isend_f08ts</code>	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 <code>TYPE(*)</code> , <code>DIMENSION(..)</code> .
2A	<code>MPI_ISEND</code>	Fortran interface and arguments, as in Annex A.5, except that in routines with a choice buffer dummy argument, this dummy argument is implemented with nonstandard extensions like <code>!\$PRAGMA IGNORE_TKR</code> , which provides a call-by-reference argument without type, kind, and dimension checking.
2B	<code>MPI_ISEND_FTS</code>	Fortran interface and arguments, as in Annex A.5, but only for routines with one or more choice buffer dummy arguments; these dummy arguments are implemented with <code>TYPE(*)</code> , <code>DIMENSION(..)</code> . In <code>mpif.h</code> only, the postfix “_FTS” for <code>MPI_NEIGHBOR_ALLGATHERV_INIT</code> , <code>MPI_NEIGHBOR_ALLTOALLV_INIT</code> , and <code>MPI_NEIGHBOR_ALLTOALLW_INIT</code> is shortened to “_F”.

Therefore, this section was rewritten as an erratum to MPI-3.0. (*End of rationale.*)

A Fortran call to an MPI routine shall result in a call to a procedure with one of the specific procedure names and calling conventions, as described in Table 19.1. Case is not significant in the names.

Note that for the deprecated routines in Section 16.1, which are reported only in Annex A.5, scheme 2A is utilized in the `mpi` module and `mpif.h`, and also in the `mpi_f08` module.

To set `MPI_SUBARRAYS_SUPPORTED` to `.TRUE.` within a Fortran support method, it is required that all nonblocking and split-collective routines with buffer arguments are implemented according to 1B and 2B, i.e., with `MPI_Xxxx_f08ts` in the `mpi_f08` module, and with `MPI_XXXX_FTS` in the `mpi` module and the `mpif.h` include file.

The `mpi` and `mpi_f08` modules and the `mpif.h` include file will each correspond to exactly one implementation scheme from Table 19.1. However, the MPI library may contain multiple implementation schemes from Table 19.1.

Advice to implementors. This may be desirable for backwards binary compatibility

1 in the scope of a single MPI implementation, for example. (*End of advice to imple-*
2 *mentors.*)

3
4 *Rationale.* After a compiler provides the facilities from TS 29113, i.e., `TYPE(*)`,
5 `DIMENSION(...)`, it is possible to change the bindings within a Fortran support method
6 to support subarrays without recompiling the complete application provided that
7 the previous interfaces with their specific procedure names are still included in the
8 library. Of course, only recompiled routines can benefit from the added facilities.
9 There is no binary compatibility conflict because each interface uses its own spe-
10 cific procedure names and all interfaces use the same constants (except the value of
11 `MPI_SUBARRAYS_SUPPORTED` and `MPI_ASYNC_PROTECTS_NONBLOCKING`) and type
12 definitions. After a compiler also ensures that buffer arguments of nonblocking MPI
13 operations can be protected through the `ASYNCHRONOUS` attribute, and the procedure
14 declarations in the `mpi_f08` and `mpi` module and the `mpif.h` include file declare
15 choice buffers with the `ASYNCHRONOUS` attribute, then the value of
16 `MPI_ASYNC_PROTECTS_NONBLOCKING` can be switched to `.TRUE.` in the module def-
17 inition and include file. (*End of rationale.*)

18
19 *Advice to users.* Partial recompilation of user applications when upgrading MPI
20 implementations is a highly complex and subtle topic. Users are strongly advised to
21 consult their MPI implementation’s documentation to see exactly what is—and what
22 is not—supported. (*End of advice to users.*)

23
24 Within the `mpi_f08` and `mpi` modules and `mpif.h`, for all MPI procedures, a second
25 procedure with the same calling conventions shall be supplied, except that the name is
26 modified by prefixing with the letter “P”, e.g., `PMPI_Isend`. The specific procedure names
27 for these `PMPI_Xxxx` procedures must be different from the specific procedure names for
28 the `MPI_Xxxx` procedures and are not specified by this standard.

29 A user-written or middleware profiling routine should provide the same specific Fortran
30 procedure names and calling conventions, and therefore can interpose itself as the MPI
31 library routine. The profiling routine can internally call the matching
32 `PMPI` routine with any of its existing bindings, except for routines that have callback routine
33 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
34 invoke the corresponding `PMPI` routine using the same Fortran support method as used in
35 the calling application program, because the C, `mpi_f08` and `mpi` callback prototypes are
36 different or the meaning of the choice buffer or `attribute_val` arguments are different.
37

38
39 *Advice to users.* Although for each support method and MPI routine (e.g.,
40 `MPI_ISEND` in `mpi_f08`), multiple routines may need to be provided to intercept the
41 specific procedures in the MPI library (e.g., `MPI_Isend_f08` and `MPI_Isend_f08ts`), each
42 profiling routine itself uses only one support method (e.g., `mpi_f08`) and calls the
43 real MPI routine through the one `PMPI` routine defined in this support method (i.e.,
44 `PMPI_Isend` in this example). (*End of advice to users.*)

45 *Advice to implementors.* If all of the following conditions are fulfilled:

- 46
47 • the handles in the `mpi_f08` module occupy one Fortran numerical storage unit
48 (same as an `INTEGER` handle),

- the internal argument passing mechanism used to pass an actual `ierror` argument to a nonoptional `ierror` dummy argument is binary compatible to passing an actual `ierror` argument to an `ierror` dummy argument that is declared as `OPTIONAL`,
- the internal argument passing mechanism for `ASYNCHRONOUS` and non-`ASYNCHRONOUS` arguments is the same,
- the internal routine call mechanism is the same for the Fortran and the C compilers for which the MPI library is compiled, and
- the compiler does not provide TS 29113,

then the implementor may use the same internal routine implementations for all Fortran support methods but with several different specific procedure names. If the accompanying Fortran compiler supports TS 29113, then the new routines are needed only for routines with choice buffer arguments. (*End of advice to implementors.*)

Advice to implementors. In the Fortran support method `mpif.h`, compile-time argument checking can be also implemented for all routines. For `mpif.h`, the argument names are not specified through the MPI standard, i.e., only positional argument lists are defined, and not key-word based lists. Due to the rule that `mpif.h` must be valid for fixed and free source form, the subroutine declaration is restricted to one line with 72 characters. To keep the argument lists short, each argument name can be shortened to a minimum of one character. With this, the three longest subroutine declaration statements are

```
SUBROUTINE PMPI_DIST_GRAPH_CREATE_ADJACENT(a,b,c,d,e,f,g,h,i,j,k)
SUBROUTINE PMPI_NEIGHBOR_ALLTOALLW_INIT(a,b,c,d,e,f,g,h,i,j,k,l)
SUBROUTINE PMPI_NEIGHBOR_ALLTOALLV_INIT(a,b,c,d,e,f,g,h,i,j,k,l)
```

with 71 and 70 characters each. With buffers implemented with TS 29113, the specific procedure names have an additional postfix. Some of the longest of such interface definitions are

```
INTERFACE PMPI_NEIGHBOR_ALLTOALLW_INIT
SUBROUTINE PMPI_NEIGHBOR_ALLTOALLW_INIT_F(a,b,c,d,e,f,g,h,i,j,j,k)
INTERFACE PMPI_NEIGHBOR_ALLGATHERV_INIT
SUBROUTINE PMPI_NEIGHBOR_ALLGATHERV_INIT_F(a,b,c,d,e,f,g,h,i,j,k)
INTERFACE PMPI_RGET_ACCUMULATE
SUBROUTINE PMPI_RGET_ACCUMULATE_FTS(a,b,c,d,e,f,g,h,i,j,k,l,m,n)
```

with 72, 71, and 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 name is `PMPI_Reduce_scatter_block_init_c_f08ts` with 38 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
```

```

1  INTEGER :: MPI_VAL
2  END TYPE MPI_Comm
3  ...
4  INTERFACE MPI_Comm_rank ! (as defined in Chapter 6)
5      SUBROUTINE MPI_Comm_rank_f08(comm, rank, ierror)
6          IMPORT :: MPI_Comm
7          TYPE(MPI_Comm),      INTENT(IN)  :: comm
8          INTEGER,              INTENT(OUT) :: rank
9          INTEGER, OPTIONAL,    INTENT(OUT) :: ierror
10     END SUBROUTINE
11 END INTERFACE
12 END MODULE mpi_f08
13
14 MODULE mpi
15     INTERFACE MPI_Comm_rank ! (as defined in Chapter 6)
16         SUBROUTINE MPI_Comm_rank(comm, rank, ierror)
17             INTEGER, INTENT(IN)  :: comm ! The INTENT may be added although
18             INTEGER, INTENT(OUT) :: rank ! it is not defined in the
19             INTEGER, INTENT(OUT) :: ierror ! official routine definition.
20         END SUBROUTINE
21     END INTERFACE
22 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):

```

23 ! 23456789012345678901234567890123456789012345678901234567890123456789012
24 INTERFACE MPI_Comm_rank ! (as defined in Chapter 6)
25     SUBROUTINE MPI_Comm_rank(comm, rank, ierror)
26         INTEGER, INTENT(IN)  :: comm ! The argument names may be
27         INTEGER, INTENT(OUT) :: rank ! shortened so that the
28         INTEGER, INTENT(OUT) :: ierror ! subroutine line fits to the
29     END SUBROUTINE
30     ! maximum of 72 characters.
31 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:

```

32 SUBROUTINE MPI_Isend_f08ts(buf, count, datatype, dest, tag, comm, request, ierror)
33     USE :: mpi_f08, my_noname => MPI_Isend_f08ts
34     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
35     INTEGER, INTENT(IN) :: count, dest, tag
36     TYPE(MPI_Datatype), INTENT(IN) :: datatype
37     TYPE(MPI_Comm), INTENT(IN) :: comm
38     TYPE(MPI_Request), INTENT(OUT) :: request
39     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40     ! ... some code for the begin of profiling
41     call PMPI_Isend (buf, count, datatype, dest, tag, comm, request, ierror)
42     ! ... some code for the end of profiling
43 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 19.1 so that the MPI Fortran routines are interceptable as described above. (*End of advice to implementors.*)

19.1.6 MPI for Different Fortran Standard Versions

This section describes which Fortran interface functionality can be provided for different versions of the Fortran standard.

- *For Fortran 77* with some extensions:
 - 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_XXX_KIND concept.

- 1 – Fortran derived `TYPE`s and the `SEQUENCE` attribute.
- 2 – The `OPTIONAL` attribute for dummy arguments.
- 3
- 4 – Cray pointers, which are a nonstandard compiler extension, are needed for the
- 5 use of `MPI_ALLOC_MEM`.

6 With these features, MPI-1.1 – MPI-2.2 can be implemented without restrictions.
 7 MPI-3.0 and later can be implemented with some restrictions. The Fortran support
 8 methods are abbreviated with `S1` = the `mpi_f08` module, `S2` = the `mpi` module, and `S3`
 9 = the `mpif.f` include file. If not stated otherwise, restrictions exist for each method
 10 that prevent implementing the complete semantics of MPI.

- 11
- 12 – `MPI_SUBARRAYS_SUPPORTED` equals `.FALSE.`, i.e., subscript triplets and non-
- 13 contiguous subarrays cannot be used as buffers in nonblocking routines, RMA,
- 14 or split-collective I/O.
- 15 – `S1`, `S2`, and `S3` can be implemented, but for `S1`, only a preliminary implementa-
- 16 tion is possible.
- 17 – In this preliminary interface of `S1`, the following changes are necessary:
- 18 * `TYPE(*)`, `DIMENSION(..)` is substituted by nonstandardized extensions like
- 19 `!$PRAGMA IGNORE_TKR`.
- 20 * The `ASYNCHRONOUS` attribute is omitted.
- 21 * `PROCEDURE(...)` callback declarations are substituted by `EXTERNAL`.
- 22
- 23 – The specific procedure names are specified in Section 19.1.5.
- 24
- 25 – Due to the rules specified in Section 19.1.5, choice buffer declarations should be
- 26 implemented only with nonstandardized extensions like `!$PRAGMA IGNORE_TKR`
- 27 (as long as F2008 with TS 29113 is not available).

28 In `S2` and `S3`: Without such extensions, routines with choice buffers should be
 29 provided with an implicit interface, instead of overloading with a different MPI
 30 function for each possible buffer type (as mentioned in Section 19.1.11). Such
 31 overloading would also imply restrictions for passing Fortran derived types as
 32 choice buffer, see also Section 19.1.15.

33

34 Only in `S1`: The implicit interfaces for routines with choice buffer arguments
 35 imply that the `ierror` argument cannot be defined as `OPTIONAL`. For this reason,
 36 it is recommended not to provide the `mpi_f08` module if such an extension is not
 37 available.

- 38 – The `ASYNCHRONOUS` attribute can **not** be used in applications to protect buffers
- 39 in nonblocking MPI calls (`S1`–`S3`).
- 40 – The `TYPE(C_PTR)` binding of the `MPI_ALLOC_MEM` and `MPI_WIN_ALLOCATE`
- 41 routines is not available.
- 42
- 43 – In `S1` and `S2`, the definition of the handle types (e.g., `TYPE(MPI_Comm)` and
- 44 the status type `TYPE(MPI_Status)` must be modified: The `SEQUENCE` attribute
- 45 must be used instead of `BIND(C)` (which is not available in Fortran 90/95). This
- 46 restriction implies that the application must be fully recompiled if one switches
- 47 to an MPI library for Fortran 2003 and later because the internal memory size of
- 48 the handles may have changed. For this reason, an implementor may choose not

to provide the `mpi_f08` module for Fortran 90 compilers. In this case, the `mpi_f08` handle types and all routines, constants and types related to `TYPE(MPI_Status)` (see Section 19.3.5) are also not available in the `mpi` module and `mpif.h`.

- *For Fortran 95:*

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 and later 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 and later 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 nonstandardized extensions like `!$PRAGMA IGNORE_TKR`.
- The specific procedure names are specified in Section 19.1.5.
- 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 nonstandardized extensions like !\$PRAGMA IGNORE_TKR are not available.

- For Fortran 2008 with TS 29113 and later and
For Fortran 2003 with TS 29113:

The major features 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 and later 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 set to .TRUE. and will use the ASYNCHRONOUS attribute in the same way as in S1.
- If nonstandardized 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.)

19.1.7 Requirements on Fortran Compilers

MPI-3.0 (and later) compliant Fortran bindings are not only a property of the MPI library itself, but rather a property of an MPI library together with the Fortran compiler suite for which it is compiled.

Advice to users. Users must take appropriate steps to ensure that proper options are specified to compilers. MPI libraries must document these options. Some MPI

libraries are shipped together with special compilation scripts (e.g., `mpif90`, `mpicc`) that set these options automatically. (*End of advice to users.*)

An MPI library together with the Fortran compiler suite is only compliant with MPI-3.0 (and later), as referred by `MPI_GET_VERSION`, if all the solutions described in Sections 19.1.11 through 19.1.19 work correctly. Based on this rule, major requirements for all three Fortran support methods (i.e., the `mpi_f08` and `mpi` modules, and `mpif.h`) are:

- The language features assumed-type and assumed-rank from Fortran 2008 TS 29113 [46] are available. This is required only for `mpi_f08`. As long as this requirement is not supported by the compiler, it is valid to build an MPI library that implements the `mpi_f08` module with `MPI_SUBARRAYS_SUPPORTED` set to `.FALSE.`
- “Simply contiguous” arrays and scalars must be passed to choice buffer dummy arguments of nonblocking routines with call by reference. This is needed only if one of the support methods does not use the `ASYNCHRONOUS` attribute. See Section 19.1.12 for more details.
- `SEQUENCE` and `BIND(C)` derived types are valid as actual arguments passed to choice buffer dummy arguments, and, in the case of `MPI_SUBARRAYS_SUPPORTED` set to `.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` set to `.FALSE.` Observation of these rules by the MPI application developer is especially recommended for backward compatibility of existing applications that use the `mpi` module or the `mpif.h` include file. The rules are as follows:

- 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 793 and Section 19.1.8, and `DD` on page 794) solve the problems described in Section 19.1.17.
- The problems with temporary data movement (described in detail in Section 19.1.18) are solved as long as the application uses different sets of variables for the nonblocking communication (or nonblocking or split collective I/O) and the computation when overlapping communication and computation.

- Problems caused by automatic and permanent data movement (e.g., within a garbage collection, see Section 19.1.19) are resolved **without** any further requirements on the application program, neither on the usage of the buffers, nor on the declaration of application routines that are involved in invoking MPI procedures.

All of these rules are valid for the `mpi_f08` and `mpi` modules and independently of whether `mpif.h` uses explicit interfaces.

Advice to implementors. Some of these rules are already part of the Fortran 2003 standard, some of these requirements require the Fortran TS 29113 [46], and some of these requirements for MPI are beyond the scope of TS 29113. (*End of advice to implementors.*)

19.1.8 Additional Support for Fortran Register-Memory-Synchronization

As described in Section 19.1.17, 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)

Fortran 2008 binding

MPI_F_sync_reg(buf)

TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf

Fortran 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.*)

19.1.9 Additional Support for Fortran Numeric Intrinsic Types

MPI provides a small number of named datatypes that correspond to named intrinsic types supported by C and Fortran. These include MPI_INTEGER, MPI_REAL, MPI_INT, MPI_DOUBLE, etc., as well as the optional types MPI_REAL4, MPI_REAL8, etc. There is a one-to-one correspondence between language declarations and MPI types.

Fortran (starting with Fortran 90) provides so-called KIND-parameterized types. These 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 nondefault 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 777) 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	<code>newtype</code>	the requested MPI datatype (handle)

C binding

`int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)`

Fortran 2008 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
```

Fortran 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 777.

It is erroneous to supply values for `p` and `r` not supported by the compiler.

`MPI_TYPE_CREATE_F90_COMPLEX(p, r, newtype)`

IN	<code>p</code>	precision, in decimal digits (integer)
IN	<code>r</code>	decimal exponent range (integer)
OUT	<code>newtype</code>	the requested MPI datatype (handle)

C binding

`int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype)`

Fortran 2008 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`

Fortran 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 777.

It is erroneous to supply values for `p` and `r` not supported by the compiler.

`MPI_TYPE_CREATE_F90_INTEGER(r, newtype)`

IN	<code>r</code>	decimal exponent range, i.e., number of decimal digits (integer)
OUT	<code>newtype</code>	the requested MPI datatype (handle)

C binding

`int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)`

Fortran 2008 binding

`MPI_Type_create_f90_integer(r, newtype, ierror)`
 INTEGER, INTENT(IN) :: `r`
 TYPE(MPI_Datatype), INTENT(OUT) :: `newtype`
 INTEGER, OPTIONAL, INTENT(OUT) :: `ierror`

Fortran binding

`MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR)`

1 INTEGER R, NEWTYPE, IERROR

2
3 This function returns a predefined MPI datatype that matches an INTEGER variable
4 of KIND `selected_int_kind(r)`. Matching rules for datatypes created by this function are
5 analogous to the matching rules for datatypes created by `MPI_TYPE_CREATE_F90_REAL`.
6 Restrictions on using the returned datatype with the "external32" data representation are
7 given on page 777.

8 It is erroneous to supply a value for `r` that is not supported by the compiler.

9 Example:

```
10  integer      longtype, quadtype
11  integer, parameter :: long = selected_int_kind(15)
12  integer(long) ii(10)
13  real(selected_real_kind(30)) x(10)
14  call MPI_TYPE_CREATE_F90_INTEGER(15, longtype, ierror)
15  call MPI_TYPE_CREATE_F90_REAL(30, MPI_UNDEFINED, quadtype, ierror)
16  ...
17  call MPI_SEND(ii, 10, longtype, ...)
18  call MPI_SEND(x, 10, quadtype, ...)
```

19
20 *Advice to users.* The datatypes returned by the above functions are predefined
21 datatypes. They cannot be freed; they do not need to be committed; they can be
22 used with predefined reduction operations. There are two situations in which they
23 behave differently syntactically, but not semantically, from the MPI named predefined
24 datatypes.

- 25 1. `MPI_TYPE_GET_ENVELOPE` returns special combiners that allow a program to
26 retrieve the values of `p` and `r`.
- 27 2. Because the datatypes are not named, they cannot be used as compile-time
28 initializers or otherwise accessed before a call to one of the
29 `MPI_TYPE_CREATE_F90_XXX` routines.

30
31 If a variable was declared specifying a nondefault KIND value that was not obtained
32 with `selected_real_kind()` or `selected_int_kind()`, the only way to obtain a match-
33 ing MPI datatype is to use the size-based mechanism described in the next section.
34 (*End of advice to users.*)

35
36 *Advice to implementors.* An application may often repeat a call to
37 `MPI_TYPE_CREATE_F90_XXX` with the same combination of `(XXX,p,r)`. The appli-
38 cation is not allowed to free the returned predefined, unnamed datatype handles. To
39 prevent the creation of a potentially huge amount of handles, a high quality MPI imple-
40 mentation should return the same datatype handle for the same `(REAL/COMPLEX/
41 INTEGER,p,r)` combination. Checking for the combination `(p,r)` in the preceding call
42 to `MPI_TYPE_CREATE_F90_XXX` and using a hash table to find formerly generated
43 handles should limit the overhead of finding a previously generated datatype with
44 same combination of `(XXX,p,r)`. (*End of advice to implementors.*)

45
46 *Rationale.* The `MPI_TYPE_CREATE_F90_REAL/COMPLEX/INTEGER` interface
47 needs as input the original range and precision values to be able to define useful
48 and compiler-independent external (Section 14.5.2) or user-defined (Section 14.5.3)

data representations, and in order to be able to perform automatic and efficient data conversions in a heterogeneous environment. (*End of rationale.*)

We now specify how the datatypes described in this section behave when used with the "external32" external data representation described in Section 14.5.2.

The "external32" representation specifies data formats for integer and floating point values. Integer values are represented in two's complement big-endian format. Floating point values are represented by one of three IEEE formats. These are the IEEE "Single," "Double," and "Double Extended" formats, requiring 4, 8, and 16 bytes of storage, respectively. For the IEEE "Double Extended" formats, MPI specifies a Format Width of 16 bytes, with 15 exponent bits, bias = +10383, 112 fraction bits, and an encoding analogous to the "Double" format.

The "external32" representations of the datatypes returned by MPI_TYPE_CREATE_F90_REAL/COMPLEX/INTEGER are given by the following rules.

For MPI_TYPE_CREATE_F90_REAL:

```

if      (p > 33) or (r > 4931) then  external32 representation
                                     is undefined
else 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:

```

1 MPI_REAL4
2 MPI_REAL8
3 MPI_REAL16
4 MPI_COMPLEX8
5 MPI_COMPLEX16
6 MPI_COMPLEX32
7 MPI_INTEGER1
8 MPI_INTEGER2
9 MPI_INTEGER4
10 MPI_INTEGER8
11 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 function allows a user to obtain a size-specific MPI datatype for any intrinsic Fortran type.

```
MPI_TYPE_MATCH_SIZE(typeclass, size, datatype)
```

IN	typeclass	generic type specifier (integer)
IN	size	size, in bytes, of representation (integer)
OUT	datatype	datatype with correct type, size (handle)

C binding

```
int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *datatype)
```

Fortran 2008 binding

```

MPI_Type_match_size(typeclass, size, datatype, ierror)
  INTEGER, INTENT(IN) :: typeclass, size
  TYPE(MPI_Datatype), INTENT(OUT) :: datatype
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

Fortran binding

```

MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, DATATYPE, IERROR)
  INTEGER TYPECLASS, SIZE, DATATYPE, IERROR

```

typeclass is one of MPI_TYPECLASS_REAL, MPI_TYPECLASS_INTEGER and MPI_TYPECLASS_COMPLEX, corresponding to the desired **typeclass**. The function returns an MPI datatype matching a local variable of type (**typeclass**, **size**).

This function returns a reference (handle) to one of the predefined named datatypes, not a duplicate. This type cannot be freed. MPI_TYPE_MATCH_SIZE can be used to obtain a size-specific type that matches a Fortran numeric intrinsic type by first calling storage_size() in order to compute the variable size in bits, dividing it by eight, and then calling MPI_TYPE_MATCH_SIZE to find a suitable datatype. In C, one can use the C operator sizeof() (which returns the size in bytes) instead of storage_size() (which returns the size in bits). In addition, for variables of default kind the variable's size can be computed by a call to MPI_TYPE_GET_EXTENT, if the typeclass is known. It is erroneous to specify a size not supported by the compiler.

Rationale. This is a convenience function. Without it, it can be tedious to find the correct named type. See note to implementors below. (*End of rationale.*)

Advice to implementors. This function could be implemented as a series of tests.

```
int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *rtype)
{
    switch(typeclass) {
        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 two'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)
size = storage_size(x) / 8
call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
if (myrank .eq. 0) then
```

```

1      ... initialize x ...
2      call MPI_SEND(x, xtype, 100, 1, ...)
3  else if (myrank .eq. 1) then
4      call MPI_RECV(x, xtype, 100, 0, ...)
5  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:

```

21  real(selected_real_kind(5)) x(100)
22  size = storage_size(x) / 8
23  call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
24
25  if (myrank .eq. 0) then
26      call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo', &
27                          MPI_MODE_CREATE+MPI_MODE_WRONLY, &
28                          MPI_INFO_NULL, fh, ierror)
29      call MPI_FILE_SET_VIEW(fh, zero, xtype, xtype, 'external32', &
30                          MPI_INFO_NULL, ierror)
31      call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
32      call MPI_FILE_CLOSE(fh, ierror)
33  endif
34
35  call MPI_BARRIER(MPI_COMM_WORLD, ierror)
36
37  if (myrank .eq. 1) then
38      call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo', MPI_MODE_RDONLY, &
39                          MPI_INFO_NULL, fh, ierror)
40      call MPI_FILE_SET_VIEW(fh, zero, xtype, xtype, 'external32', &
41                          MPI_INFO_NULL, ierror)
42      call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
43      call MPI_FILE_CLOSE(fh, ierror)
44  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.*)

19.1.10 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 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 with 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 MPI-2 functions also take INTEGER arguments of nondefault KIND. See Section 2.6 and Section 5.1.1 for more information.

Sections 19.1.11 through 19.1.19 describe several problems in detail that concern the interaction of MPI and Fortran as well as their solutions. Some of these solutions require special capabilities from the compilers. Major requirements are summarized in Section 19.1.7.

19.1.11 Problems Due to Strong Typing

All MPI functions with choice arguments associate actual arguments of different Fortran datatypes with the same dummy argument. This is not allowed by Fortran 77, and in Fortran 90, it is technically only allowed if the function is overloaded with a different function for each type (see also Section 19.1.6). In C, the use of void* formal arguments avoids these problems. Similar to C, with Fortran 2008 with TS 29113 (and later) together with the mpi_f08 module, the problem is avoided by declaring choice arguments with TYPE(*), DIMENSION(..), i.e., as assumed-type and assumed-rank dummy arguments.

Using INCLUDE 'mpif.h', the following code fragment is technically invalid and may generate a compile-time error.

```

integer i(5)
real    x(5)
...
call mpi_send(x, 5, MPI_REAL, ...)
call mpi_send(i, 5, MPI_INTEGER, ...)

```

In practice, it is rare for compilers to do more than issue a warning. When using either the mpi_f08 or mpi module, the problem is usually resolved through the assumed-type and assumed-rank declarations of the dummy arguments, or with a compiler-dependent mechanism that overrides type checking for choice arguments.

It is also technically invalid in Fortran to pass a scalar actual argument to an array dummy argument that is not a choice buffer argument. Thus, when using the mpi_f08 or mpi module, the following code fragment usually generates an error since the dims and periods arguments to MPI_CART_CREATE are declared as assumed size arrays INTEGER :: DIMS(*) and LOGICAL :: PERIODS(*).

```

USE mpi_f08      ! or USE mpi
INTEGER size
CALL MPI_Cart_create(comm_old, 1, size, .TRUE., .TRUE., comm_cart, ierror)

```

Although this is a nonconforming MPI call, compiler warnings are not expected (but may occur) when using INCLUDE 'mpif.h' and this include file does not use Fortran explicit interfaces.

19.1.12 Problems Due to Data Copying and Sequence Association with Subscript Triplets

Arrays with subscript **triplets** describe Fortran subarrays with or without strides, e.g.,

```
REAL a(100,100,100)
CALL MPI_Send(a(11:17, 12:99:3, 1:100), 7*30*100, MPI_REAL, ...)
```

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 noncontiguous 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 nonblocking 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 noncontiguous user buffers when the nonblocking operation is started, and release this buffer not before the nonblocking communication has completed (e.g., the `MPI_WAIT` routine). Efficient implementations may avoid such additional memory-to-memory data copying. (*End of advice to implementors.*)

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

1 can be from the beginning of a nonblocking operation until the completion of the
 2 operation although beginning and completion are implemented in different rou-
 3 tines. (*End of rationale.*)

- 4 • If `MPI_SUBARRAYS_SUPPORTED` equals `.FALSE.`:

5
 6 In this case, the use of Fortran arrays with subscript triplets as actual choice buffer
 7 arguments in any nonblocking MPI operation (which also includes persistent request,
 8 and split collectives) may cause undefined behavior. They may, however, be used in
 9 blocking MPI operations.

10 Implicit in MPI is the idea of a contiguous chunk of memory accessible through a
 11 linear address space. MPI copies data to and from this memory. An MPI program
 12 specifies the location of data by providing memory addresses and offsets. In the C
 13 language, sequence association rules plus pointers provide all the necessary low-level
 14 structure.

15 In Fortran, array data is not necessarily stored contiguously. For example, the array
 16 section `A(1:N:2)` involves only the elements of `A` with indices 1, 3, 5, The same is
 17 true for a pointer array whose target is such a section. Most compilers ensure that an
 18 array that is a dummy argument is held in contiguous memory if it is declared with
 19 an explicit shape (e.g., `B(N)`) or is of assumed size (e.g., `B(*)`). If necessary, they do
 20 this by making a copy of the array into contiguous memory.¹

21 Because MPI dummy buffer arguments are assumed-size arrays if
 22 `MPI_SUBARRAYS_SUPPORTED` equals `.FALSE.`, this leads to a serious problem for a
 23 nonblocking call: the compiler copies the temporary array back on return but MPI
 24 continues to copy data to the memory that held it. For example, consider the following
 25 code fragment:
 26

```
27 real a(100)
28 call MPI_Irecv(a(1:100:2), MPI_REAL, 50, ...)
```

29 Since the first dummy argument to `MPI_Irecv` is an assumed-size array (`<type>`
 30 `buf(*)`), the array section `a(1:100:2)` is copied to a temporary before being passed
 31 to `MPI_Irecv`, so that it is contiguous in memory. `MPI_Irecv` returns immediately,
 32 and data is copied from the temporary back into the array `a`. Sometime later, MPI
 33 may write to the address of the deallocated temporary. Copying is also a problem
 34 for `MPI_Isend` since the temporary array may be deallocated before the data has all
 35 been sent from it.

36 Most Fortran 90 compilers do not make a copy if the actual argument is the whole
 37 of an explicit-shape or assumed-size array or is a “simply contiguous” section such
 38 as `A(1:N)` of such an array. (“Simply contiguous” is defined in the next paragraph.)
 39 Also, many compilers treat allocatable arrays the same as they treat explicit-shape
 40 arrays in this regard (though we know of one that does not). However, the same is not
 41 true for assumed-shape and pointer arrays; since they may be discontinuous, copying
 42 is often done. It is this copying that causes problems for MPI as described in the
 43 previous paragraph.
 44

45 According to the Fortran 2008 Standard, Section 6.5.4, a “simply contiguous” array
 46 section is

47 ¹Technically, the Fortran standard is worded to allow noncontiguous storage of any array data, unless the
 48 dummy argument has the `CONTIGUOUS` attribute.

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 nontrivial 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.

19.1.13 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, ...)

```

1 Fortran arrays with a vector subscript must not be used as actual choice buffer argu-
 2 ments in any nonblocking or split collective MPI operations. They may, however, be used
 3 in blocking MPI operations.
 4

5 19.1.14 Special Constants

6 MPI requires a number of special “constants” that cannot be implemented as normal Fortran
 7 constants, e.g., MPI_BOTTOM. The complete list can be found in Section 2.5.4. In C, these
 8 are implemented as constant pointers, usually as NULL and are used where the function
 9 prototype calls for a pointer to a variable, not the variable itself.
 10

11 In Fortran, using special values for the constants (e.g., by defining them through
 12 parameter statements) is not possible because an implementation cannot distinguish these
 13 values from valid data. Typically these constants are implemented as predefined static vari-
 14 ables (e.g., a variable in an MPI-declared COMMON block), relying on the fact that the target
 15 compiler passes data by address. Inside the subroutine, the address of the actual choice
 16 buffer argument can be compared with the address of such a predefined static variable.

17 These special constants also cause an exception with the usage of Fortran INTENT: with
 18 USE mpi_f08, the attributes INTENT(IN), INTENT(OUT), and INTENT(INOUT) are used in the
 19 Fortran interface. In most cases, INTENT(IN) is used if the C interface uses call-by-value.
 20 For all buffer arguments and for dummy arguments that may be modified and allow one of
 21 these special constants as input, an INTENT is not specified.
 22

23 19.1.15 Fortran Derived Types

24 MPI supports passing Fortran entities of BIND(C) and SEQUENCE derived types to choice
 25 dummy arguments, provided no type component has the ALLOCATABLE or POINTER attribute.
 26

27 The following code fragment shows some possible ways to send scalars or arrays of
 28 interoperable derived types in Fortran. The example assumes that all data is passed by
 29 address.

```

30 type, BIND(C) :: mytype
31   integer :: i
32   real :: x
33   double precision :: d
34   logical :: l
35 end type mytype

36 type(mytype) :: foo, fooarr(5)
37 integer :: blocklen(4), dtype(4)
38 integer(KIND=MPI_ADDRESS_KIND) :: disp(4), base, lb, extent

39 call MPI_GET_ADDRESS(foo%i, disp(1), ierr)
40 call MPI_GET_ADDRESS(foo%x, disp(2), ierr)
41 call MPI_GET_ADDRESS(foo%d, disp(3), ierr)
42 call MPI_GET_ADDRESS(foo%l, disp(4), ierr)

43
44 base = disp(1)
45 disp(1) = disp(1) - base
46 disp(2) = disp(2) - base
47 disp(3) = disp(3) - base
48 disp(4) = disp(4) - base

```

```

blocklen(1) = 1
blocklen(2) = 1
blocklen(3) = 1
blocklen(4) = 1

dtype(1) = MPI_INTEGER
dtype(2) = MPI_REAL
dtype(3) = MPI_DOUBLE_PRECISION
dtype(4) = MPI_LOGICAL

call MPI_TYPE_CREATE_STRUCT(4, blocklen, disp, dtype, 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 nonstandardized extension such as `!$PRAGMA IGNORE_TKR`; see Section 19.1.6.

To use a derived type in an array requires a correct extent of the datatype handle 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 subsequent 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 5.1.6, 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`.

Table 19.2: Occurrence of Fortran optimization problems in several usage areas

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

19.1.16 Optimization Problems, an Overview

MPI provides operations that may be hidden from the user code and run concurrently with it, accessing the same memory as user code. Examples include the data transfer for an `MPI_IRecv`. The optimizer of a compiler will assume that it can recognize periods when a copy of a variable can be kept in a register without reloading from or storing to memory. When the user code is working with a register copy of some variable while the hidden operation reads or writes the memory copy, problems occur. These problems are independent of the Fortran support method; i.e., they occur with the `mpi_f08` module, the `mpi` module, and the `mpif.h` include file.

This section shows four problematic usage areas (the abbreviations in parentheses are used in the table below):

- Use of nonblocking routines or persistent requests (*Nonbl.*).
- Use of one-sided routines (*1-sided*).
- Use of MPI parallel file I/O split collective operations (*Split*).
- Use of `MPI_BOTTOM` together with absolute displacements in MPI datatypes, or relative displacements between two variables in such datatypes (*Bottom*).

The following compiler optimization strategies (valid for serial code) may cause problems in MPI applications:

- Code movement and register optimization problems; see Section 19.1.17.
- Temporary data movement and temporary memory modifications; see Section 19.1.18.
- Permanent data movement (e.g., through garbage collection); see Section 19.1.19.

Table 19.2 shows the only usage areas where these optimization problems may occur.

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 791–795,
- to minimize the drawbacks on compiler based optimization, and
- to minimize the requirements defined in Section 19.1.7.

19.1.17 Problems with Code Movement and Register Optimization

Nonblocking Operations

If a variable is local to a Fortran subroutine (i.e., not in a module or a COMMON block), the compiler will assume that it cannot be modified by a called subroutine unless it is an actual argument of the call. In the most common linkage convention, the subroutine is expected to save and restore certain registers. Thus, the optimizer will assume that a register that held a valid copy of such a variable before the call will still hold a valid copy on return.

Example 19.1. Fortran 90 register optimization—extreme.

Source	compiled as	or compiled as
REAL :: buf, b1	REAL :: buf, b1	REAL :: buf, b1
call MPI_Irecv(buf,..req)	call MPI_Irecv(buf,..req) register = buf	call MPI_Irecv(buf,..req) b1 = buf
call MPI_Wait(req,..)	call MPI_Wait(req,..)	call MPI_Wait(req,..)
b1 = buf	b1 = register	

Example 19.1 shows extreme, but allowed, possibilities. MPI_WAIT on a concurrent thread modifies buf between the invocation of MPI_Irecv and the completion of MPI_WAIT. But the compiler cannot see any possibility that buf can be changed after MPI_Irecv has returned, and may schedule the load of buf earlier than typed in the source. The compiler has no reason to avoid using a register to hold buf across the call to MPI_WAIT. It also may reorder the instructions as illustrated in the rightmost column.

Example 19.2. Similar example with MPI_ISEND

Source	compiled as	with a possible MPIinternal 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		

Due to valid compiler code movement optimizations in Example 19.2, the content of buf may already have been overwritten by the compiler when the content of buf is sent. The code movement is permitted because the compiler cannot detect a possible access to buf in MPI_WAIT (or in a second thread between the start of MPI_Isend and the end of MPI_WAIT).

Such register optimization is based on moving code; here, the access to buf was moved from after MPI_WAIT to before MPI_WAIT. Note that code movement may also occur across subroutine boundaries when subroutines or functions are inlined.

This register optimization/code movement problem for nonblocking operations does not occur with MPI parallel file I/O split collective operations, because in the MPI_XXX_BEGIN and MPI_XXX_END calls, the same buffer has to be provided as an actual argument. The register optimization / code movement problem for MPI_BOTTOM and derived MPI datatypes may occur in each blocking and nonblocking communication call, as well as in each parallel file I/O operation.

Persistent Operations

With persistent requests, the buffer argument is hidden from the `MPI_START` and `MPI_STARTALL` calls, i.e., the Fortran compiler may move buffer accesses across the `MPI_START` or `MPI_STARTALL` call, similar to the `MPI_WAIT` call as described in the Nonblocking Operations subsection in Section 19.1.17.

One-sided Communication

An example with instruction reordering due to register optimization can be found in Section 12.7.4.

MPI_BOTTOM and Combining Independent Variables in Datatypes

This section is only relevant if the MPI program uses a buffer argument to an `MPI_SEND`, `MPI_RECV`, etc., that hides the actual variables involved in the communication. `MPI_BOTTOM` with an `MPI_Datatype` containing *absolute addresses* is one example. Creating a datatype that uses one variable as an anchor and brings along others by using `MPI_GET_ADDRESS` to determine their offsets from the anchor is another. The anchor variable would be the only one referenced in the call. Also attention must be paid if MPI operations are used that run in parallel with the user's application.

Example 19.3 shows what Fortran compilers are allowed to do.

Example 19.3. Fortran 90 register optimization.

This source	can be compiled as
<code>call MPI_GET_ADDRESS(buf, bufaddr, ierror)</code>	<code>call MPI_GET_ADDRESS(buf, ...)</code>
<code>call MPI_TYPE_CREATE_STRUCT(1, 1, bufaddr, MPI_REAL, dtype, ierror)</code>	<code>call MPI_TYPE_CREATE_STRUCT(...)</code>
<code>call MPI_TYPE_COMMIT(dtype, ierror)</code>	<code>call MPI_TYPE_COMMIT(...)</code>
<code>val_old = buf</code>	<code>register = buf</code>
	<code>val_old = register</code>
<code>call MPI_RECV(MPI_BOTTOM, 1, dtype, ...)</code>	<code>call MPI_RECV(MPI_BOTTOM, ...)</code>
<code>val_new = buf</code>	<code>val_new = register</code>

In Example 19.3, the compiler does not invalidate the register because it cannot see that `MPI_RECV` changes the value of `buf`. The access to `buf` is hidden by the use of `MPI_GET_ADDRESS` and `MPI_BOTTOM`.

Example 19.4. Similar example with MPI_SEND

This source	can be compiled as
<code>! buf contains val_old</code>	<code>! buf contains val_old</code>
<code>buf = val_new</code>	
<code>call MPI_SEND(MPI_BOTTOM, 1, dtype, ...)</code>	<code>call MPI_SEND(...)</code>
<code>! with buf as a displacement in dtype</code>	<code>! i.e., val_old is sent</code>
	<code>!</code>
	<code>! buf=val_new is moved to here</code>
	<code>! and detected as dead code</code>
	<code>! and therefore removed</code>
	<code>!</code>

```
buf = val_overwrite
```

```
buf = val_overwrite
```

In Example 19.4, several successive assignments to the same variable `buf` can be combined in a way such that only the last assignment is executed. “Successive” means that no interfering load access to this variable occurs between the assignments. The compiler cannot detect that the call to `MPI_SEND` statement is interfering because the load access to `buf` is hidden by the usage of `MPI_BOTTOM`.

Solutions

The following sections show in detail how the problems with code movement and register optimization can be portably solved. Application writers can partially or fully avoid these compiler optimization problems by using one or more of the special Fortran declarations 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 19.5. Protecting nonblocking communication with the `ASYNCHRONOUS` attribute.

```
USE mpi_f08
REAL, ASYNCHRONOUS :: b(0:101) ! elements 0 and 101 are halo cells
REAL :: bnew(0:101)           ! elements 1 and 100 are newly computed
TYPE(MPI_Request) :: req(4)
INTEGER :: left, right, i
CALL MPI_Cart_shift(...,left,right,...)
CALL MPI_Irecv(b( 0), ..., left, ..., req(1), ...)
CALL MPI_Irecv(b(101), ..., right, ..., req(2), ...)
CALL MPI_Isend(b( 1), ..., left, ..., req(3), ...)
CALL MPI_Isend(b(100), ..., right, ..., req(4), ...)

#ifdef WITHOUT_OVERLAPPING_COMMUNICATION_AND_COMPUTATION
! Case (a)
CALL MPI_Waitall(4, req, ...)
DO i=1,100 ! compute all new local data
  bnew(i) = function(b(i-1), b(i), b(i+1))
END DO
#endif

#ifdef WITH_OVERLAPPING_COMMUNICATION_AND_COMPUTATION
! Case (b)
DO i=2,99 ! compute only elements for which halo data is not needed
  bnew(i) = function(b(i-1), b(i), b(i+1))
END DO
CALL MPI_Waitall(4, req, ...)
```

```

1  i=1 ! compute leftmost element
2  bnew(i) = function(b(i-1), b(i), b(i+1))
3  i=100 ! compute rightmost element
4  bnew(i) = function(b(i-1), b(i), b(i+1))
5  #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 19.1.7. 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 19.1.6), then the compiler has to guarantee call by reference and should report a compile-time error if call by reference is impossible, e.g., if vector subscripts are used. The `MPI_ASYNC_PROTECTS_NONBLOCKING` is set to `.TRUE.` if both the protection of the actual buffer argument through `ASYNCHRONOUS` according to the TS 29113 extension and the declaration of the dummy argument with `ASYNCHRONOUS` in the Fortran support method is guaranteed for all nonblocking routines, otherwise it is set to `.FALSE.`

The `ASYNCHRONOUS` attribute has some restrictions. Section 5.4.2 of the TS 29113 specifies:

“Asynchronous communication for a Fortran variable occurs through the action of procedures defined by means other than Fortran. It is initiated by execution of an asynchronous communication initiation procedure and completed by execution of an asynchronous communication completion procedure. Between the execution of the initiation and completion procedures, any variable of which any part is associated with any part of the asynchronous communication variable is a pending communication affector. Whether a procedure is an asynchronous communication initiation or completion procedure is processor dependent.

Asynchronous communication is either input communication or output communication. For input communication, a pending communication affector shall

not be referenced, become defined, become undefined, become associated with a dummy argument that has the VALUE attribute, or have its pointer association status changed. For output communication, a pending communication affector shall not be redefined, become undefined, or have its pointer association status changed.”

In Example 19.5 Case (a) on page 791, 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_IXXX` 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 that are passed through different dummy arguments into a subroutine, as shown in Example 19.9.

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 19.3 and Example 19.4, 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 19.1.8.

- The problems illustrated by the Examples 19.1 and 19.2 can be solved by calling `MPI_F_SYNC_REG(buf)` once immediately after `MPI_WAIT`.

Example 19.1

can be solved with

```
call MPI_IRecv(buf, .. req)

call MPI_WAIT(req, ..)
call MPI_F_SYNC_REG(buf)
b1 = buf
```

Example 19.2

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 19.3 and 19.4 can be solved with two additional `MPI_F_SYNC_REG(buf)` statements; one directly before `MPI_RECV/MPI_SEND`, and one directly after this communication operation.

Example 19.3
can be solved with

```
call MPI_F_SYNC_REG(buf)
call MPI_RECV(MPI_BOTTOM, ...)
call MPI_F_SYNC_REG(buf)
```

Example 19.4
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/MPI_SEND`.

- In the example in Section 12.7.4, two asynchronous accesses must be protected: in Process 1, the access to `bbbb` must be protected similar to Example 19.1, i.e., a call to `MPI_F_SYNC_REG(bbbb)` is needed after the second `MPI_WIN_FENCE` to guarantee that further accesses to `bbbb` are not moved ahead of the call to `MPI_WIN_FENCE`. In Process 2, both calls to `MPI_WIN_FENCE` together act as a communication call with `MPI_BOTTOM` as the buffer. That is, before the first fence and after the second fence, a call to `MPI_F_SYNC_REG(buff)` is needed to guarantee that accesses to `buff` are not moved after or ahead of the calls to `MPI_WIN_FENCE`. Using `MPI_GET` instead of `MPI_PUT`, the same calls to `MPI_F_SYNC_REG` are necessary.

Source of Process 1

```
bbbb = 777

call MPI_WIN_FENCE
call MPI_PUT(bbbb
into buff of process 2)

call MPI_WIN_FENCE
call MPI_F_SYNC_REG(bbbb)
```

Source of Process 2

```
buff = 999
call MPI_F_SYNC_REG(buff)
call MPI_WIN_FENCE

call MPI_WIN_FENCE
call MPI_F_SYNC_REG(buff)
ccc = buff
```

- The temporary memory modification problem, i.e., Example 19.6, can **not** be solved with this method.

A User Defined Routine Instead of `MPI_F_SYNC_REG`

Instead of `MPI_F_SYNC_REG`, one can also use a user defined external subroutine, which is separately compiled:

```
subroutine DD(buf)
  integer buf
end
```

Note that if the `INTENT` is declared in an explicit interface for the external subroutine, it must be `OUT` or `INOUT`. The subroutine itself may have an empty body, but the compiler does not know this and has to assume that the buffer may be altered. For example, a call to `MPI_RECV` with `MPI_BOTTOM` as buffer might be replaced by

```

call DD(buf)
call MPI_RECV(MPI_BOTTOM, ...)
call DD(buf)

```

Such a user-defined routine was introduced in MPI-2.0 and is still included here to document such usage in existing application programs although new applications should prefer MPI_F_SYNC_REG or one of the other possibilities. In an existing application, calls to 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 19.1.7.

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.*)

19.1.18 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 19.5, Case (b) on page 791. Example 19.6 also shows a possibility that could be problematic.

Example 19.6. 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 19.7. 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,...)

```

Example 19.8. 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,...)

```

In the compiler-generated, possible optimization in Example 19.7, `buf(100,100)` from Example 19.6 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 loop is temporarily using this part of the buffer. When the `tmp` data is written back to `buf`, the previous data of `buf(1,1:100)` is restored and the received data is lost. The principle behind this optimization is that the receive buffer data `buf(1,1:100)` was temporarily moved

to `tmp`.

Example 19.8 shows a second possible optimization. The whole array is temporarily moved to `local_buf`.

When storing `local_buf` back to the original location `buf`, then this implies overwriting the section of `buf` that serves as a receive buffer in the nonblocking MPI call, i.e., this storing back of `local_buf` is therefore likely to interfere with asynchronously received data in `buf(1,1:100)`.

Note that this problem may also occur:

- With the local buffer at the origin process, between an RMA communication call and the ensuing synchronization call; see Chapter 12.
- With the window buffer at the target process between two ensuing RMA synchronization calls.
- With the local buffer in MPI parallel file I/O split collective operations between the `MPI_XXX_BEGIN` and `MPI_XXX_END` calls; see Section 14.4.5.

As already mentioned in Section [The Fortran ASYNCHRONOUS Attribute](#) on page 792 of Section 19.1.17, the `ASYNCHRONOUS` attribute can prevent compiler optimization with temporary data movement, but only if the receive buffer and the local references are separated into different variables, as shown in Example 19.9 and in Example 19.10.

Note also that the methods

- calling `MPI_F_SYNC_REG` (or such a user-defined routine),
- using module variables and `COMMON` blocks, and
- the `TARGET` attribute

cannot be used to prevent such temporary data movement. These methods influence compiler optimization when library routines are called. They cannot prevent the optimizations of the code fragments shown in Example 19.6 and 19.7.

Note also that compiler optimization with temporary data movement should **not** be prevented by declaring `buf` as `VOLATILE` because the `VOLATILE` implies that all accesses to any storage unit (word) of `buf` must be directly done in the main memory exactly in the sequence defined by the application program. The `VOLATILE` attribute prevents all register and cache optimizations. Therefore, `VOLATILE` may cause a huge performance degradation.

Instead of solving the problem, it is better to **prevent** the problem: when overlapping communication and computation, the nonblocking communication (or nonblocking or split collective I/O) and the computation should be executed **on different variables**, and the communication should be *protected* with the `ASYNCHRONOUS` attribute. In this case, the temporary memory modifications are done only on the variables used in the computation and cannot have any side effect on the data used in the nonblocking MPI operations.

Rationale. This is a strong restriction for application programs. To weaken this 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 19.9 (which is a solution for the problem shown in Example 19.5 and in Example 19.10 (which is a solution for the problem shown in Example 19.8), 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 nonblocking communication only with a TS 29113 based MPI library.

19.1.19 Permanent Data Movement

A Fortran compiler may implement permanent data movement during the execution of a Fortran program. This would require that pointers to such data are appropriately updated. An implementation with automatic garbage collection is one use case. Such permanent data movement is in conflict with MPI in several areas:

- MPI datatype handles with absolute addresses in combination with `MPI_BOTTOM`.
- All nonblocking MPI operations if the internally used pointers to the buffers are not updated by the Fortran runtime, or if within an MPI process, the data movement is executed in parallel with the MPI operation.

This problem can be also solved by using the `ASYNCHRONOUS` attribute for such buffers. This MPI standard requires that the problems with permanent data movement do not occur by imposing suitable restrictions on the MPI library together with the compiler used; see Section 19.1.7.

Example 19.9. Using separated variables for overlapping communication and computation to allow the protection of nonblocking communication with the `ASYNCHRONOUS` attribute.

```

28 USE mpi_f08
29 REAL :: b(0:101)      ! elements 0 and 101 are halo cells
30 REAL :: bnew(0:101)  ! elements 1 and 100 are newly computed
31 INTEGER :: i
32 CALL separated_sections(b(0), b(1:100), b(101), bnew(0:101))
33 i=1 ! compute leftmost element
34   bnew(i) = function(b(i-1), b(i), b(i+1))
35 i=100 ! compute rightmost element
36   bnew(i) = function(b(i-1), b(i), b(i+1))
37 END
38
39 SUBROUTINE separated_sections(b_lefthalo, b_inner, b_righthalo, bnew)
40 USE mpi_f08
41 REAL, ASYNCHRONOUS :: b_lefthalo(0:0), b_inner(1:100), b_righthalo(101:101)
42 REAL :: bnew(0:101) ! elements 1 and 100 are newly computed
43 TYPE(MPI_Request) :: req(4)
44 INTEGER :: left, right, i
45 CALL MPI_Cart_shift(...,left, right,...)
46 CALL MPI_Irecv(b_lefthalo( 0), ..., left, ..., req(1), ...)
47 CALL MPI_Irecv(b_righthalo(101), ..., right, ..., req(2), ...)
48 ! b_lefthalo and b_righthalo is written asynchronously.
49 ! There is no other concurrent access to b_lefthalo and b_righthalo.
50 CALL MPI_Isend(b_inner( 1), ..., left, ..., req(3), ...)
51 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

```

19.1.20 Comparison with C

In C, subroutines that 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 that 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 19.10. 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,...)

```

19.2 Support for Large Count and Large Byte Displacement in MPI Language Bindings

The following types, which were used prior to MPI-4.0, have been deemed too small to hold values that applications wish to use:

- The C `int` type and the Fortran `INTEGER` type were used for *count* parameters.
- The C `int` type and the Fortran `INTEGER` type were used for some parameters that represent *byte displacement* in memory.
- The C `MPI_Aint` type and the Fortran `INTEGER(KIND=MPI_ADDRESS_KIND)` type were used for some parameters that represent *byte displacement* in files (e.g., in constructors of MPI datatypes that can be used with files).

In order to avoid breaking backwards compatibility, this version of MPI supports larger types via separate additional MPI procedures in C (suffixed with “_c”) and via interface polymorphism in Fortran when using `USE mpi_f08`. For better readability, all Fortran large count procedure declarations are marked with a comment “!(`_c`)”. No polymorphic support for larger types is provided in Fortran when using `mpif.h` and `use mpi`.

For the large count versions of three datatype constructors, `MPI_TYPE_CREATE_HINDEXED`, `MPI_TYPE_CREATE_HINDEXED_BLOCK`, and `MPI_TYPE_CREATE_STRUCT`, absolute addresses shall not be used to specify byte displacements since the parameter is of type `MPI_COUNT` instead of type `MPI_AINT` (see Section 2.5.8).

In addition, the functions `MPI_TYPE_GET_ENVELOPE` and `MPI_TYPE_GET_CONTENTS` also support large count types via *additional parameters* in separate additional MPI procedures in C (suffixed with “_c”) and interface polymorphism in Fortran when using `USE mpi_f08` (see Section 5.1.13).

Further, the callbacks of type `MPI_User_function` and `MPI_Datarep_conversion_function` also support large count types via separate additional callback prototypes in C (suffixed with “_c”) and multiple abstract interfaces in Fortran when using `USE mpi_f08` (see Sections 6.9.5 and 14.5.3, respectively). An additional large count predefined callback function `MPI_CONVERSION_FN_NULL_C` is provided within each of these two language bindings.

In C bindings, for each MPI procedure that had at least one *count* or *byte displacement* parameter that used the `int` and/or `MPI_Aint` types prior to MPI-4.0, an additional MPI procedure is provided, with the same name but suffixed by “_c”. The MPI procedure without the “_c” token has the same name and parameter types as versions prior to MPI-4.0. The “_c” suffixed MPI procedure has `MPI_Count` for all *count* parameters, `MPI_Aint` for parameters that represent *byte displacement* in memory, `MPI_Offset` for parameters that represent *byte displacement* in files, and `MPI_Count` for parameters that may represent *byte displacement* in both memory and files.

In Fortran, when using `USE mpi_f08`, for each MPI procedure that had at least one *count* or *byte displacement* parameter that used the `INTEGER` or `INTEGER(KIND=MPI_ADDRESS_KIND)` types prior to MPI-4.0, a polymorphic interface containing two specific procedures is provided. One of the specific procedures has the same name and dummy parameter types as in versions prior to MPI-4.0. `INTEGER` and/or `INTEGER(KIND=MPI_ADDRESS_KIND)` for *count* and *byte displacement* parameters. The other specific procedure has the same name followed by “_c”, and then suffixed by the token specified in Table 19.1 for `USE mpi_f08`.

It also has `INTEGER(KIND=MPI_COUNT_KIND)` for all *count* parameters, `INTEGER(KIND=MPI_ADDRESS_KIND)` for parameters that represent *byte displacement* in memory, `INTEGER(KIND=MPI_OFFSET_KIND)` for parameters that represent *byte displacement* in files, and `INTEGER(KIND=MPI_COUNT_KIND)` for parameters that may represent *byte displacement* in both memory and files (for more details on specific Fortran procedure names and related calling conventions, refer to Table 19.1 in Section 19.1.5). There is one exception: if the type signatures of the two specific procedures are identical (e.g., if `INTEGER(KIND=MPI_COUNT_KIND)` is the same type as `INTEGER(KIND=MPI_ADDRESS_KIND)`), then the implementation shall not provide the “_c” specific procedure.

It is erroneous to directly invoke the “_c” specific procedures in the Fortran `mpi_f08` module with the exception of the following procedures: `MPI_Op_create_c` and `MPI_Register_datarep_c`.

In older Fortran bindings (`mpif.h` and `use mpi`), no new interfaces and no new specific procedures for larger types are provided beyond what existed in MPI-3.1; all MPI procedures have the same types as in the versions prior to MPI-4.0.

19.3 Language Interoperability

19.3.1 Introduction

It is not uncommon for library developers to use one language to develop an application library that may be called by an application program written in a different language. MPI currently supports ISO (previously ANSI) C and Fortran bindings. It should be possible for applications in any of the supported languages to call MPI-related functions in another language.

Moreover, MPI allows the development of client-server code, with MPI communication used between a parallel client and a parallel server. It should be possible to code the server in one language and the clients in another language. To do so, communications should be possible between applications written in different languages.

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.

19.3.2 Assumptions

We assume that conventions exist for programs written in one language to call routines written in another language. These conventions specify how to link routines in different languages into one program, how to call functions in a different language, how to pass arguments between languages, and the correspondence between basic datatypes in different

1 languages. In general, these conventions will be implementation dependent. Furthermore,
2 not every basic datatype may have a matching type in other languages. For example,
3 C character strings may not be compatible with Fortran CHARACTER variables. However,
4 we assume that a Fortran INTEGER, as well as a (sequence associated) Fortran array of
5 INTEGERS, can be passed to a C program. We also assume that Fortran and C have address-
6 sized integers. This does not mean that the default-size integers are the same size as
7 default-sized pointers, but only that there is some way to hold (and pass) a C address in
8 a Fortran integer. It is also assumed that INTEGER(KIND=MPI_OFFSET_KIND) can be passed
9 from Fortran to C as MPI_Offset.

11 19.3.3 Initialization

12 Two approaches are available for initializing MPI: the World Model(Section 11.2) , and the
13 Sessions Model(Section 11.3).

15 Concerns specific to the World Model

16 A call to MPI_INIT or MPI_INIT_THREAD, from any language, initializes MPI for execution
17 in all languages.

18
19
20 *Advice to users.* Certain implementations use the (inout) argc, argv arguments
21 of the C version of MPI_INIT in order to propagate values for argc and argv to all
22 executing processes. Use of the Fortran version of MPI_INIT to initialize MPI may
23 result in a loss of this ability. (*End of advice to users.*)

24
25 The function MPI_INITIALIZED returns the same answer in all languages.

26 The function MPI_FINALIZE finalizes the MPI environments for all languages.

27 The function MPI_FINALIZED returns the same answer in all languages.

28 The MPI environment is initialized in the same manner for all languages by
29 MPI_INIT. E.g., MPI_COMM_WORLD carries the same information regardless of language:
30 same processes, same environmental attributes, same error handlers.

31
32 *Advice to users.* The use of several languages in one MPI program may require the
33 use of special options at compile and/or link time. (*End of advice to users.*)

34
35 *Advice to implementors.* Implementations may selectively link language specific
36 MPI libraries only to codes that need them, so as not to increase the size of binaries
37 for codes that use only one language. The MPI initialization code needs to perform
38 initialization for a language only if that language library is loaded. (*End of advice to
39 implementors.*)

41 Concerns specific to the Sessions Model

42 A call to MPI_SESSION_INIT from any language initializes a session that can be used from
43 all languages.

44 A call to MPI_SESSION_FINALIZE from any language finalizes the session for all lan-
45 guages.

Concerns common to both the World Model and the Sessions Model

The function `MPI_ABORT` kills MPI processes in the group of the supplied communicator, irrespective of the language used by the caller or by the MPI processes killed.

Information can be added to info objects in one language and retrieved in another.

19.3.4 Transfer of Handles

Handles are passed between Fortran and C by using an explicit C wrapper to convert Fortran handles to C handles. There is no direct access to C handles in Fortran.

The type definition `MPI_Fint` is provided in C for an integer of the size that matches a 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)
```

```

1 MPI_Fint MPI_Op_c2f(MPI_Op op)
2
3 MPI_Info MPI_Info_f2c(MPI_Fint info)
4
5 MPI_Fint MPI_Info_c2f(MPI_Info info)
6
7 MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)
8
9 MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)
10
11 MPI_Message MPI_Message_f2c(MPI_Fint message)
12
13 MPI_Session MPI_Session_f2c(MPI_Fint session)
14
15 MPI_Fint MPI_Session_c2f(MPI_Session session)

```

Example 19.11. The example below illustrates how the Fortran MPI function `MPI_TYPE_COMMIT` can be implemented by wrapping the C MPI function `MPI_Type_commit` with a C wrapper to do handle conversions. In this example a Fortran-C interface is assumed where a Fortran function is all upper case when referred to from C and arguments are passed by addresses.

```

21 ! FORTRAN PROCEDURE
22 SUBROUTINE MPI_TYPE_COMMIT(DATATYPE, IERR)
23   INTEGER :: DATATYPE, IERR
24   CALL MPI_X_TYPE_COMMIT(DATATYPE, IERR)
25   RETURN
26   END
27
28 /* C wrapper */
29
30 void MPI_X_TYPE_COMMIT(MPI_Fint *f_handle, MPI_Fint *ierr)
31 {
32     MPI_Datatype datatype;
33
34     datatype = MPI_Type_f2c(*f_handle);
35     *ierr = (MPI_Fint)MPI_Type_commit(&datatype);
36     *f_handle = MPI_Type_c2f(datatype);
37     return;
38 }

```

The same approach can be used for all other MPI functions. The call to `MPI_XXX_f2c` (resp. `MPI_XXX_c2f`) can be omitted when the handle is an OUT (resp. IN) argument, rather than INOUT.

Rationale. The design here provides a convenient solution for the prevalent case, where a C wrapper is used to allow Fortran code to call a C library, or C code to call a Fortran library. The use of C wrappers is much more likely than the use of Fortran wrappers, because it is much more likely that a variable of type `INTEGER` can be passed to C, than a C handle can be passed to Fortran.

Returning the converted value as a function value rather than through the argument list allows the generation of efficient inlined code when these functions are simple

(e.g., the identity). The conversion function in the wrapper does not catch an invalid handle argument. Instead, an invalid handle is passed below to the library function, which, presumably, checks its input arguments. (*End of rationale.*)

19.3.5 Status

The following two procedures are provided in C to convert from a Fortran (with the `mpi` module or `mpif.h`) status (which is an array of integers) to a C status (which is a structure), and vice versa. The conversion occurs on all the information in status, including that which is hidden. That is, no status information is lost in the conversion.

```
int MPI_Status_f2c(const MPI_Fint *f_status, MPI_Status *c_status)
```

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.

In C, such an `f_status` array can be defined with `MPI_Fint f_status[MPI_F_STATUS_SIZE]`. Within this array, one can use in C the indexes `MPI_F_SOURCE`, `MPI_F_TAG`, and `MPI_F_ERROR`, to access the same elements as in Fortran with `MPI_SOURCE`, `MPI_TAG` and `MPI_ERROR`. The C indexes are 1 less than the corresponding indexes in Fortran due to the different default array start indexes in both languages.

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 19.1. (*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 19.1 illustrates all status conversion routines. Some are only available in C, some in both C and the Fortran `mpi` and `mpi_f08` interfaces (but not in the `mpif.h` interface).

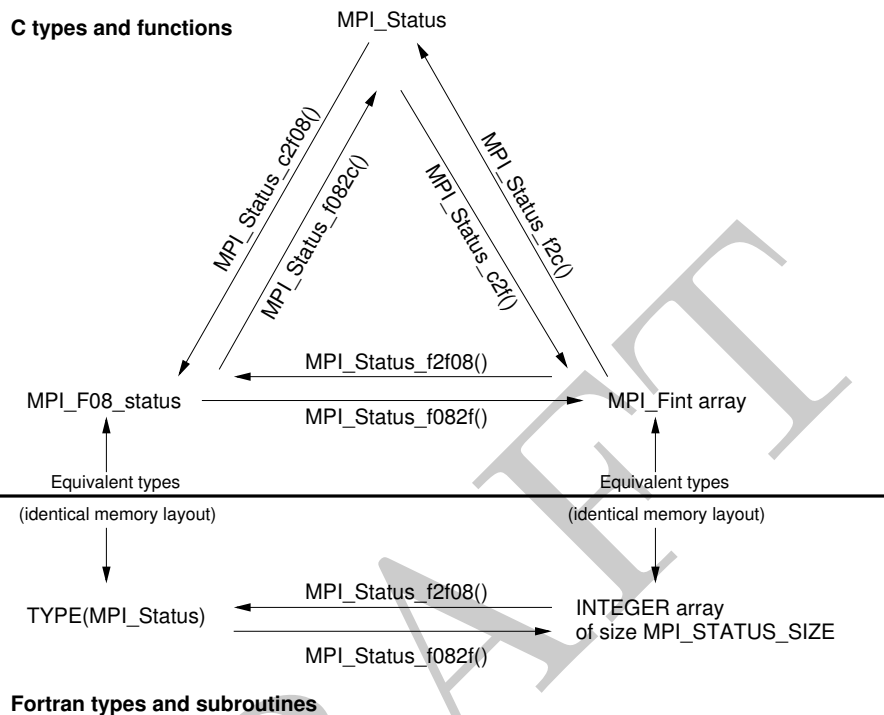


Figure 19.1: Status conversion routines

```
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 (status)

OUT `f08_status` status object declared as named type (status)

C binding

```
int MPI_Status_f2f08(const MPI_Fint *f_status, MPI_F08_status *f08_status)
```

Fortran 2008 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
```

Fortran binding (the following procedure is not available with mpif.h)

```
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 (status)
OUT	f_status	status object declared as array (status)

C binding

```
int MPI_Status_f082f(const MPI_F08_status *f08_status, MPI_Fint *f_status)
```

Fortran 2008 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
```

Fortran binding (the following procedure is not available with mpif.h)

```
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.

19.3.6 MPI Opaque Objects

Unless said otherwise, opaque objects are “the same” in all languages: they carry the same information, and have the same meaning in both languages. The mechanism described in the previous section can be used to pass references to MPI objects from language to language. An object created in one language can be accessed, modified or freed in another language.

We examine below in more detail issues that arise for each type of MPI object.

Datatypes

Datatypes encode the same information in all languages. E.g., a datatype accessor like `MPI_TYPE_GET_EXTENT` will return the same information in all languages. If a datatype defined in one language is used for a communication call in another language, then the message sent will be identical to the message that would be sent from the first language: the same communication buffer is accessed, and the same representation conversion is performed, if needed. All predefined datatypes can be used in datatype constructors in any language. If a datatype is committed, it can be used for communication in any language.

The function `MPI_GET_ADDRESS` returns the same value in all languages. Note that we do not require that the constant `MPI_BOTTOM` have the same value in all languages (see Section 19.3.9).

Example 19.12.

```

14 ! FORTRAN CODE
15 REAL :: R(5)
16 INTEGER :: DTYPE, IERR, AOBLLEN(1), AOTYPE(1)
17 INTEGER(KIND=MPI_ADDRESS_KIND) :: AODISP(1)
18
19 ! create an absolute datatype for array R
20 AOBLLEN(1) = 5
21 CALL MPI_GET_ADDRESS(R, AODISP(1), IERR)
22 AOTYPE(1) = MPI_REAL
23 CALL MPI_TYPE_CREATE_STRUCT(1, AOBLLEN, AODISP, AOTYPE, DTYPE, IERR)
24 CALL C_ROUTINE(DTYPE)
25
26 /* C code */
27 void C_ROUTINE(MPI_Fint *fctype)
28 {
29     int count = 5;
30     int lens[2] = {1,1};
31     MPI_Aint displs[2];
32     MPI_Datatype types[2], newtype;
33
34     /* create an absolute datatype for buffer that consists */
35     /* of count, followed by R(5) */
36
37     MPI_Get_address(&count, &displs[0]);
38     displs[1] = 0;
39     types[0] = MPI_INT;
40     types[1] = MPI_Type_f2c(*fctype);
41     MPI_Type_create_struct(2, lens, displs, types, &newtype);
42     MPI_Type_commit(&newtype);
43
44     MPI_Send(MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
45     /* the message sent contains an int count of 5, followed */
46     /* by the 5 REAL entries of the Fortran array R. */
47 }

```

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, files, windows, and sessions; 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 348. (*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 6.9.2 can be used in the listed predefined operations independent of the programming language from which the MPI routine is called.

Advice to users. Reduce operations receive as one of their arguments the datatype of the operands. Thus, one can define “polymorphic” reduce operations that work for C and Fortran datatypes. (*End of advice to users.*)

19.3.7 Attributes

Attribute keys can be allocated in one language and freed in another. Similarly, attribute values can be set in one language and accessed in another. To achieve this, attribute keys will be allocated in an integer range that is valid all languages. The same holds true for system-defined attribute values (such as `MPI_TAG_UB`, `MPI_WTIME_IS_GLOBAL`, etc.).

Attribute keys declared in one language are associated with copy and delete functions in that language (the functions provided by the `MPI_XXX_CREATE_KEYVAL` call). When a communicator is duplicated, for each attribute, the corresponding copy function is called, using the right calling convention for the language of that function; and similarly, for the delete callback function.

Advice to implementors. This requires that attributes be tagged either as “C” or “Fortran” and that the language tag be checked in order to use the right calling convention for the callback function. (*End of advice to implementors.*)

The attribute manipulation functions described in Section 7.7 defines attributes arguments to be of type `void*` in C, and of type `INTEGER`, in Fortran. On some systems, `INTEGER`s 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 `INTEGER`s are smaller, then the (deprecated) Fortran function `MPI_ATTR_GET` will return the least significant part of the attribute word; the (deprecated) Fortran function `MPI_ATTR_PUT` will set the least significant part of the attribute word, which will be sign extended to the entire word. (These two functions may be invoked explicitly by user code, or implicitly, by attribute copying callback functions.)

As for addresses, new functions are provided that manipulate Fortran address sized attributes, and have the same functionality as the old functions in C. These functions are described in Section 7.7. Users are encouraged to use these new functions.

MPI supports two types of attributes: address-valued (pointer) attributes, and 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 19.13.

A. Setting an attribute value in C

```
int set_val = 3;
struct foo set_struct;

/* Set a value that is a pointer to an int */

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 19.14. A. Setting an attribute value with the (deprecated) Fortran MPI-1 call

```

1  INTEGER IERR, VAL
2  VAL = 7
3  CALL MPI_ATTR_PUT(MPI_COMM_WORLD, KEYVAL, VAL, IERR)
4
5

```

B. Reading the attribute value in C

```

7  int flag;
8  int *value;
9
10 /* Upon successful return, value points to internal MPI storage and
11    *value == (int) 7 */
12 MPI_Comm_get_attr(MPI_COMM_WORLD, keyval, &value, &flag);
13

```

C. Reading the attribute value with (deprecated) Fortran MPI-1 calls

```

14 LOGICAL FLAG
15 INTEGER IERR, VALUE
16
17 ! Upon successful return, VALUE == 7
18 CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR)
19

```

D. Reading the attribute value with Fortran MPI-2 calls

```

20
21 LOGICAL FLAG
22 INTEGER IERR
23 INTEGER(KIND=MPI_ADDRESS_KIND) VALUE
24
25 ! Upon successful return, VALUE == 7 (sign extended)
26 CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR)
27

```

Example 19.15. A. Setting an attribute value via a Fortran MPI-2 call

```

28
29 INTEGER IERR
30 INTEGER(KIND=MPI_ADDRESS_KIND) VALUE1
31 INTEGER(KIND=MPI_ADDRESS_KIND) VALUE2
32 VALUE1 = 42
33 VALUE2 = INT(2, KIND=MPI_ADDRESS_KIND) ** 40
34
35 CALL MPI_COMM_SET_ATTR(MPI_COMM_WORLD, KEYVAL1, VALUE1, IERR)
36 CALL MPI_COMM_SET_ATTR(MPI_COMM_WORLD, KEYVAL2, VALUE2, IERR)
37

```

B. Reading the attribute value in C

```

38 int flag;
39 MPI_Aint *value1, *value2;
40
41 /* Upon successful return, value1 points to internal MPI storage and
42    *value1 == 42 */
43 MPI_Comm_get_attr(MPI_COMM_WORLD, keyval1, &value1, &flag);
44 /* Upon successful return, value2 points to internal MPI storage and
45    *value2 == 2^40 */
46 MPI_Comm_get_attr(MPI_COMM_WORLD, keyval2, &value2, &flag);
47

```

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.*)

19.3.8 Extra-State

Extra-state should not be modified by the copy or delete callback functions. (This is obvious from the C binding, but not obvious from the Fortran binding). However, these functions may update state that is indirectly accessed via extra-state. E.g., in C, extra-state can be

a pointer to a data structure that is modified by the copy or callback functions; in Fortran, extra-state can be an index into an entry in a COMMON array that is modified by the copy or callback functions. In a multithreaded environment, users should be aware that distinct threads may invoke the same callback function concurrently: if this function modifies state associated with extra-state, then mutual exclusion code must be used to protect updates and accesses to the shared state.

19.3.9 Constants

MPI constants have the same value in all languages, unless specified otherwise. This does not apply to constant handles (MPI_INT, MPI_COMM_WORLD, MPI_ERRORS_RETURN, MPI_SUM, etc.) These handles need to be converted, as explained in Section 19.3.4. Constants that specify maximum lengths of strings (see Section A.1.1 for a listing) have a value one less in Fortran than C since in C the length includes the null terminating character. Thus, these constants represent the amount of space that must be allocated to hold the largest possible such string, rather than the maximum number of printable characters the string could contain.

Advice to users. This definition means that it is safe in C to allocate a buffer to receive a string using a declaration like

```
char name [MPI_MAX_OBJECT_NAME];
```

(*End of advice to users.*)

Also constant “addresses,” i.e., special values for reference arguments that are not handles, such as MPI_BOTTOM or MPI_STATUS_IGNORE may have different values in different languages.

Rationale. The current MPI standard specifies that MPI_BOTTOM can be used in initialization expressions in C, but not in Fortran. Since Fortran does not normally support call by value, then MPI_BOTTOM in Fortran must be the name of a predefined static variable, e.g., a variable in an MPI declared COMMON block. On the other hand, in C, it is natural to take MPI_BOTTOM = 0 (Caveat: Defining MPI_BOTTOM = 0 implies that NULL pointer cannot be distinguished from MPI_BOTTOM; it may be that MPI_BOTTOM = 1 is better. See the advice to implementors in the [Datatypes](#) subsection in Section 19.3.6) Requiring that the Fortran and C values be the same will complicate the initialization process. (*End of rationale.*)

19.3.10 Interlanguage Communication

The type matching rules for communication in MPI are not changed: the datatype specification for each item sent should match, in type signature, the datatype specification used to receive this item (unless one of the types is MPI_PACKED). Also, the type of a message item should match the type declaration for the corresponding communication buffer location, unless the type is MPI_BYTE or MPI_PACKED. Interlanguage communication is allowed if it complies with these rules.

Example 19.16. In the example below, a Fortran array is sent from Fortran and received in C.

```

1  ! FORTRAN CODE
2  SUBROUTINE MYEXAMPLE()
3  USE mpi_f08
4  REAL :: R(5)
5  INTEGER :: IERR, MYRANK, AOBLEN(1)
6  TYPE(MPI_Datatype) :: DTYPE, AOTYPE(1)
7  INTEGER(KIND=MPI_ADDRESS_KIND) :: AODISP(1)
8
9  ! create an absolute datatype for array R
10 AOBLEN(1) = 5
11 CALL MPI_GET_ADDRESS(R, AODISP(1), IERR)
12 AOTYPE(1) = MPI_REAL
13 CALL MPI_TYPE_CREATE_STRUCT(1, AOBLEN, AODISP, AOTYPE, DTYPE, IERR)
14 CALL MPI_TYPE_COMMIT(DTYPE, IERR)
15
16 CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYRANK, IERR)
17 IF (MYRANK .EQ. 0) THEN
18     CALL MPI_SEND(MPI_BOTTOM, 1, DTYPE, 1, 0, MPI_COMM_WORLD, IERR)
19 ELSE
20     CALL C_ROUTINE(DTYPE%MPI_VAL)
21 END IF
22 END SUBROUTINE
23
24 /* C code */
25
26 void C_ROUTINE(MPI_Fint *fhandle)
27 {
28     MPI_Datatype type;
29     MPI_Status status;
30
31     type = MPI_Type_f2c(*fhandle);
32
33     MPI_Recv(MPI_BOTTOM, 1, type, 0, 0, MPI_COMM_WORLD, &status);
34 }

```

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.

DRAFT

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`)

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)

1	Error classes (continued)
2	C type: const int (or unnamed enum)
3	Fortran type: INTEGER
4	<hr/> MPI_ERR_IN_STATUS
5	MPI_ERR_ACCESS
6	MPI_ERR_AMODE
7	MPI_ERR_ASSERT
8	MPI_ERR_BAD_FILE
9	MPI_ERR_BASE
10	MPI_ERR_CONVERSION
11	MPI_ERR_DISP
12	MPI_ERR_DUP_DATAREP
13	MPI_ERR_ERRHANDLER
14	MPI_ERR_FILE_EXISTS
15	MPI_ERR_FILE_IN_USE
16	MPI_ERR_FILE
17	MPI_ERR_INFO_KEY
18	MPI_ERR_INFO_NOKEY
19	MPI_ERR_INFO_VALUE
20	MPI_ERR_INFO
21	MPI_ERR_IO
22	MPI_ERR_KEYVAL
23	MPI_ERR_LOCKTYPE
24	MPI_ERR_NAME
25	MPI_ERR_NO_MEM
26	MPI_ERR_NOT_SAME
27	MPI_ERR_NO_SPACE
28	MPI_ERR_NO_SUCH_FILE
29	MPI_ERR_PORT
30	MPI_ERR_PROC_ABORTED
31	MPI_ERR_QUOTA
32	MPI_ERR_READ_ONLY
33	MPI_ERR_RMA_ATTACH
34	MPI_ERR_RMA_CONFLICT
35	MPI_ERR_RMA_RANGE
36	MPI_ERR_RMA_SHARED
37	MPI_ERR_RMA_SYNC
38	MPI_ERR_RMA_FLAVOR
39	MPI_ERR_SERVICE
40	MPI_ERR_SESSION
41	MPI_ERR_SIZE
42	MPI_ERR_SPAWN
43	MPI_ERR_UNSUPPORTED_DATAREP
44	MPI_ERR_UNSUPPORTED_OPERATION
45	MPI_ERR_VALUE_TOO_LARGE
46	MPI_ERR_WIN
47	<hr/> (Continued on next page) <hr/>
48	

Error classes (continued)

C type: const int (or unnamed enum)

Fortran type: INTEGER

MPI_T_ERR_CANNOT_INIT

MPI_T_ERR_NOT_ACCESSIBLE

MPI_T_ERR_NOT_INITIALIZED

MPI_T_ERR_NOT_SUPPORTED

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 array size and reserved index values (Fortran only)

Fortran type: INTEGER

MPI_STATUS_SIZE

MPI_SOURCE

MPI_TAG

MPI_ERROR

Fortran status array size and reserved index values (C only)

C type: int

MPI_F_STATUS_SIZE

MPI_F_SOURCE

MPI_F_TAG

MPI_F_ERROR

Variable Address Size (Fortran only)

Fortran type: INTEGER

MPI_ADDRESS_KIND

MPI_COUNT_KIND

MPI_INTEGER_KIND

MPI_OFFSET_KIND

Error-handling specifiers

C type: MPI_Errhandler

Fortran type: INTEGER or TYPE(MPI_Errhandler)

MPI_ERRORS_ARE_FATAL

MPI_ERRORS_ABORT

MPI_ERRORS_RETURN

Maximum Sizes for Strings

C type: const int (or unnamed enum)

Fortran type: INTEGER

MPI_MAX_DATAREP_STRING

MPI_MAX_ERROR_STRING

MPI_MAX_INFO_KEY

MPI_MAX_INFO_VAL

MPI_MAX_LIBRARY_VERSION_STRING

MPI_MAX_OBJECT_NAME

MPI_MAX_PORT_NAME

MPI_MAX_PROCESSOR_NAME

MPI_MAX_STRINGTAG_LEN

MPI_MAX_PSET_NAME_LEN

Named Predefined Datatypes	C types	
C type: MPI_Datatype		1
Fortran type: INTEGER		2
or TYPE(MPI_Datatype)		3
MPI_CHAR	char	4
	(treated as printable character)	5
MPI_SHORT	signed short int	6
MPI_INT	signed int	7
MPI_LONG	signed long	8
MPI_LONG_LONG_INT	signed long long	9
MPI_LONG_LONG (as a synonym)	signed long long	10
MPI_SIGNED_CHAR	signed char	11
	(treated as integral value)	12
MPI_UNSIGNED_CHAR	unsigned char	13
	(treated as integral value)	14
MPI_UNSIGNED_SHORT	unsigned short	15
MPI_UNSIGNED	unsigned int	16
MPI_UNSIGNED_LONG	unsigned long	17
MPI_UNSIGNED_LONG_LONG	unsigned long long	18
MPI_FLOAT	float	19
MPI_DOUBLE	double	20
MPI_LONG_DOUBLE	long double	21
MPI_WCHAR	wchar_t	22
	(defined in <stddef.h>)	23
	(treated as printable character)	24
MPI_C_BOOL	_Bool	25
MPI_INT8_T	int8_t	26
MPI_INT16_T	int16_t	27
MPI_INT32_T	int32_t	28
MPI_INT64_T	int64_t	29
MPI_UINT8_T	uint8_t	30
MPI_UINT16_T	uint16_t	31
MPI_UINT32_T	uint32_t	32
MPI_UINT64_T	uint64_t	33
MPI_AINT	MPI_Aint	34
MPI_COUNT	MPI_Count	35
MPI_OFFSET	MPI_Offset	36
MPI_C_COMPLEX	float _Complex	37
MPI_C_FLOAT_COMPLEX	float _Complex	38
MPI_C_DOUBLE_COMPLEX	double _Complex	39
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex	40
MPI_BYTE	(any C type)	41
MPI_PACKED	(any C type)	42

43
44
45
46
47
48

Named Predefined Datatypes	Fortran types
C type: MPI_Datatype Fortran type: INTEGER 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 Fortran type: INTEGER 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 Fortran type: INTEGER 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)	1
C type: MPI_Datatype	2
Fortran type: INTEGER or TYPE(MPI_Datatype)	3
MPI_FLOAT_INT	4
MPI_DOUBLE_INT	5
MPI_LONG_INT	6
MPI_2INT	7
MPI_SHORT_INT	8
MPI_LONG_DOUBLE_INT	9
Datatypes for reduction functions (Fortran)	10
C type: MPI_Datatype	11
Fortran type: INTEGER or TYPE(MPI_Datatype)	12
MPI_2REAL	13
MPI_2DOUBLE_PRECISION	14
MPI_2INTEGER	15
Reserved communicators	16
C type: MPI_Comm	17
Fortran type: INTEGER or TYPE(MPI_Comm)	18
MPI_COMM_WORLD	19
MPI_COMM_SELF	20
Communicator split type constants	21
C type: const int (or unnamed enum)	22
Fortran type: INTEGER	23
MPI_COMM_TYPE_SHARED	24
MPI_COMM_TYPE_HW_UNGUIDED	25
MPI_COMM_TYPE_HW_GUIDED	26
Results of communicator and group comparisons	27
C type: const int (or unnamed enum)	28
Fortran type: INTEGER	29
MPI_IDENT	30
MPI_CONGRUENT	31
MPI_SIMILAR	32
MPI_UNEQUAL	33
Environmental inquiry info key	34
C type: MPI_Info	35
Fortran type: INTEGER or TYPE(MPI_Info)	36
MPI_INFO_ENV	37
Environmental inquiry keys	38
C type: const int (or unnamed enum)	39
Fortran type: INTEGER	40
MPI_TAG_UB	41
MPI_IO	42
MPI_HOST	43
MPI_WTIME_IS_GLOBAL	44

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

Collective Operations

C type:	MPI_Op
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	
C type / Fortran type	

MPI_GROUP_NULL	
MPI_Group / INTEGER or TYPE(MPI_Group)	
MPI_COMM_NULL	
MPI_Comm / INTEGER or TYPE(MPI_Comm)	
MPI_DATATYPE_NULL	
MPI_Datatype / INTEGER or TYPE(MPI_Datatype)	
MPI_REQUEST_NULL	
MPI_Request / INTEGER or TYPE(MPI_Request)	
MPI_OP_NULL	
MPI_Op / INTEGER or TYPE(MPI_Op)	
MPI_ERRHANDLER_NULL	
MPI_Errhandler / INTEGER or TYPE(MPI_Errhandler)	
MPI_FILE_NULL	
MPI_File / INTEGER or TYPE(MPI_File)	
MPI_INFO_NULL	
MPI_Info / INTEGER or TYPE(MPI_Info)	
MPI_SESSION_NULL	
MPI_Session / INTEGER or TYPE(MPI_Session)	
MPI_WIN_NULL	
MPI_Win / INTEGER or TYPE(MPI_Win)	
MPI_MESSAGE_NULL	
MPI_Message / INTEGER or TYPE(MPI_Message)	

Empty group

C type:	MPI_Group
Fortran type:	INTEGER or TYPE(MPI_Group)

MPI_GROUP_EMPTY

Topologies

C type: const int (or unnamed enum)
Fortran type: INTEGER
MPI_GRAPH
MPI_CART
MPI_DIST_GRAPH

Predefined functions

C/Fortran name	
C type	
/ Fortran type with mpi module	/ Fortran type with mpi_f08 module
MPI_COMM_NULL_COPY_FN	
MPI_Comm_copy_attr_function	
/ COMM_COPY_ATTR_FUNCTION	/ PROCEDURE(MPI_Comm_copy_attr_function) ¹⁾
MPI_COMM_DUP_FN	
MPI_Comm_copy_attr_function	
/ COMM_COPY_ATTR_FUNCTION	/ PROCEDURE(MPI_Comm_copy_attr_function) ¹⁾
MPI_COMM_NULL_DELETE_FN	
MPI_Comm_delete_attr_function	
/ COMM_DELETE_ATTR_FUNCTION	/ PROCEDURE(MPI_Comm_delete_attr_function) ¹⁾
MPI_WIN_NULL_COPY_FN	
MPI_Win_copy_attr_function	
/ WIN_COPY_ATTR_FUNCTION	/ PROCEDURE(MPI_Win_copy_attr_function) ¹⁾
MPI_WIN_DUP_FN	
MPI_Win_copy_attr_function	
/ WIN_COPY_ATTR_FUNCTION	/ PROCEDURE(MPI_Win_copy_attr_function) ¹⁾
MPI_WIN_NULL_DELETE_FN	
MPI_Win_delete_attr_function	
/ WIN_DELETE_ATTR_FUNCTION	/ PROCEDURE(MPI_Win_delete_attr_function) ¹⁾
MPI_TYPE_NULL_COPY_FN	
MPI_Type_copy_attr_function	
/ TYPE_COPY_ATTR_FUNCTION	/ PROCEDURE(MPI_Type_copy_attr_function) ¹⁾
MPI_TYPE_DUP_FN	
MPI_Type_copy_attr_function	
/ TYPE_COPY_ATTR_FUNCTION	/ PROCEDURE(MPI_Type_copy_attr_function) ¹⁾
MPI_TYPE_NULL_DELETE_FN	
MPI_Type_delete_attr_function	
/ TYPE_DELETE_ATTR_FUNCTION	/ PROCEDURE(MPI_Type_delete_attr_function) ¹⁾
MPI_CONVERSION_FN_NULL	
MPI_Datarep_conversion_function	
/ DATAREP_CONVERSION_FUNCTION	/ PROCEDURE(MPI_Datarep_conversion_function) ¹⁾
MPI_CONVERSION_FN_NULL_C	
MPI_Datarep_conversion_function_c	
/ (n/a)	/ PROCEDURE(MPI_Datarep_conversion_function_c)

¹⁾ See the advice to implementors (on page 347) and advice to users (on page 348) on the predefined Fortran functions MPI_COMM_NULL_COPY_FN, ... in Section 7.7.2.

1	
2	Deprecated predefined functions
3	<hr/>
4	C/Fortran name
5	C type / Fortran type with mpi module
6	<hr/>
7	MPI_NULL_COPY_FN
8	MPI_Copy_function / COPY_FUNCTION
9	MPI_DUP_FN
10	MPI_Copy_function / COPY_FUNCTION
11	MPI_NULL_DELETE_FN
12	MPI_Delete_function / DELETE_FUNCTION
13	<hr/>
14	Predefined Attribute Keys
15	<hr/>
16	C type: const int (or unnamed enum)
17	Fortran type: INTEGER
18	<hr/>
19	MPI_APPNUM
20	MPI_LASTUSED_CODE
21	MPI_UNIVERSE_SIZE
22	MPI_WIN_BASE
23	MPI_WIN_DISP_UNIT
24	MPI_WIN_SIZE
25	MPI_WIN_CREATE_FLAVOR
26	MPI_WIN_MODEL
27	<hr/>
28	MPI Window Create Flavors
29	<hr/>
30	C type: const int (or unnamed enum)
31	Fortran type: INTEGER
32	<hr/>
33	MPI_WIN_FLAVOR_CREATE
34	MPI_WIN_FLAVOR_ALLOCATE
35	MPI_WIN_FLAVOR_DYNAMIC
36	MPI_WIN_FLAVOR_SHARED
37	<hr/>
38	MPI Window Models
39	<hr/>
40	C type: const int (or unnamed enum)
41	Fortran type: INTEGER
42	<hr/>
43	MPI_WIN_SEPARATE
44	MPI_WIN_UNIFIED
45	<hr/>
46	
47	
48	

Mode Constants	1
C type: const int (or unnamed enum)	2
Fortran type: INTEGER	3
<hr/>	
MPI_MODE_APPEND	4
MPI_MODE_CREATE	5
MPI_MODE_DELETE_ON_CLOSE	6
MPI_MODE_EXCL	7
MPI_MODE_NOCHECK	8
MPI_MODE_NOPRECEDE	9
MPI_MODE_NOPUT	10
MPI_MODE_NOSTORE	11
MPI_MODE_NOSUCCEED	12
MPI_MODE_RDONLY	13
MPI_MODE_RDWR	14
MPI_MODE_SEQUENTIAL	15
MPI_MODE_UNIQUE_OPEN	16
MPI_MODE_WRONLY	17
<hr/>	18
Datatype Decoding Constants	19
C type: const int (or unnamed enum)	20
Fortran type: INTEGER	21
<hr/>	
MPI_COMBINER_CONTIGUOUS	22
MPI_COMBINER_DARRAY	23
MPI_COMBINER_DUP	24
MPI_COMBINER_F90_COMPLEX	25
MPI_COMBINER_F90_INTEGER	26
MPI_COMBINER_F90_REAL	27
MPI_COMBINER_HINDEXED	28
MPI_COMBINER_HVECTOR	29
MPI_COMBINER_INDEXED_BLOCK	30
MPI_COMBINER_HINDEXED_BLOCK	31
MPI_COMBINER_INDEXED	32
MPI_COMBINER_NAMED	33
MPI_COMBINER_RESIZED	34
MPI_COMBINER_STRUCT	35
MPI_COMBINER_SUBARRAY	36
MPI_COMBINER_VECTOR	37
<hr/>	38
Threads Constants	39
C type: const int (or unnamed enum)	40
Fortran type: INTEGER	41
<hr/>	
MPI_THREAD_FUNNELED	42
MPI_THREAD_MULTIPLE	43
MPI_THREAD_SERIALIZED	44
MPI_THREAD_SINGLE	45
<hr/>	46
	47
	48

File Operation Constants, Part 1

C type: const MPI_Offset (or unnamed enum)
 Fortran type: INTEGER(KIND=MPI_OFFSET_KIND)

MPI_DISPLACEMENT_CURRENT

File Operation Constants, Part 2

C type: const int (or unnamed enum)
 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)
 Fortran type: INTEGER

MPI_TYPECLASS_COMPLEX
 MPI_TYPECLASS_INTEGER
 MPI_TYPECLASS_REAL

Constants Specifying Empty or Ignored Input

C/Fortran name	C type / Fortran type ¹
MPI_ARGVS_NULL	char*** / 2-dim. array of CHARACTER*(*)
MPI_ARGV_NULL	char** / array of CHARACTER*(*)
MPI_ERRCODES_IGNORE	int* / INTEGER array
MPI_STATUSES_IGNORE	MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE, *) or TYPE(MPI_Status), DIMENSION(*)
MPI_STATUS_IGNORE	MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status)
MPI_UNWEIGHTED	int* / INTEGER array
MPI_WEIGHTS_EMPTY	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 constant (type: MPI_Fint*)	is equivalent to the Fortran constant	1
MPI_F_STATUSES_IGNORE	MPI_STATUSES_IGNORE in mpi / mpif.h	2
MPI_F_STATUS_IGNORE	MPI_STATUS_IGNORE in mpi / mpif.h	3
C constant (type: MPI_F08_status*)	is equivalent to the Fortran constant	4
MPI_F08_STATUSES_IGNORE	MPI_STATUSES_IGNORE in mpi_f08	5
MPI_F08_STATUS_IGNORE	MPI_STATUS_IGNORE in mpi_f08	6

C preprocessor Constants and Fortran Parameters

C type: C-preprocessor macro that expands to an int value

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

1 **Constants to identify associations of variables**
 2 **in the MPI tool information interface**

3 C type: const int (or unnamed enum)

4 MPI_T_BIND_NO_OBJECT
 5 MPI_T_BIND_MPI_COMM
 6 MPI_T_BIND_MPI_DATATYPE
 7 MPI_T_BIND_MPI_ERRHANDLER
 8 MPI_T_BIND_MPI_FILE
 9 MPI_T_BIND_MPI_GROUP
 10 MPI_T_BIND_MPI_OP
 11 MPI_T_BIND_MPI_REQUEST
 12 MPI_T_BIND_MPI_WIN
 13 MPI_T_BIND_MPI_MESSAGE
 14 MPI_T_BIND_MPI_INFO
 15 MPI_T_BIND_MPI_SESSION

16
 17 **Constants describing the scope of a control variable**
 18 **in the MPI tool information interface**

19 C type: const int (or unnamed enum)

20 MPI_T_SCOPE_CONSTANT
 21 MPI_T_SCOPE_READONLY
 22 MPI_T_SCOPE_LOCAL
 23 MPI_T_SCOPE_GROUP
 24 MPI_T_SCOPE_GROUP_EQ
 25 MPI_T_SCOPE_ALL
 26 MPI_T_SCOPE_ALL_EQ

27
 28 **Additional constants used**
 29 **by the MPI tool information interface**

30 C type: MPI_T_pvar_handle

31 MPI_T_PVAR_ALL_HANDLES

32
 33 **Performance variables classes used by the**
 34 **MPI tool information interface**

35 C type: const int (or unnamed enum)

36 MPI_T_PVAR_CLASS_STATE
 37 MPI_T_PVAR_CLASS_LEVEL
 38 MPI_T_PVAR_CLASS_SIZE
 39 MPI_T_PVAR_CLASS_PERCENTAGE
 40 MPI_T_PVAR_CLASS_HIGHWATERMARK
 41 MPI_T_PVAR_CLASS_LOWWATERMARK
 42 MPI_T_PVAR_CLASS_COUNTER
 43 MPI_T_PVAR_CLASS_AGGREGATE
 44 MPI_T_PVAR_CLASS_TIMER
 45 MPI_T_PVAR_CLASS_GENERIC

**Source event ordering guarantees in the
MPI tool information interface**

C type: MPI_T_source_order

MPI_T_SOURCE_ORDERED

MPI_T_SOURCE_UNORDERED

**Callback safety requirement levels used in the
MPI tool information interface**

C type: MPI_T_cb_safety

MPI_T_CB_REQUIRE_NONE

MPI_T_CB_REQUIRE_MPI_RESTRICTED

MPI_T_CB_REQUIRE_THREAD_SAFE

MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE

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_Session

MPI_Win

/ Types for the MPI_T interface */*

MPI_T_enum

MPI_T_cvar_handle

MPI_T_pvar_handle

MPI_T_pvar_session

MPI_T_event_instance

MPI_T_event_registration

MPI_T_source_order

MPI_T_cb_safety

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_Session)
```

```
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 void MPI_User_function_c(void *invec, void *inoutvec, MPI_Count *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,      1
                                         int type_keyval, void *attribute_val, void *extra_state);      2
                                                                 3
typedef void MPI_Comm_errhandler_function(MPI_Comm *comm, int *error_code,      4
                                         ...);                                                                 5
                                                                 6
typedef void MPI_Win_errhandler_function(MPI_Win *win, int *error_code, ...);      7
                                                                 8
typedef void MPI_File_errhandler_function(MPI_File *file, int *error_code,      9
                                         ...);                                                                10
                                                                 11
typedef void MPI_Session_errhandler_function(MPI_Session *session,      12
                                             int *error_code, ...);                                          13
                                                                 14
typedef int MPI_Grequest_query_function(void *extra_state, MPI_Status *status);  15
                                                                 16
typedef int MPI_Grequest_free_function(void *extra_state);              17
                                                                 18
typedef int MPI_Grequest_cancel_function(void *extra_state, int complete);    19
                                                                 20
typedef int MPI_Datarep_extent_function(MPI_Datatype datatype,      21
                                         MPI_Aint *extent, void *extra_state);  22
                                                                 23
typedef int MPI_Datarep_conversion_function(void *userbuf,          24
                                             MPI_Datatype datatype, int count, void *filebuf,
                                             MPI_Offset position, void *extra_state);  25
                                                                 26
typedef int MPI_Datarep_conversion_function_c(void *userbuf,        27
                                             MPI_Datatype datatype, MPI_Count count, void *filebuf,
                                             MPI_Offset position, void *extra_state);  28
                                                                 29
typedef void MPI_T_event_cb_function(MPI_T_event_instance event_instance,    30
                                     MPI_T_event_registration event_registration,
                                     MPI_T_cb_safety cb_safety, void *user_data);  31
                                                                 32
typedef void MPI_T_event_free_cb_function(                               33
    MPI_T_event_registration event_registration,
    MPI_T_cb_safety cb_safety, void *user_data);                          34
                                                                 35
typedef void MPI_T_event_dropped_cb_function(MPI_Count count,        36
                                             MPI_T_event_registration event_registration, int source_index,
                                             MPI_T_cb_safety cb_safety, void *user_data);  37
                                                                 38

```

Fortran 2008 Bindings with the `mpi_f08` Module 39

The callback prototypes when using the Fortran `mpi_f08` module are shown below: 40

The user-function argument to `MPI_Op_create` and `MPI_Op_create_c` should be declared according to: 41

ABSTRACT INTERFACE 42

 SUBROUTINE `MPI_User_function`(`invec`, `inoutvec`, `len`, `datatype`) 43

 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR 44

 TYPE(C_PTR), VALUE :: `invec`, `inoutvec` 45

 INTEGER :: `len` 46

47

```
1     TYPE(MPI_Datatype) :: datatype
```

```
2
```

```
3 ABSTRACT INTERFACE
```

```
4     SUBROUTINE MPI_User_function_c(invec, inoutvec, len, datatype) !(_c)
```

```
5     USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
```

```
6     TYPE(C_PTR), VALUE :: invec, inoutvec
```

```
7     INTEGER(KIND=MPI_COUNT_KIND) :: len
```

```
8     TYPE(MPI_Datatype) :: datatype
```

```
9
```

The copy and delete function arguments to MPI_Comm_create_keyval should be declared according to:

```
10 ABSTRACT INTERFACE
```

```
11     SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,  
12         attribute_val_in, attribute_val_out, flag, ierror)
```

```
13     TYPE(MPI_Comm) :: oldcomm
```

```
14     INTEGER :: comm_keyval, ierror
```

```
15     INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,  
16         attribute_val_out
```

```
17     LOGICAL :: flag
```

```
18
```

```
19 ABSTRACT INTERFACE
```

```
20     SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval, attribute_val,  
21         extra_state, ierror)
```

```
22     TYPE(MPI_Comm) :: comm
```

```
23     INTEGER :: comm_keyval, ierror
```

```
24     INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
```

```
25
```

The copy and delete function arguments to MPI_Win_create_keyval should be declared according to:

```
26 ABSTRACT INTERFACE
```

```
27     SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state,  
28         attribute_val_in, attribute_val_out, flag, ierror)
```

```
29     TYPE(MPI_Win) :: oldwin
```

```
30     INTEGER :: win_keyval, ierror
```

```
31     INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,  
32         attribute_val_out
```

```
33     LOGICAL :: flag
```

```
34
```

```
35 ABSTRACT INTERFACE
```

```
36     SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,  
37         extra_state, ierror)
```

```
38     TYPE(MPI_Win) :: win
```

```
39     INTEGER :: win_keyval, ierror
```

```
40     INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
```

```
41
```

The copy and delete function arguments to MPI_Type_create_keyval should be declared according to:

```
42 ABSTRACT INTERFACE
```

```
43     SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state,  
44         attribute_val_in, attribute_val_out, flag, ierror)
```

```
45
```

```

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 handler-function argument to MPI_Session_create_errhandler should be declared
like this:
ABSTRACT INTERFACE
SUBROUTINE MPI_Session_errhandler_function(session, error_code)
TYPE(MPI_Session) :: session
INTEGER :: error_code
The query, free, and cancel function arguments to MPI_Grequest_start should be de-
clared according to:
ABSTRACT INTERFACE
SUBROUTINE MPI_Grequest_query_function(extra_state, status, ierror)
INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
TYPE(MPI_Status) :: status
INTEGER :: ierror
ABSTRACT INTERFACE
SUBROUTINE MPI_Grequest_free_function(extra_state, ierror)

```

```

1      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
2      INTEGER :: ierror
3
4  ABSTRACT INTERFACE
5      SUBROUTINE MPI_Grequest_cancel_function(extra_state, complete, ierror)
6          INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
7          LOGICAL :: complete
8          INTEGER :: ierror
9
10     The extent and conversion function arguments to MPI_Register_datarep and
11     MPI_Register_datarep_c should be declared according to:
12     ABSTRACT INTERFACE
13         SUBROUTINE MPI_Datarep_extent_function(datatype, extent, extra_state, ierror)
14             TYPE(MPI_Datatype) :: datatype
15             INTEGER(KIND=MPI_ADDRESS_KIND) :: extent, extra_state
16             INTEGER :: ierror
17     ABSTRACT INTERFACE
18         SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count, filebuf,
19             position, extra_state, ierror)
20             USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
21             TYPE(C_PTR), VALUE :: userbuf, filebuf
22             TYPE(MPI_Datatype) :: datatype
23             INTEGER :: count, ierror
24             INTEGER(KIND=MPI_OFFSET_KIND) :: position
25             INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
26     ABSTRACT INTERFACE
27         SUBROUTINE MPI_Datarep_conversion_function_c(userbuf, datatype, count,
28             filebuf, position, extra_state, ierror) !(_c)
29             USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
30             TYPE(C_PTR), VALUE :: userbuf, filebuf
31             TYPE(MPI_Datatype) :: datatype
32             INTEGER(KIND=MPI_COUNT_KIND) :: count
33             INTEGER(KIND=MPI_OFFSET_KIND) :: position
34             INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
35             INTEGER :: ierror
36
37

```

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:

```

42     SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, DATATYPE)
43         <type> INVEC(LEN), INOUTVEC(LEN)
44         INTEGER LEN, DATATYPE
45

```

The copy and delete function arguments to `MPI_COMM_CREATE_KEYVAL` should be declared like these:

48

```

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

```

1 The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de-
2 clared like this:

```
3 SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)
4     INTEGER FILE, ERROR_CODE
```

5 The handler-function argument to MPI_SESSION_CREATE_ERRHANDLER should be
6 declared like this:

```
7 SUBROUTINE SESSION_ERRHANDLER_FUNCTION(SESSION, ERROR_CODE)
8     INTEGER SESSION, ERROR_CODE
```

9 The query, free, and cancel function arguments to MPI_GREQUEST_START should be
10 declared like these:

```
11 SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)
12     INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
13     INTEGER STATUS(MPI_STATUS_SIZE), IERROR
```

```
14 SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)
15     INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
16     INTEGER IERROR
```

```
17 SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)
18     INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
19     LOGICAL COMPLETE
20     INTEGER IERROR
```

21 The extent and conversion function arguments to MPI_REGISTER_DATAREP should
22 be declared like these:

```
23 SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)
24     INTEGER DATATYPE, IERROR
25     INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE
```

```
26 SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF,
27     POSITION, EXTRA_STATE, IERROR)
28     <TYPE> USERBUF(*), FILEBUF(*)
29     INTEGER DATATYPE, COUNT, IERROR
30     INTEGER(KIND=MPI_OFFSET_KIND) POSITION
31     INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

32 A.1.4 Deprecated Prototype Definitions

33 The following are defined C typedefs for deprecated user-defined functions, also included in
34 the file `mpi.h`.

```
35 /* prototypes for user-defined functions */
```

```
36 typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval, void *extra_state,
37     void *attribute_val_in, void *attribute_val_out, int *flag);
```

```
38 typedef int MPI_Delete_function(MPI_Comm comm, int keyval, void *attribute_val,
39     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 String Values

Default Communicator Names

The following default communicator names are defined by MPI.

```

"MPI_COMM_WORLD"
"MPI_COMM_SELF"
"MPI_COMM_PARENT"
"MPI_COMM_NULL"

```

Default Datatype Names

Named predefined datatypes have the default names of the datatype name. In addition, the following default datatype name is defined by MPI.

```

"MPI_DATATYPE_NULL"

```

Default Window Names

The following default window name is defined by MPI.

```

"MPI_WIN_NULL"

```

Reserved Data Representations

The following data representations are supported by MPI.

```

"native"
"internal"
"external32"

```

Process Set Names

Process set name	Comment
"mpi://"	reserved namespace
"mpi://SELF"	mandatory process set name
"mpi://WORLD"	mandatory process set name

1 Info Keys

2 The following info keys are reserved. They are strings.
3

4 "access_style"
5 "accumulate_ops"
6 "accumulate_ordering"
7 "alloc_shared_noncontig"
8 "appnum"
9 "arch"
10 "argv"
11 "cb_block_size"
12 "cb_buffer_size"
13 "cb_nodes"
14 "chunked_item"
15 "chunked_size"
16 "chunked"
17 "collective_buffering"
18 "command"
19 "file"
20 "file_perm"
21 "filename"
22 "host"
23 "io_node_list"
24 "ip_address"
25 "ip_port"
26 "maxprocs"
27 "mpi_assert_allow_overtaking"
28 "mpi_assert_exact_length"
29 "mpi_assert_no_any_source"
30 "mpi_assert_no_any_tag"
31 "mpi_hw_resource_type"
32 "mpi_initial_errhandler"
33 "mpi_minimum_memory_alignment"
34 "mpi_size"
35 "nb_proc"
36 "no_locks"
37 "num_io_nodes"
38 "path"
39 "same_disp_unit"
40 "same_size"
41 "soft"
42 "striping_factor"
43 "striping_unit"
44 "thread_level"
45 "wdir"

46
47
48

Info Values

The following info values are reserved. They are strings.

"false"

"mpi_errors_abort"

"mpi_errors_are_fatal"

"mpi_errors_return"

"mpi_shared_memory"

"MPI_THREAD_FUNNELED"

"MPI_THREAD_MULTIPLE"

"MPI_THREAD_SERIALIZED"

"MPI_THREAD_SINGLE"

"none"

"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 Summary of the Semantics of all Operation-Related MPI Procedures

A summary of the semantics of all operation-related MPI procedures can be found in [51].

A.3 C Bindings

A.3.1 Point-to-Point Communication C Bindings

```
1 int MPI_Bsend(const void *buf, int count, MPI_Datatype datatype, int dest,
2               int tag, MPI_Comm comm)
3
4 int MPI_Bsend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,
5                int dest, int tag, MPI_Comm comm)
6
7 int MPI_Bsend_init(const void *buf, int count, MPI_Datatype datatype, int dest,
8                   int tag, MPI_Comm comm, MPI_Request *request)
9
10 int MPI_Bsend_init_c(const void *buf, MPI_Count count, MPI_Datatype datatype,
11                    int dest, int tag, MPI_Comm comm, MPI_Request *request)
12
13 int MPI_Buffer_attach(void *buffer, int size)
14
15 int MPI_Buffer_attach_c(void *buffer, MPI_Count size)
16
17 int MPI_Buffer_detach(void *buffer_addr, int *size)
18
19 int MPI_Buffer_detach_c(void *buffer_addr, MPI_Count *size)
20
21 int MPI_Cancel(MPI_Request *request)
22
23 int MPI_Get_count(const MPI_Status *status, MPI_Datatype datatype, int *count)
24
25 int MPI_Get_count_c(const MPI_Status *status, MPI_Datatype datatype,
26                   MPI_Count *count)
27
28 int MPI_Ibsend(const void *buf, int count, MPI_Datatype datatype, int dest,
29               int tag, MPI_Comm comm, MPI_Request *request)
30
31 int MPI_Ibsend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,
32                 int dest, int tag, MPI_Comm comm, MPI_Request *request)
33
34 int MPI_Improbe(int source, int tag, MPI_Comm comm, int *flag,
35                MPI_Message *message, MPI_Status *status)
36
37 int MPI_Imrecv(void *buf, int count, MPI_Datatype datatype,
38               MPI_Message *message, MPI_Request *request)
39
40 int MPI_Imrecv_c(void *buf, MPI_Count count, MPI_Datatype datatype,
41                 MPI_Message *message, MPI_Request *request)
42
43 int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
44               MPI_Status *status)
45
46 int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
47               MPI_Comm comm, MPI_Request *request)
48
49 int MPI_Irecv_c(void *buf, MPI_Count count, MPI_Datatype datatype, int source,
50                int tag, MPI_Comm comm, MPI_Request *request)
51
52 int MPI_Irsend(const void *buf, int count, MPI_Datatype datatype, int dest,
53               int tag, MPI_Comm comm, MPI_Request *request)
```

```
int MPI_Irsend_c(const void *buf, MPI_Count count, MPI_Datatype datatype, 1
                int dest, int tag, MPI_Comm comm, MPI_Request *request) 2
int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, 3
              int tag, MPI_Comm comm, MPI_Request *request) 4
int MPI_Isend_c(const void *buf, MPI_Count count, MPI_Datatype datatype, 5
               int dest, int tag, MPI_Comm comm, MPI_Request *request) 6
int MPI_Isendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, 7
                 int dest, int sendtag, void *recvbuf, int recvcount, 8
                 MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, 9
                 MPI_Request *request) 10
int MPI_Isendrecv_c(const void *sendbuf, MPI_Count sendcount, 11
                   MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, 12
                   MPI_Count recvcount, MPI_Datatype recvtype, int source, 13
                   int recvtag, MPI_Comm comm, MPI_Request *request) 14
int MPI_Isendrecv_replace(void *buf, int count, MPI_Datatype datatype, 15
                          int dest, int sendtag, int source, int recvtag, MPI_Comm comm, 16
                          MPI_Request *request) 17
int MPI_Isendrecv_replace_c(void *buf, MPI_Count count, MPI_Datatype datatype, 18
                            int dest, int sendtag, int source, int recvtag, MPI_Comm comm, 19
                            MPI_Request *request) 20
int MPI_Issend(const void *buf, int count, MPI_Datatype datatype, int dest, 21
              int tag, MPI_Comm comm, MPI_Request *request) 22
int MPI_Issend_c(const void *buf, MPI_Count count, MPI_Datatype datatype, 23
                int dest, int tag, MPI_Comm comm, MPI_Request *request) 24
int MPI_Mprobe(int source, int tag, MPI_Comm comm, MPI_Message *message, 25
              MPI_Status *status) 26
int MPI_Mrecv(void *buf, int count, MPI_Datatype datatype, 27
             MPI_Message *message, MPI_Status *status) 28
int MPI_Mrecv_c(void *buf, MPI_Count count, MPI_Datatype datatype, 29
               MPI_Message *message, MPI_Status *status) 30
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status) 31
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, 32
            MPI_Comm comm, MPI_Status *status) 33
int MPI_Recv_c(void *buf, MPI_Count count, MPI_Datatype datatype, int source, 34
              int tag, MPI_Comm comm, MPI_Status *status) 35
int MPI_Recv_init(void *buf, int count, MPI_Datatype datatype, int source, 36
                 int tag, MPI_Comm comm, MPI_Request *request) 37
int MPI_Recv_init_c(void *buf, MPI_Count count, MPI_Datatype datatype, 38
                   int source, int tag, MPI_Comm comm, MPI_Request *request) 39
int MPI_Recv_init_c(void *buf, MPI_Count count, MPI_Datatype datatype, 40
                   int source, int tag, MPI_Comm comm, MPI_Request *request) 41
int MPI_Recv_init_c(void *buf, MPI_Count count, MPI_Datatype datatype, 42
                   int source, int tag, MPI_Comm comm, MPI_Request *request) 43
int MPI_Recv_init_c(void *buf, MPI_Count count, MPI_Datatype datatype, 44
                   int source, int tag, MPI_Comm comm, MPI_Request *request) 45
int MPI_Recv_init_c(void *buf, MPI_Count count, MPI_Datatype datatype, 46
                   int source, int tag, MPI_Comm comm, MPI_Request *request) 47
int MPI_Recv_init_c(void *buf, MPI_Count count, MPI_Datatype datatype, 48
                   int source, int tag, MPI_Comm comm, MPI_Request *request) 48
```

```
1 int MPI_Request_free(MPI_Request *request)
2
3 int MPI_Request_get_status(MPI_Request request, int *flag, MPI_Status *status)
4
5 int MPI_Rsend(const void *buf, int count, MPI_Datatype datatype, int dest,
6             int tag, MPI_Comm comm)
7
8 int MPI_Rsend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,
9             int dest, int tag, MPI_Comm comm)
10
11 int MPI_Rsend_init(const void *buf, int count, MPI_Datatype datatype, int dest,
12                 int tag, MPI_Comm comm, MPI_Request *request)
13
14 int MPI_Rsend_init_c(const void *buf, MPI_Count count, MPI_Datatype datatype,
15                    int dest, int tag, MPI_Comm comm, MPI_Request *request)
16
17 int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest,
18             int tag, MPI_Comm comm)
19
20 int MPI_Send_c(const void *buf, MPI_Count count, MPI_Datatype datatype,
21             int dest, int tag, MPI_Comm comm)
22
23 int MPI_Send_init(const void *buf, int count, MPI_Datatype datatype, int dest,
24                 int tag, MPI_Comm comm, MPI_Request *request)
25
26 int MPI_Send_init_c(const void *buf, MPI_Count count, MPI_Datatype datatype,
27                    int dest, int tag, MPI_Comm comm, MPI_Request *request)
28
29 int MPI_Sendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
30                 int dest, int sendtag, void *recvbuf, int recvcount,
31                 MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm,
32                 MPI_Status *status)
33
34 int MPI_Sendrecv_c(const void *sendbuf, MPI_Count sendcount,
35                   MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf,
36                   MPI_Count recvcount, MPI_Datatype recvtype, int source,
37                   int recvtag, MPI_Comm comm, MPI_Status *status)
38
39 int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype, int dest,
40                          int sendtag, int source, int recvtag, MPI_Comm comm,
41                          MPI_Status *status)
42
43 int MPI_Sendrecv_replace_c(void *buf, MPI_Count count, MPI_Datatype datatype,
44                           int dest, int sendtag, int source, int recvtag, MPI_Comm comm,
45                           MPI_Status *status)
46
47 int MPI_Ssend(const void *buf, int count, MPI_Datatype datatype, int dest,
48             int tag, MPI_Comm comm)
49
50 int MPI_Ssend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,
51             int dest, int tag, MPI_Comm comm)
52
53 int MPI_Ssend_init(const void *buf, int count, MPI_Datatype datatype, int dest,
54                 int tag, MPI_Comm comm, MPI_Request *request)
```

```

int MPI_Ssend_init_c(const void *buf, MPI_Count count, MPI_Datatype datatype, 1
                    int dest, int tag, MPI_Comm comm, MPI_Request *request) 2
int MPI_Start(MPI_Request *request) 3
int MPI_Startall(int count, MPI_Request array_of_requests[]) 4
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status) 5
int MPI_Test_cancelled(const MPI_Status *status, int *flag) 6
int MPI_Testall(int count, MPI_Request array_of_requests[], int *flag, 7
                MPI_Status array_of_statuses[]) 8
int MPI_Testany(int count, MPI_Request array_of_requests[], int *index, 9
                int *flag, MPI_Status *status) 10
int MPI_Testsome(int incount, MPI_Request array_of_requests[], int *outcount, 11
                 int array_of_indices[], MPI_Status array_of_statuses[]) 12
int MPI_Wait(MPI_Request *request, MPI_Status *status) 13
int MPI_Waitall(int count, MPI_Request array_of_requests[], 14
                MPI_Status array_of_statuses[]) 15
int MPI_Waitany(int count, MPI_Request array_of_requests[], int *index, 16
                MPI_Status *status) 17
int MPI_Waitsome(int incount, MPI_Request array_of_requests[], int *outcount, 18
                 int array_of_indices[], MPI_Status array_of_statuses[]) 19
int MPI_Waitany(int count, MPI_Request array_of_requests[], int *index, 20
                MPI_Status *status) 21
int MPI_Waitsome(int incount, MPI_Request array_of_requests[], int *outcount, 22
                 int array_of_indices[], MPI_Status array_of_statuses[]) 23
int MPI_Waitany(int count, MPI_Request array_of_requests[], int *index, 24
                MPI_Status *status) 25
int MPI_Waitsome(int incount, MPI_Request array_of_requests[], int *outcount, 26
                 int array_of_indices[], MPI_Status array_of_statuses[]) 27
A.3.2 Partitioned Communication C Bindings 28
int MPI_Parrived(MPI_Request request, int partition, int *flag) 29
int MPI_Pready(int partition, MPI_Request request) 30
int MPI_Pready_list(int length, const int array_of_partitions[], 31
                    MPI_Request request) 32
int MPI_Pready_range(int partition_low, int partition_high, 33
                     MPI_Request request) 34
int MPI_Precv_init(void *buf, int partitions, MPI_Count count, 35
                  MPI_Datatype datatype, int source, int tag, MPI_Comm comm, 36
                  MPI_Info info, MPI_Request *request) 37
int MPI_Psend_init(const void *buf, int partitions, MPI_Count count, 38
                  MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, 39
                  MPI_Info info, MPI_Request *request) 40
int MPI_Psend_init(const void *buf, int partitions, MPI_Count count, 41
                  MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, 42
                  MPI_Info info, MPI_Request *request) 43
A.3.3 Datatypes C Bindings 44
MPI_Aint MPI_Aint_add(MPI_Aint base, MPI_Aint disp) 45

```

```
1 MPI_Aint MPI_Aint_diff(MPI_Aint addr1, MPI_Aint addr2)
2
3 int MPI_Get_address(const void *location, MPI_Aint *address)
4
5 int MPI_Get_elements(const MPI_Status *status, MPI_Datatype datatype,
6     int *count)
7
8 int MPI_Get_elements_c(const MPI_Status *status, MPI_Datatype datatype,
9     MPI_Count *count)
10
11 int MPI_Get_elements_x(const MPI_Status *status, MPI_Datatype datatype,
12     MPI_Count *count)
13
14 int MPI_Pack(const void *inbuf, int incount, MPI_Datatype datatype,
15     void *outbuf, int outsize, int *position, MPI_Comm comm)
16
17 int MPI_Pack_c(const void *inbuf, MPI_Count incount, MPI_Datatype datatype,
18     void *outbuf, MPI_Count outsize, MPI_Count *position,
19     MPI_Comm comm)
20
21 int MPI_Pack_external(const char datarep[], const void *inbuf, int incount,
22     MPI_Datatype datatype, void *outbuf, MPI_Aint outsize,
23     MPI_Aint *position)
24
25 int MPI_Pack_external_c(const char datarep[], const void *inbuf,
26     MPI_Count incount, MPI_Datatype datatype, void *outbuf,
27     MPI_Count outsize, MPI_Count *position)
28
29 int MPI_Pack_external_size(const char datarep[], int incount,
30     MPI_Datatype datatype, MPI_Aint *size)
31
32 int MPI_Pack_external_size_c(const char datarep[], MPI_Count incount,
33     MPI_Datatype datatype, MPI_Count *size)
34
35 int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm, int *size)
36
37 int MPI_Pack_size_c(MPI_Count incount, MPI_Datatype datatype, MPI_Comm comm,
38     MPI_Count *size)
39
40 int MPI_Type_commit(MPI_Datatype *datatype)
41
42 int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
43
44 int MPI_Type_contiguous_c(MPI_Count count, MPI_Datatype oldtype,
45     MPI_Datatype *newtype)
46
47 int MPI_Type_create_darray(int size, int rank, int ndims,
48     const int array_of_gsizes[], const int array_of_distrib[],
49     const int array_of_dargs[], const int array_of_psize[],
50     int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
51
52 int MPI_Type_create_darray_c(int size, int rank, int ndims,
53     const MPI_Count array_of_gsizes[], const int array_of_distrib[],
54     const int array_of_dargs[], const int array_of_psize[],
55     int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```

int MPI_Type_create_hindexed(int count, const int array_of_blocklengths[],      1
                             const MPI_Aint array_of_displacements[], MPI_Datatype oldtype,  2
                             MPI_Datatype *newtype)                                  3
int MPI_Type_create_hindexed_block(int count, int blocklength,                  4
                                   const MPI_Aint array_of_displacements[], MPI_Datatype oldtype,  5
                                   MPI_Datatype *newtype)                          6
int MPI_Type_create_hindexed_block_c(MPI_Count count, MPI_Count blocklength,    8
                                     const MPI_Count array_of_displacements[], MPI_Datatype oldtype,  9
                                     MPI_Datatype *newtype)                       10
int MPI_Type_create_hindexed_c(MPI_Count count,                               11
                               const MPI_Count array_of_blocklengths[],          12
                               const MPI_Count array_of_displacements[], MPI_Datatype oldtype,  13
                               MPI_Datatype *newtype)                            14
int MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride,        16
                             MPI_Datatype oldtype, MPI_Datatype *newtype)        17
int MPI_Type_create_hvector_c(MPI_Count count, MPI_Count blocklength,           18
                              MPI_Count stride, MPI_Datatype oldtype, MPI_Datatype *newtype)  19
int MPI_Type_create_indexed_block(int count, int blocklength,                  21
                                  const int array_of_displacements[], MPI_Datatype oldtype,  22
                                  MPI_Datatype *newtype)                          23
int MPI_Type_create_indexed_block_c(MPI_Count count, MPI_Count blocklength,    24
                                    const MPI_Count array_of_displacements[], MPI_Datatype oldtype,  25
                                    MPI_Datatype *newtype)                       26
int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint extent,  28
                             MPI_Datatype *newtype)                              29
int MPI_Type_create_resized_c(MPI_Datatype oldtype, MPI_Count lb,              31
                              MPI_Count extent, MPI_Datatype *newtype)          32
int MPI_Type_create_struct(int count, const int array_of_blocklengths[],        33
                            const MPI_Aint array_of_displacements[],             34
                            const MPI_Datatype array_of_types[], MPI_Datatype *newtype)  35
int MPI_Type_create_struct_c(MPI_Count count,                                  37
                              const MPI_Count array_of_blocklengths[],           38
                              const MPI_Count array_of_displacements[],          39
                              const MPI_Datatype array_of_types[], MPI_Datatype *newtype)  40
int MPI_Type_create_subarray(int ndims, const int array_of_sizes[],             41
                              const int array_of_subsizes[], const int array_of_starts[],  42
                              int order, MPI_Datatype oldtype, MPI_Datatype *newtype)    43
int MPI_Type_create_subarray_c(int ndims, const MPI_Count array_of_sizes[],     45
                               const MPI_Count array_of_subsizes[],             46
                               const MPI_Count array_of_starts[], int order,      47
                               MPI_Datatype oldtype, MPI_Datatype *newtype)       48

```

```
1 int MPI_Type_dup(MPI_Datatype oldtype, MPI_Datatype *newtype)
2
3 int MPI_Type_free(MPI_Datatype *datatype)
4
5 int MPI_Type_get_contents(MPI_Datatype datatype, int max_integers,
6     int max_addresses, int max_datatypes, int array_of_integers[],
7     MPI_Aint array_of_addresses[], MPI_Datatype array_of_datatypes[])
8
9 int MPI_Type_get_contents_c(MPI_Datatype datatype, MPI_Count max_integers,
10     MPI_Count max_addresses, MPI_Count max_large_counts,
11     MPI_Count max_datatypes, int array_of_integers[],
12     MPI_Aint array_of_addresses[], MPI_Count array_of_large_counts[],
13     MPI_Datatype array_of_datatypes[])
14
15 int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers,
16     int *num_addresses, int *num_datatypes, int *combiner)
17
18 int MPI_Type_get_envelope_c(MPI_Datatype datatype, MPI_Count *num_integers,
19     MPI_Count *num_addresses, MPI_Count *num_large_counts,
20     MPI_Count *num_datatypes, int *combiner)
21
22 int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb, MPI_Aint *extent)
23
24 int MPI_Type_get_extent_c(MPI_Datatype datatype, MPI_Count *lb,
25     MPI_Count *extent)
26
27 int MPI_Type_get_extent_x(MPI_Datatype datatype, MPI_Count *lb,
28     MPI_Count *extent)
29
30 int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb,
31     MPI_Aint *true_extent)
32
33 int MPI_Type_get_true_extent_c(MPI_Datatype datatype, MPI_Count *true_lb,
34     MPI_Count *true_extent)
35
36 int MPI_Type_get_true_extent_x(MPI_Datatype datatype, MPI_Count *true_lb,
37     MPI_Count *true_extent)
38
39 int MPI_Type_indexed(int count, const int array_of_blocklengths[],
40     const int array_of_displacements[], MPI_Datatype oldtype,
41     MPI_Datatype *newtype)
42
43 int MPI_Type_indexed_c(MPI_Count count,
44     const MPI_Count array_of_blocklengths[],
45     const MPI_Count array_of_displacements[], MPI_Datatype oldtype,
46     MPI_Datatype *newtype)
47
48 int MPI_Type_size(MPI_Datatype datatype, int *size)
49
50 int MPI_Type_size_c(MPI_Datatype datatype, MPI_Count *size)
51
52 int MPI_Type_size_x(MPI_Datatype datatype, MPI_Count *size)
53
54 int MPI_Type_vector(int count, int blocklength, int stride,
55     MPI_Datatype oldtype, MPI_Datatype *newtype)
```



```

int MPI_Type_vector_c(MPI_Count count, MPI_Count blocklength, MPI_Count 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_c(const void *inbuf, MPI_Count insize, MPI_Count *position,
    void *outbuf, MPI_Count 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 outcount,
    MPI_Datatype datatype)
int MPI_Unpack_external_c(const char datarep[], const void *inbuf,
    MPI_Count insize, MPI_Count *position, void *outbuf,
    MPI_Count outcount, MPI_Datatype datatype)

A.3.4 Collective Communication C Bindings
int MPI_Allgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtpe,
    MPI_Comm comm)
int MPI_Allgather_c(const void *sendbuf, MPI_Count sendcount,
    MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
    MPI_Datatype recvtpe, MPI_Comm comm)
int MPI_Allgather_init(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf, int recvcount,
    MPI_Datatype recvtpe, MPI_Comm comm, MPI_Info info,
    MPI_Request *request)
int MPI_Allgather_init_c(const void *sendbuf, MPI_Count sendcount,
    MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
    MPI_Datatype recvtpe, MPI_Comm comm, MPI_Info info,
    MPI_Request *request)
int MPI_Allgatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, const int recvcnts[], const int displs[],
    MPI_Datatype recvtpe, MPI_Comm comm)
int MPI_Allgatherv_c(const void *sendbuf, MPI_Count sendcount,
    MPI_Datatype sendtype, void *recvbuf,
    const MPI_Count recvcnts[], const MPI_Aint displs[],
    MPI_Datatype recvtpe, MPI_Comm comm)
int MPI_Allgatherv_init(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf, const int recvcnts[],
    const int displs[], MPI_Datatype recvtpe, MPI_Comm comm,
    MPI_Info info, MPI_Request *request)

```

```
1 int MPI_Allgatherv_init_c(const void *sendbuf, MPI_Count sendcount,
2     MPI_Datatype sendtype, void *recvbuf,
3     const MPI_Count recvcnts[], const MPI_Aint displs[],
4     MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
5     MPI_Request *request)
6
7 int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,
8     MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
9
10 int MPI_Allreduce_c(const void *sendbuf, void *recvbuf, MPI_Count count,
11     MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
12
13 int MPI_Allreduce_init(const void *sendbuf, void *recvbuf, int count,
14     MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info,
15     MPI_Request *request)
16
17 int MPI_Allreduce_init_c(const void *sendbuf, void *recvbuf, MPI_Count count,
18     MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info,
19     MPI_Request *request)
20
21 int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
22     void *recvbuf, int recvcnt, MPI_Datatype recvtype,
23     MPI_Comm comm)
24
25 int MPI_Alltoall_c(const void *sendbuf, MPI_Count sendcount,
26     MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcnt,
27     MPI_Datatype recvtype, MPI_Comm comm)
28
29 int MPI_Alltoall_init(const void *sendbuf, int sendcount,
30     MPI_Datatype sendtype, void *recvbuf, int recvcnt,
31     MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
32     MPI_Request *request)
33
34 int MPI_Alltoall_init_c(const void *sendbuf, MPI_Count sendcount,
35     MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcnt,
36     MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
37     MPI_Request *request)
38
39 int MPI_Alltoallv(const void *sendbuf, const int sendcounts[],
40     const int sdispls[], MPI_Datatype sendtype, void *recvbuf,
41     const int recvcnts[], const int rdispls[],
42     MPI_Datatype recvtype, MPI_Comm comm)
43
44 int MPI_Alltoallv_c(const void *sendbuf, const MPI_Count sendcounts[],
45     const MPI_Aint sdispls[], MPI_Datatype sendtype, void *recvbuf,
46     const MPI_Count recvcnts[], const MPI_Aint rdispls[],
47     MPI_Datatype recvtype, MPI_Comm comm)
48
49 int MPI_Alltoallv_init(const void *sendbuf, const int sendcounts[],
50     const int sdispls[], MPI_Datatype sendtype, void *recvbuf,
51     const int recvcnts[], const int rdispls[],
52     MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
53     MPI_Request *request)
```

```

int MPI_Alltoallv_init_c(const void *sendbuf, const MPI_Count sendcounts[],      1
                        const MPI_Aint sdispls[], MPI_Datatype sendtype, void *recvbuf,      2
                        const MPI_Count recvcnts[], const MPI_Aint rdispls[],          3
                        MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,          4
                        MPI_Request *request)                                        5
                                                                                      6
int MPI_Alltoallw(const void *sendbuf, const int sendcounts[],                  7
                  const int sdispls[], const MPI_Datatype sendtypes[],          8
                  void *recvbuf, const int recvcnts[], const int rdispls[],        9
                  const MPI_Datatype recvtypes[], MPI_Comm comm)                10
                                                                                      11
int MPI_Alltoallw_c(const void *sendbuf, const MPI_Count sendcounts[],          12
                    const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],      13
                    void *recvbuf, const MPI_Count recvcnts[],                    14
                    const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],      15
                    MPI_Comm comm)                                               16
                                                                                      17
int MPI_Alltoallw_init(const void *sendbuf, const int sendcounts[],             18
                       const int sdispls[], const MPI_Datatype sendtypes[],       19
                       void *recvbuf, const int recvcnts[], const int rdispls[],   20
                       const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info, 21
                       MPI_Request *request)
                                                                                      22
int MPI_Alltoallw_init_c(const void *sendbuf, const MPI_Count sendcounts[],      23
                         const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],  24
                         void *recvbuf, const MPI_Count recvcnts[],                25
                         const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],  26
                         MPI_Comm comm, MPI_Info info, MPI_Request *request)      27
                                                                                      28
int MPI_Barrier(MPI_Comm comm)                                                  29
                                                                                      30
int MPI_Barrier_init(MPI_Comm comm, MPI_Info info, MPI_Request *request)         31
                                                                                      32
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root,          33
              MPI_Comm comm)                                                    34
                                                                                      35
int MPI_Bcast_c(void *buffer, MPI_Count count, MPI_Datatype datatype, int root,   36
                MPI_Comm comm)                                                  37
                                                                                      38
int MPI_Bcast_init(void *buffer, int count, MPI_Datatype datatype, int root,      39
                  MPI_Comm comm, MPI_Info info, MPI_Request *request)           40
                                                                                      41
int MPI_Bcast_init_c(void *buffer, MPI_Count count, MPI_Datatype datatype,        42
                    int root, MPI_Comm comm, MPI_Info info, MPI_Request *request) 43
                                                                                      44
int MPI_Exscan(const void *sendbuf, void *recvbuf, int count,                   45
               MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)                 46
                                                                                      47
int MPI_Exscan_c(const void *sendbuf, void *recvbuf, MPI_Count count,            48
                 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

```

```
1 int MPI_Exscan_init_c(const void *sendbuf, void *recvbuf, MPI_Count count,
2                     MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info,
3                     MPI_Request *request)
4
5 int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
6               void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
7               MPI_Comm comm)
8
9 int MPI_Gather_c(const void *sendbuf, MPI_Count sendcount,
10                MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
11                MPI_Datatype recvtype, int root, MPI_Comm comm)
12
13 int MPI_Gather_init(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
14                   void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
15                   MPI_Comm comm, MPI_Info info, MPI_Request *request)
16
17 int MPI_Gather_init_c(const void *sendbuf, MPI_Count sendcount,
18                     MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
19                     MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,
20                     MPI_Request *request)
21
22 int MPI_Gatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
23               void *recvbuf, const int recvcounts[], const int displs[],
24               MPI_Datatype recvtype, int root, MPI_Comm comm)
25
26 int MPI_Gatherv_c(const void *sendbuf, MPI_Count sendcount,
27                  MPI_Datatype sendtype, void *recvbuf,
28                  const MPI_Count recvcounts[], const MPI_Aint displs[],
29                  MPI_Datatype recvtype, int root, MPI_Comm comm)
30
31 int MPI_Gatherv_init(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
32                    void *recvbuf, const int recvcounts[], const int displs[],
33                    MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,
34                    MPI_Request *request)
35
36 int MPI_Gatherv_init_c(const void *sendbuf, MPI_Count sendcount,
37                       MPI_Datatype sendtype, void *recvbuf,
38                       const MPI_Count recvcounts[], const MPI_Aint displs[],
39                       MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,
40                       MPI_Request *request)
41
42 int MPI_Iallgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
43                  void *recvbuf, int recvcount, MPI_Datatype recvtype,
44                  MPI_Comm comm, MPI_Request *request)
45
46 int MPI_Iallgather_c(const void *sendbuf, MPI_Count sendcount,
47                     MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
48                     MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
49
50 int MPI_Iallgatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
51                   void *recvbuf, const int recvcounts[], const int displs[],
52                   MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
```

```

int MPI_Iallgatherv_c(const void *sendbuf, MPI_Count sendcount,      1
                    MPI_Datatype sendtype, void *recvbuf,          2
                    const MPI_Count recvcnts[], const MPI_Aint displs[], 3
                    MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) 4
                                                                    5
int MPI_Iallreduce(const void *sendbuf, void *recvbuf, int count,  6
                  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, 7
                  MPI_Request *request)                               8
                                                                    9
int MPI_Iallreduce_c(const void *sendbuf, void *recvbuf, MPI_Count count, 10
                    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, 11
                    MPI_Request *request)                             12
                                                                    13
int MPI_Ialltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype, 14
                 void *recvbuf, int recvcount, MPI_Datatype recvtype, 15
                 MPI_Comm comm, MPI_Request *request)                16
                                                                    17
int MPI_Ialltoall_c(const void *sendbuf, MPI_Count sendcount, 18
                   MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount, 19
                   MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) 20
                                                                    21
int MPI_Ialltoallv(const void *sendbuf, const int sendcounts[], 22
                  const int sdispls[], MPI_Datatype sendtype, void *recvbuf, 23
                  const int recvcnts[], const int rdispls[], 24
                  MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) 25
                                                                    26
int MPI_Ialltoallv_c(const void *sendbuf, const MPI_Count sendcounts[], 27
                   const MPI_Aint sdispls[], MPI_Datatype sendtype, void *recvbuf, 28
                   const MPI_Count recvcnts[], const MPI_Aint rdispls[], 29
                   MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) 30
                                                                    31
int MPI_Ialltoallw(const void *sendbuf, const int sendcounts[], 32
                  const int sdispls[], const MPI_Datatype sendtypes[], 33
                  void *recvbuf, const int recvcnts[], const int rdispls[], 34
                  const MPI_Datatype recvtypes[], MPI_Comm comm, 35
                  MPI_Request *request)                               36
                                                                    37
int MPI_Ialltoallw_c(const void *sendbuf, const MPI_Count sendcounts[], 38
                   const MPI_Aint sdispls[], const MPI_Datatype sendtypes[], 39
                   void *recvbuf, const MPI_Count recvcnts[], 40
                   const MPI_Aint rdispls[], const MPI_Datatype recvtypes[], 41
                   MPI_Comm comm, MPI_Request *request)              42
                                                                    43
int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)                44
                                                                    45
int MPI_Ibcast(void *buffer, int count, MPI_Datatype datatype, int root, 46
              MPI_Comm comm, MPI_Request *request)                   47
                                                                    48
int MPI_Ibcast_c(void *buffer, MPI_Count count, MPI_Datatype datatype, 49
                int root, MPI_Comm comm, MPI_Request *request)        50
                                                                    51
int MPI_Iexscan(const void *sendbuf, void *recvbuf, int count, 52
               MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, 53
               MPI_Request *request)                                    54

```

```
1         MPI_Request *request)
2
3 int MPI_Iexscan_c(const void *sendbuf, void *recvbuf, MPI_Count count,
4                 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
5                 MPI_Request *request)
6
7 int MPI_Igather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
8                void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
9                MPI_Comm comm, MPI_Request *request)
10
11 int MPI_Igather_c(const void *sendbuf, MPI_Count sendcount,
12                  MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
13                  MPI_Datatype recvtype, int root, MPI_Comm comm,
14                  MPI_Request *request)
15
16 int MPI_Igatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
17                 void *recvbuf, const int recvcnts[], const int displs[],
18                 MPI_Datatype recvtype, int root, MPI_Comm comm,
19                 MPI_Request *request)
20
21 int MPI_Igatherv_c(const void *sendbuf, MPI_Count sendcount,
22                   MPI_Datatype sendtype, void *recvbuf,
23                   const MPI_Count recvcnts[], const MPI_Aint displs[],
24                   MPI_Datatype recvtype, int root, MPI_Comm comm,
25                   MPI_Request *request)
26
27 int MPI_Ireduce(const void *sendbuf, void *recvbuf, int count,
28                MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
29                MPI_Request *request)
30
31 int MPI_Ireduce_c(const void *sendbuf, void *recvbuf, MPI_Count count,
32                  MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
33                  MPI_Request *request)
34
35 int MPI_Ireduce_scatter(const void *sendbuf, void *recvbuf,
36                         const int recvcnts[], MPI_Datatype datatype, MPI_Op op,
37                         MPI_Comm comm, MPI_Request *request)
38
39 int MPI_Ireduce_scatter_block(const void *sendbuf, void *recvbuf,
40                               int recvcount, MPI_Datatype datatype, MPI_Op op,
41                               MPI_Comm comm, MPI_Request *request)
42
43 int MPI_Ireduce_scatter_block_c(const void *sendbuf, void *recvbuf,
44                                 MPI_Count recvcount, MPI_Datatype datatype, MPI_Op op,
45                                 MPI_Comm comm, MPI_Request *request)
46
47 int MPI_Ireduce_scatter_c(const void *sendbuf, void *recvbuf,
48                           const MPI_Count recvcnts[], MPI_Datatype datatype, MPI_Op op,
49                           MPI_Comm comm, MPI_Request *request)
50
51 int MPI_Iscan(const void *sendbuf, void *recvbuf, int count,
52              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
53              MPI_Request *request)
```

```
int MPI_Iscan_c(const void *sendbuf, void *recvbuf, MPI_Count count,      1
                MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,        2
                MPI_Request *request)                                    3
                                                                    4
int MPI_Iscatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,  5
                 void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,  6
                 MPI_Comm comm, MPI_Request *request)                    7
                                                                    8
int MPI_Iscatter_c(const void *sendbuf, MPI_Count sendcount,              9
                  MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount, 10
                  MPI_Datatype recvtype, int root, MPI_Comm comm,          11
                  MPI_Request *request)                                    12
                                                                    13
int MPI_Iscatterv(const void *sendbuf, const int sendcounts[],           14
                 const int displs[], MPI_Datatype sendtype, void *recvbuf,   15
                 int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, 16
                 MPI_Request *request)                                    17
                                                                    18
int MPI_Iscatterv_c(const void *sendbuf, const MPI_Count sendcounts[],    19
                  const MPI_Aint displs[], MPI_Datatype sendtype, void *recvbuf, 20
                  MPI_Count recvcount, MPI_Datatype recvtype, int root,      21
                  MPI_Comm comm, MPI_Request *request)                    22
                                                                    23
int MPI_Op_commutative(MPI_Op op, int *commute)                          24
                                                                    25
int MPI_Op_create(MPI_User_function *user_fn, int commute, MPI_Op *op)    26
                                                                    27
int MPI_Op_create_c(MPI_User_function_c *user_fn, int commute, MPI_Op *op) 28
                                                                    29
int MPI_Op_free(MPI_Op *op)                                              30
                                                                    31
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,            32
               MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm) 33
                                                                    34
int MPI_Reduce_c(const void *sendbuf, void *recvbuf, MPI_Count count,     35
                 MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm) 36
                                                                    37
int MPI_Reduce_init(const void *sendbuf, void *recvbuf, int count,        38
                   MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm, 39
                   MPI_Info info, MPI_Request *request)                    40
                                                                    41
int MPI_Reduce_init_c(const void *sendbuf, void *recvbuf, MPI_Count count, 42
                     MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm, 43
                     MPI_Info info, MPI_Request *request)                  44
                                                                    45
int MPI_Reduce_local(const void *inbuf, void *inoutbuf, int count,        46
                    MPI_Datatype datatype, MPI_Op op)                     47
                                                                    48
int MPI_Reduce_local_c(const void *inbuf, void *inoutbuf, MPI_Count count, 49
                      MPI_Datatype datatype, MPI_Op op)                    50
                                                                    51
int MPI_Reduce_scatter(const void *sendbuf, void *recvbuf,                52
                      const int recvcounts[], MPI_Datatype datatype, MPI_Op op, 53
                      MPI_Comm comm)                                       54
```

```
1 int MPI_Reduce_scatter_block(const void *sendbuf, void *recvbuf, int recvcount,
2                             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
3
4 int MPI_Reduce_scatter_block_c(const void *sendbuf, void *recvbuf,
5                               MPI_Count recvcount, MPI_Datatype datatype, MPI_Op op,
6                               MPI_Comm comm)
7
8 int MPI_Reduce_scatter_block_init(const void *sendbuf, void *recvbuf,
9                                   int recvcount, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
10                                  MPI_Info info, MPI_Request *request)
11
12 int MPI_Reduce_scatter_block_init_c(const void *sendbuf, void *recvbuf,
13                                     MPI_Count recvcount, MPI_Datatype datatype, MPI_Op op,
14                                     MPI_Comm comm, MPI_Info info, MPI_Request *request)
15
16 int MPI_Reduce_scatter_c(const void *sendbuf, void *recvbuf,
17                          const MPI_Count recvcounts[], MPI_Datatype datatype, MPI_Op op,
18                          MPI_Comm comm)
19
20 int MPI_Reduce_scatter_init(const void *sendbuf, void *recvbuf,
21                            const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
22                            MPI_Comm comm, MPI_Info info, MPI_Request *request)
23
24 int MPI_Reduce_scatter_init_c(const void *sendbuf, void *recvbuf,
25                              const MPI_Count recvcounts[], MPI_Datatype datatype, MPI_Op op,
26                              MPI_Comm comm, MPI_Info info, MPI_Request *request)
27
28 int MPI_Scan(const void *sendbuf, void *recvbuf, int count,
29             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
30
31 int MPI_Scan_c(const void *sendbuf, void *recvbuf, MPI_Count count,
32              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
33
34 int MPI_Scan_init(const void *sendbuf, void *recvbuf, int count,
35                 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info,
36                 MPI_Request *request)
37
38 int MPI_Scan_init_c(const void *sendbuf, void *recvbuf, MPI_Count count,
39                   MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info,
40                   MPI_Request *request)
41
42 int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
43               void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
44               MPI_Comm comm)
45
46 int MPI_Scatter_c(const void *sendbuf, MPI_Count sendcount,
47                 MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
48                 MPI_Datatype recvtype, int root, MPI_Comm comm)
49
50 int MPI_Scatter_init(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
51                    void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
52                    MPI_Comm comm, MPI_Info info, MPI_Request *request)
53
54 int MPI_Scatter_init_c(const void *sendbuf, MPI_Count sendcount,
```



```

    MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcnt,
    MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,
    MPI_Request *request)
1
2
3
4
int MPI_Scatterv(const void *sendbuf, const int sendcounts[],
    const int displs[], MPI_Datatype sendtype, void *recvbuf,
    int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)
5
6
7
8
int MPI_Scatterv_c(const void *sendbuf, const MPI_Count sendcounts[],
    const MPI_Aint displs[], MPI_Datatype sendtype, void *recvbuf,
    MPI_Count recvcnt, MPI_Datatype recvtype, int root,
    MPI_Comm comm)
9
10
11
12
int MPI_Scatterv_init(const void *sendbuf, const int sendcounts[],
    const int displs[], MPI_Datatype sendtype, void *recvbuf,
    int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm,
    MPI_Info info, MPI_Request *request)
13
14
15
16
int MPI_Scatterv_init_c(const void *sendbuf, const MPI_Count sendcounts[],
    const MPI_Aint displs[], MPI_Datatype sendtype, void *recvbuf,
    MPI_Count recvcnt, MPI_Datatype recvtype, int root,
    MPI_Comm comm, MPI_Info info, MPI_Request *request)
17
18
19
20
21
22
A.3.5 Groups, Contexts, Communicators, and Caching C Bindings
23
int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)
24
25
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
26
27
int MPI_Comm_create_from_group(MPI_Group group, const char *stringtag,
    MPI_Info info, MPI_Errhandler errhandler, MPI_Comm *newcomm)
28
29
int MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag,
    MPI_Comm *newcomm)
30
31
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)
32
33
34
35
int MPI_Comm_delete_attr(MPI_Comm comm, int comm_keyval)
36
37
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
38
39
int MPI_COMM_DUP_FN(MPI_Comm oldcomm, int comm_keyval, void *extra_state,
    void *attribute_val_in, void *attribute_val_out, int *flag)
40
41
int MPI_Comm_dup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm)
42
43
int MPI_Comm_free(MPI_Comm *comm)
44
45
int MPI_Comm_free_keyval(int *comm_keyval)
46
47
int MPI_Comm_get_attr(MPI_Comm comm, int comm_keyval, void *attribute_val,
    int *flag)
48

```

```
1 int MPI_Comm_get_info(MPI_Comm comm, MPI_Info *info_used)
2
3 int MPI_Comm_get_name(MPI_Comm comm, char *comm_name, int *resultlen)
4
5 int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
6
7 int MPI_Comm_idup(MPI_Comm comm, MPI_Comm *newcomm, MPI_Request *request)
8
9 int MPI_Comm_idup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm,
10                             MPI_Request *request)
11
12 int MPI_COMM_NULL_COPY_FN(MPI_Comm oldcomm, int comm_keyval, void *extra_state,
13                             void *attribute_val_in, void *attribute_val_out, int *flag)
14
15 int MPI_COMM_NULL_DELETE_FN(MPI_Comm comm, int comm_keyval,
16                               void *attribute_val, void *extra_state)
17
18 int MPI_Comm_rank(MPI_Comm comm, int *rank)
19
20 int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
21
22 int MPI_Comm_remote_size(MPI_Comm comm, int *size)
23
24 int MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void *attribute_val)
25
26 int MPI_Comm_set_info(MPI_Comm comm, MPI_Info info)
27
28 int MPI_Comm_set_name(MPI_Comm comm, const char *comm_name)
29
30 int MPI_Comm_size(MPI_Comm comm, int *size)
31
32 int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
33
34 int MPI_Comm_split_type(MPI_Comm comm, int split_type, int key, MPI_Info info,
35                             MPI_Comm *newcomm)
36
37 int MPI_Comm_test_inter(MPI_Comm comm, int *flag)
38
39 int MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result)
40
41 int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
42                             MPI_Group *newgroup)
43
44 int MPI_Group_excl(MPI_Group group, int n, const int ranks[],
45                             MPI_Group *newgroup)
46
47 int MPI_Group_free(MPI_Group *group)
48
49 int MPI_Group_from_session_pset(MPI_Session session, const char *pset_name,
50                             MPI_Group *newgroup)
51
52 int MPI_Group_incl(MPI_Group group, int n, const int ranks[],
53                             MPI_Group *newgroup)
54
55 int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,
56                             MPI_Group *newgroup)
57
58 int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],
59                             MPI_Group *newgroup)
```

```

int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],      1
                        MPI_Group *newgroup)                          2
int MPI_Group_rank(MPI_Group group, int *rank)                        3
int MPI_Group_size(MPI_Group group, int *size)                       4
int MPI_Group_translate_ranks(MPI_Group group1, int n, const int ranks1[], 5
                              MPI_Group group2, int ranks2[])        6
int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup) 7
int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,      8
                        MPI_Comm peer_comm, int remote_leader, int tag, 9
                        MPI_Comm *newintercomm)                      10
int MPI_Intercomm_create_from_groups(MPI_Group local_group, int local_leader, 11
                                     MPI_Group remote_group, int remote_leader, const char *stringtag, 12
                                     MPI_Info info, MPI_Errhandler errhandler, MPI_Comm *newintercomm) 13
int MPI_Intercomm_merge(MPI_Comm intercomm, int high, MPI_Comm *newintracomm) 14
int MPI_Type_create_keyval(MPI_Type_copy_attr_function *type_copy_attr_fn, 15
                          MPI_Type_delete_attr_function *type_delete_attr_fn, 16
                          int *type_keyval, void *extra_state)      17
int MPI_Type_delete_attr(MPI_Datatype datatype, int type_keyval)    18
int MPI_TYPE_DUP_FN(MPI_Datatype oldtype, int type_keyval, void *extra_state, 19
                   void *attribute_val_in, void *attribute_val_out, int *flag) 20
int MPI_Type_free_keyval(int *type_keyval)                          21
int MPI_Type_get_attr(MPI_Datatype datatype, int type_keyval,      22
                     void *attribute_val, int *flag)                23
int MPI_Type_get_name(MPI_Datatype datatype, char *type_name, int *resultlen) 24
int MPI_TYPE_NULL_COPY_FN(MPI_Datatype oldtype, int type_keyval,    25
                          void *extra_state, void *attribute_val_in, 26
                          void *attribute_val_out, int *flag)      27
int MPI_TYPE_NULL_DELETE_FN(MPI_Datatype datatype, int type_keyval, 28
                            void *attribute_val, void *extra_state) 29
int MPI_Type_set_attr(MPI_Datatype datatype, int type_keyval,      30
                     void *attribute_val)                          31
int MPI_Type_set_name(MPI_Datatype datatype, const char *type_name) 32
int MPI_Win_create_keyval(MPI_Win_copy_attr_function *win_copy_attr_fn, 33
                          MPI_Win_delete_attr_function *win_delete_attr_fn, 34
                          int *win_keyval, void *extra_state)      35
int MPI_Win_delete_attr(MPI_Win win, int win_keyval)                36

```

```

1  int MPI_WIN_DUP_FN(MPI_Win oldwin, int win_keyval, void *extra_state,
2      void *attribute_val_in, void *attribute_val_out, int *flag)
3
4  int MPI_Win_free_keyval(int *win_keyval)
5
6  int MPI_Win_get_attr(MPI_Win win, int win_keyval, void *attribute_val,
7      int *flag)
8
9  int MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)
10
11 int MPI_WIN_NULL_COPY_FN(MPI_Win oldwin, int win_keyval, void *extra_state,
12     void *attribute_val_in, void *attribute_val_out, int *flag)
13
14 int MPI_WIN_NULL_DELETE_FN(MPI_Win win, int win_keyval, void *attribute_val,
15     void *extra_state)
16
17 int MPI_Win_set_attr(MPI_Win win, int win_keyval, void *attribute_val)
18
19 int MPI_Win_set_name(MPI_Win win, const char *win_name)
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

```

A.3.6 Process Topologies C Bindings

```

19  int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])
20
21  int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[],
22     const int periods[], int reorder, MPI_Comm *comm_cart)
23
24  int MPI_Cart_get(MPI_Comm comm, int maxdims, int dims[], int periods[],
25     int coords[])
26
27  int MPI_Cart_map(MPI_Comm comm, int ndims, const int dims[],
28     const int periods[], int *newrank)
29
30  int MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank)
31
32  int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source,
33     int *rank_dest)
34
35  int MPI_Cart_sub(MPI_Comm comm, const int remain_dims[], MPI_Comm *newcomm)
36
37  int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
38
39  int MPI_Dims_create(int nnodes, int ndims, int dims[])
40
41  int MPI_Dist_graph_create(MPI_Comm comm_old, int n, const int sources[],
42     const int degrees[], const int destinations[],
43     const int weights[], MPI_Info info, int reorder,
44     MPI_Comm *comm_dist_graph)
45
46  int MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree,
47     const int sources[], const int sourceweights[], int outdegree,
48     const int destinations[], const int destweights[], MPI_Info info,
49     int reorder, MPI_Comm *comm_dist_graph)
50
51  int MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[],
52     int sourceweights[], int maxoutdegree, int destinations[],
53     int destweights[])

```

```

        int destweights[])
1
2
int MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree,
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
        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_allgather_c(const void *sendbuf, MPI_Count sendcount,
        MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
        MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
int MPI_Ineighbor_allgather_v(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_allgather_v_c(const void *sendbuf, MPI_Count sendcount,
        MPI_Datatype sendtype, void *recvbuf,
        const MPI_Count recvcounts[], const MPI_Aint 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_alltoall_c(const void *sendbuf, MPI_Count sendcount,
        MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
        MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
int MPI_Ineighbor_alltoall_v(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_alltoall_v_c(const void *sendbuf,
        const MPI_Count sendcounts[], const MPI_Aint sdispls[],
        MPI_Datatype sendtype, void *recvbuf,
```

```
1         const MPI_Count recvcounts[], const MPI_Aint rdispls[],
2         MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
3
4     int MPI_Ineighbor_alltoallw(const void *sendbuf, const int sendcounts[],
5         const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],
6         void *recvbuf, const int recvcounts[], const MPI_Aint rdispls[],
7         const MPI_Datatype recvtypes[], MPI_Comm comm,
8         MPI_Request *request)
9
10    int MPI_Ineighbor_alltoallw_c(const void *sendbuf,
11        const MPI_Count sendcounts[], const MPI_Aint sdispls[],
12        const MPI_Datatype sendtypes[], void *recvbuf,
13        const MPI_Count recvcounts[], const MPI_Aint rdispls[],
14        const MPI_Datatype recvtypes[], MPI_Comm comm,
15        MPI_Request *request)
16
17    int MPI_Neighbor_allgather(const void *sendbuf, int sendcount,
18        MPI_Datatype sendtype, void *recvbuf, int recvcount,
19        MPI_Datatype recvtype, MPI_Comm comm)
20
21    int MPI_Neighbor_allgather_c(const void *sendbuf, MPI_Count sendcount,
22        MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
23        MPI_Datatype recvtype, MPI_Comm comm)
24
25    int MPI_Neighbor_allgather_init(const void *sendbuf, int sendcount,
26        MPI_Datatype sendtype, void *recvbuf, int recvcount,
27        MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
28        MPI_Request *request)
29
30    int MPI_Neighbor_allgather_init_c(const void *sendbuf, MPI_Count sendcount,
31        MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
32        MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
33        MPI_Request *request)
34
35    int MPI_Neighbor_allgatherv(const void *sendbuf, int sendcount,
36        MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
37        const int displs[], MPI_Datatype recvtype, MPI_Comm comm)
38
39    int MPI_Neighbor_allgatherv_c(const void *sendbuf, MPI_Count sendcount,
40        MPI_Datatype sendtype, void *recvbuf,
41        const MPI_Count recvcounts[], const MPI_Aint displs[],
42        MPI_Datatype recvtype, MPI_Comm comm)
43
44    int MPI_Neighbor_allgatherv_init(const void *sendbuf, int sendcount,
45        MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
46        const int displs[], MPI_Datatype recvtype, MPI_Comm comm,
47        MPI_Info info, MPI_Request *request)
48
49    int MPI_Neighbor_allgatherv_init_c(const void *sendbuf, MPI_Count sendcount,
50        MPI_Datatype sendtype, void *recvbuf,
51        const MPI_Count recvcounts[], const MPI_Aint displs[],
52        MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
```

```

        MPI_Request *request) 1
int MPI_Neighbor_alltoall(const void *sendbuf, int sendcount, 2
        MPI_Datatype sendtype, void *recvbuf, int recvcount, 3
        MPI_Datatype recvtype, MPI_Comm comm) 4
int MPI_Neighbor_alltoall_c(const void *sendbuf, MPI_Count sendcount, 5
        MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount, 6
        MPI_Datatype recvtype, MPI_Comm comm) 7
int MPI_Neighbor_alltoall_init(const void *sendbuf, int sendcount, 8
        MPI_Datatype sendtype, void *recvbuf, int recvcount, 9
        MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, 10
        MPI_Request *request) 11
int MPI_Neighbor_alltoall_init_c(const void *sendbuf, MPI_Count sendcount, 12
        MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount, 13
        MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, 14
        MPI_Request *request) 15
int MPI_Neighbor_alltoallv(const void *sendbuf, const int sendcounts[], 16
        const int sdispls[], MPI_Datatype sendtype, void *recvbuf, 17
        const int rcvcounts[], const int rdispls[], 18
        MPI_Datatype recvtype, MPI_Comm comm) 19
int MPI_Neighbor_alltoallv_c(const void *sendbuf, const MPI_Count sendcounts[], 20
        const MPI_Aint sdispls[], MPI_Datatype sendtype, void *recvbuf, 21
        const MPI_Count rcvcounts[], const MPI_Aint rdispls[], 22
        MPI_Datatype recvtype, MPI_Comm comm) 23
int MPI_Neighbor_alltoallv_init(const void *sendbuf, const int sendcounts[], 24
        const int sdispls[], MPI_Datatype sendtype, void *recvbuf, 25
        const int rcvcounts[], const int rdispls[], 26
        MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, 27
        MPI_Request *request) 28
int MPI_Neighbor_alltoallv_init_c(const void *sendbuf, 29
        const MPI_Count sendcounts[], const MPI_Aint sdispls[], 30
        MPI_Datatype sendtype, void *recvbuf, 31
        const MPI_Count rcvcounts[], const MPI_Aint rdispls[], 32
        MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, 33
        MPI_Request *request) 34
int MPI_Neighbor_alltoallw(const void *sendbuf, const int sendcounts[], 35
        const MPI_Aint sdispls[], const MPI_Datatype sendtypes[], 36
        void *recvbuf, const int rcvcounts[], const MPI_Aint rdispls[], 37
        const MPI_Datatype recvtypes[], MPI_Comm comm) 38
int MPI_Neighbor_alltoallw_c(const void *sendbuf, const MPI_Count sendcounts[], 39
        const MPI_Aint sdispls[], const MPI_Datatype sendtypes[], 40
        void *recvbuf, const MPI_Count rcvcounts[], 41
        const MPI_Aint rdispls[], const MPI_Datatype recvtypes[], 42
        MPI_Comm comm) 43
int MPI_Neighbor_alltoallw_init(const void *sendbuf, const int sendcounts[], 44
        const int sdispls[], MPI_Datatype sendtype, void *recvbuf, 45
        const int rcvcounts[], const int rdispls[], 46
        MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, 47
        MPI_Request *request) 48
int MPI_Neighbor_alltoallw_init_c(const void *sendbuf, MPI_Count sendcounts[], 48
        MPI_Aint sdispls[], MPI_Datatype sendtype, void *recvbuf,
        MPI_Count rcvcounts[], MPI_Aint rdispls[],
        MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
        MPI_Request *request)

```

```
1         MPI_Comm comm)
2
3 int MPI_Neighbor_alltoallw_init(const void *sendbuf, const int sendcounts[],
4         const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],
5         void *recvbuf, const int recvcounts[], const MPI_Aint rdispls[],
6         const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info,
7         MPI_Request *request)
8
9 int MPI_Neighbor_alltoallw_init_c(const void *sendbuf,
10        const MPI_Count sendcounts[], const MPI_Aint sdispls[],
11        const MPI_Datatype sendtypes[], void *recvbuf,
12        const MPI_Count recvcounts[], const MPI_Aint rdispls[],
13        const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info,
14        MPI_Request *request)
15
16 int MPI_Topo_test(MPI_Comm comm, int *status)
17
18 A.3.7 MPI Environmental Management C Bindings
19
20 int MPI_Add_error_class(int *errorclass)
21
22 int MPI_Add_error_code(int errorclass, int *errorcode)
23
24 int MPI_Add_error_string(int errorcode, const char *string)
25
26 int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)
27
28 int MPI_Comm_call_errhandler(MPI_Comm comm, int errorcode)
29
30 int MPI_Comm_create_errhandler(
31     MPI_Comm_errhandler_function *comm_errhandler_fn,
32     MPI_Errhandler *errhandler)
33
34 int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)
35
36 int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)
37
38 int MPI_Errhandler_free(MPI_Errhandler *errhandler)
39
40 int MPI_Error_class(int errorcode, int *errorclass)
41
42 int MPI_Error_string(int errorcode, char *string, int *resultlen)
43
44 int MPI_File_call_errhandler(MPI_File fh, int errorcode)
45
46 int MPI_File_create_errhandler(
47     MPI_File_errhandler_function *file_errhandler_fn,
48     MPI_Errhandler *errhandler)
49
50 int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)
51
52 int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)
53
54 int MPI_Free_mem(void *base)
55
56 int MPI_Get_library_version(char *version, int *resultlen)
```



```

int MPI_Get_processor_name(char *name, int *resultlen) 1
int MPI_Get_version(int *version, int *subversion) 2
int MPI_Session_call_errhandler(MPI_Session session, int errorcode) 3
int MPI_Session_create_errhandler( 4
    MPI_Session_errhandler_function *session_errhandler_fn, 5
    MPI_Errhandler *errhandler) 6
int MPI_Session_get_errhandler(MPI_Session session, MPI_Errhandler *errhandler) 7
int MPI_Session_set_errhandler(MPI_Session session, MPI_Errhandler errhandler) 8
int MPI_Win_call_errhandler(MPI_Win win, int errorcode) 9
int MPI_Win_create_errhandler(MPI_Win_errhandler_function *win_errhandler_fn, 10
    MPI_Errhandler *errhandler) 11
int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler) 12
int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler) 13
double MPI_Wtick(void) 14
double MPI_Wtime(void) 15

```

A.3.8 The Info Object C Bindings

```

int MPI_Info_create(MPI_Info *info) 16
int MPI_Info_create_env(int argc, char *argv[], MPI_Info *info) 17
int MPI_Info_delete(MPI_Info info, const char *key) 18
int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo) 19
int MPI_Info_free(MPI_Info *info) 20
int MPI_Info_get_nkeys(MPI_Info info, int *nkeys) 21
int MPI_Info_get_nthkey(MPI_Info info, int n, char *key) 22
int MPI_Info_get_string(MPI_Info info, const char *key, int *buflen, 23
    char *value, int *flag) 24
int MPI_Info_set(MPI_Info info, const char *key, const char *value) 25

```

A.3.9 Process Creation and Management C Bindings

```

int MPI_Abort(MPI_Comm comm, int errorcode) 26
int MPI_Close_port(const char *port_name) 27
int MPI_Comm_accept(const char *port_name, MPI_Info info, int root, 28
    MPI_Comm comm, MPI_Comm *newcomm) 29

```

```
1 int MPI_Comm_connect(const char *port_name, MPI_Info info, int root,
2                     MPI_Comm comm, MPI_Comm *newcomm)
3
4 int MPI_Comm_disconnect(MPI_Comm *comm)
5
6 int MPI_Comm_get_parent(MPI_Comm *parent)
7
8 int MPI_Comm_join(int fd, MPI_Comm *intercomm)
9
10 int MPI_Comm_spawn(const char *command, char *argv[], int maxprocs,
11                   MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm,
12                   int array_of_errcodes[])
13
14 int MPI_Comm_spawn_multiple(int count, char *array_of_commands[],
15                             char **array_of_argv[], const int array_of_maxprocs[],
16                             const MPI_Info array_of_info[], int root, MPI_Comm comm,
17                             MPI_Comm *intercomm, int array_of_errcodes[])
18
19 int MPI_Finalize(void)
20
21 int MPI_Finalized(int *flag)
22
23 int MPI_Init(int *argc, char ***argv)
24
25 int MPI_Init_thread(int *argc, char ***argv, int required, int *provided)
26
27 int MPI_Initialized(int *flag)
28
29 int MPI_Is_thread_main(int *flag)
30
31 int MPI_Lookup_name(const char *service_name, MPI_Info info, char *port_name)
32
33 int MPI_Open_port(MPI_Info info, char *port_name)
34
35 int MPI_Publish_name(const char *service_name, MPI_Info info,
36                    const char *port_name)
37
38 int MPI_Query_thread(int *provided)
39
40 int MPI_Session_finalize(MPI_Session *session)
41
42 int MPI_Session_get_info(MPI_Session session, MPI_Info *info_used)
43
44 int MPI_Session_get_nth_pset(MPI_Session session, MPI_Info info, int n,
45                             int *pset_len, char *pset_name)
46
47 int MPI_Session_get_num_psets(MPI_Session session, MPI_Info info,
48                             int *npset_names)
49
50 int MPI_Session_get_pset_info(MPI_Session session, const char *pset_name,
51                             MPI_Info *info)
52
53 int MPI_Session_init(MPI_Info info, MPI_Errhandler errhandler,
54                    MPI_Session *session)
55
56 int MPI_Unpublish_name(const char *service_name, MPI_Info info,
57                      const char *port_name)
58
```

A.3.10 One-Sided Communications C Bindings

```
1
2
3 int MPI_Accumulate(const void *origin_addr, int origin_count,
4                 MPI_Datatype origin_datatype, int target_rank,
5                 MPI_Aint target_disp, int target_count,
6                 MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
7
8 int MPI_Accumulate_c(const void *origin_addr, MPI_Count origin_count,
9                   MPI_Datatype origin_datatype, int target_rank,
10                  MPI_Aint target_disp, MPI_Count target_count,
11                  MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
12
13 int MPI_Compare_and_swap(const void *origin_addr, const void *compare_addr,
14                        void *result_addr, MPI_Datatype datatype, int target_rank,
15                        MPI_Aint target_disp, MPI_Win win)
16
17 int MPI_Fetch_and_op(const void *origin_addr, void *result_addr,
18                   MPI_Datatype datatype, int target_rank, MPI_Aint target_disp,
19                   MPI_Op op, MPI_Win win)
20
21 int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype origin_datatype,
22            int target_rank, MPI_Aint target_disp, int target_count,
23            MPI_Datatype target_datatype, MPI_Win win)
24
25 int MPI_Get_accumulate(const void *origin_addr, int origin_count,
26                      MPI_Datatype origin_datatype, void *result_addr,
27                      int result_count, MPI_Datatype result_datatype, int target_rank,
28                      MPI_Aint target_disp, int target_count,
29                      MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
30
31 int MPI_Get_accumulate_c(const void *origin_addr, MPI_Count origin_count,
32                        MPI_Datatype origin_datatype, void *result_addr,
33                        MPI_Count result_count, MPI_Datatype result_datatype,
34                        int target_rank, MPI_Aint target_disp, MPI_Count target_count,
35                        MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
36
37 int MPI_Get_c(void *origin_addr, MPI_Count origin_count,
38             MPI_Datatype origin_datatype, int target_rank,
39             MPI_Aint target_disp, MPI_Count target_count,
40             MPI_Datatype target_datatype, MPI_Win win)
41
42 int MPI_Put(const void *origin_addr, int origin_count,
43           MPI_Datatype origin_datatype, int target_rank,
44           MPI_Aint target_disp, int target_count,
45           MPI_Datatype target_datatype, MPI_Win win)
46
47 int MPI_Put_c(const void *origin_addr, MPI_Count origin_count,
48             MPI_Datatype origin_datatype, int target_rank,
49             MPI_Aint target_disp, MPI_Count target_count,
50             MPI_Datatype target_datatype, MPI_Win win)
51
52 int MPI_Raccumulate(const void *origin_addr, int origin_count,
53                  MPI_Datatype origin_datatype, int target_rank,
```

```
1         MPI_Aint target_disp, int target_count,
2         MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
3         MPI_Request *request)
4
5 int MPI_Raccumulate_c(const void *origin_addr, MPI_Count origin_count,
6         MPI_Datatype origin_datatype, int target_rank,
7         MPI_Aint target_disp, MPI_Count target_count,
8         MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
9         MPI_Request *request)
10
11 int MPI_Rget(void *origin_addr, int origin_count, MPI_Datatype origin_datatype,
12         int target_rank, MPI_Aint target_disp, int target_count,
13         MPI_Datatype target_datatype, MPI_Win win, MPI_Request *request)
14
15 int MPI_Rget_accumulate(const void *origin_addr, int origin_count,
16         MPI_Datatype origin_datatype, void *result_addr,
17         int result_count, MPI_Datatype result_datatype, int target_rank,
18         MPI_Aint target_disp, int target_count,
19         MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
20         MPI_Request *request)
21
22 int MPI_Rget_accumulate_c(const void *origin_addr, MPI_Count origin_count,
23         MPI_Datatype origin_datatype, void *result_addr,
24         MPI_Count result_count, MPI_Datatype result_datatype,
25         int target_rank, MPI_Aint target_disp, MPI_Count target_count,
26         MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
27         MPI_Request *request)
28
29 int MPI_Rget_c(void *origin_addr, MPI_Count origin_count,
30         MPI_Datatype origin_datatype, int target_rank,
31         MPI_Aint target_disp, MPI_Count target_count,
32         MPI_Datatype target_datatype, MPI_Win win, MPI_Request *request)
33
34 int MPI_Rput(const void *origin_addr, int origin_count,
35         MPI_Datatype origin_datatype, int target_rank,
36         MPI_Aint target_disp, int target_count,
37         MPI_Datatype target_datatype, MPI_Win win, MPI_Request *request)
38
39 int MPI_Rput_c(const void *origin_addr, MPI_Count origin_count,
40         MPI_Datatype origin_datatype, int target_rank,
41         MPI_Aint target_disp, MPI_Count target_count,
42         MPI_Datatype target_datatype, MPI_Win win, MPI_Request *request)
43
44 int MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info,
45         MPI_Comm comm, void *baseptr, MPI_Win *win)
46
47 int MPI_Win_allocate_c(MPI_Aint size, MPI_Aint disp_unit, MPI_Info info,
48         MPI_Comm comm, void *baseptr, MPI_Win *win)
49
50 int MPI_Win_allocate_shared(MPI_Aint size, int disp_unit, MPI_Info info,
51         MPI_Comm comm, void *baseptr, MPI_Win *win)
52
53 int MPI_Win_allocate_shared_c(MPI_Aint size, MPI_Aint disp_unit, MPI_Info info,
```

```
        MPI_Comm comm, void *baseptr, MPI_Win *win) 1
int MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size) 2
int MPI_Win_complete(MPI_Win win) 3
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info, 4
        MPI_Comm comm, MPI_Win *win) 5
int MPI_Win_create_c(void *base, MPI_Aint size, MPI_Aint disp_unit, 6
        MPI_Info info, MPI_Comm comm, MPI_Win *win) 7
int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win) 8
int MPI_Win_detach(MPI_Win win, const void *base) 9
int MPI_Win_fence(int assert, MPI_Win win) 10
int MPI_Win_flush(int rank, MPI_Win win) 11
int MPI_Win_flush_all(MPI_Win win) 12
int MPI_Win_flush_local(int rank, MPI_Win win) 13
int MPI_Win_flush_local_all(MPI_Win win) 14
int MPI_Win_free(MPI_Win *win) 15
int MPI_Win_get_group(MPI_Win win, MPI_Group *group) 16
int MPI_Win_get_info(MPI_Win win, MPI_Info *info_used) 17
int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win) 18
int MPI_Win_lock_all(int assert, MPI_Win win) 19
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win) 20
int MPI_Win_set_info(MPI_Win win, MPI_Info info) 21
int MPI_Win_shared_query(MPI_Win win, int rank, MPI_Aint *size, int *disp_unit, 22
        void *baseptr) 23
int MPI_Win_shared_query_c(MPI_Win win, int rank, MPI_Aint *size, 24
        MPI_Aint *disp_unit, void *baseptr) 25
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win) 26
int MPI_Win_sync(MPI_Win win) 27
int MPI_Win_test(MPI_Win win, int *flag) 28
int MPI_Win_unlock(int rank, MPI_Win win) 29
int MPI_Win_unlock_all(MPI_Win win) 30
int MPI_Win_wait(MPI_Win win) 31
```

1 A.3.11 External Interfaces C Bindings

```
2 int MPI_Grequest_complete(MPI_Request request)
3
4 int MPI_Grequest_start(MPI_Grequest_query_function *query_fn,
5     MPI_Grequest_free_function *free_fn,
6     MPI_Grequest_cancel_function *cancel_fn, void *extra_state,
7     MPI_Request *request)
```

```
8
9 int MPI_Status_set_cancelled(MPI_Status *status, int flag)
```

```
10 int MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype,
11     int count)
```

```
12
13 int MPI_Status_set_elements_c(MPI_Status *status, MPI_Datatype datatype,
14     MPI_Count count)
```

```
15 int MPI_Status_set_elements_x(MPI_Status *status, MPI_Datatype datatype,
16     MPI_Count count)
```

18 A.3.12 I/O C Bindings

```
19
20 int MPI_CONVERSION_FN_NULL(void *userbuf, MPI_Datatype datatype, int count,
21     void *filebuf, MPI_Offset position, void *extra_state)
```

```
22
23 int MPI_CONVERSION_FN_NULL_C(void *userbuf, MPI_Datatype datatype,
24     MPI_Count count, void *filebuf, MPI_Offset position,
25     void *extra_state)
```

```
26
27 int MPI_File_close(MPI_File *fh)
```

```
28 int MPI_File_delete(const char *filename, MPI_Info info)
```

```
29
30 int MPI_File_get_amode(MPI_File fh, int *amode)
```

```
31 int MPI_File_get_atomicity(MPI_File fh, int *flag)
```

```
32
33 int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset, MPI_Offset *disp)
```

```
34 int MPI_File_get_group(MPI_File fh, MPI_Group *group)
```

```
35
36 int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)
```

```
37 int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)
```

```
38
39 int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)
```

```
40 int MPI_File_get_size(MPI_File fh, MPI_Offset *size)
```

```
41
42 int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype,
43     MPI_Aint *extent)
```

```
44 int MPI_File_get_type_extent_c(MPI_File fh, MPI_Datatype datatype,
45     MPI_Count *extent)
```

```
46
47 int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype,
48     MPI_Datatype *filetype, char *datarep)
```

```
int MPI_File_iread(MPI_File fh, void *buf, int count, MPI_Datatype datatype, 1
                    MPI_Request *request) 2
int MPI_File_iread_all(MPI_File fh, void *buf, int count, 3
                       MPI_Datatype datatype, MPI_Request *request) 4
int MPI_File_iread_all_c(MPI_File fh, void *buf, MPI_Count count, 5
                          MPI_Datatype datatype, MPI_Request *request) 6
int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count, 7
                      MPI_Datatype datatype, MPI_Request *request) 8
int MPI_File_iread_at_all(MPI_File fh, MPI_Offset offset, void *buf, int count, 9
                           MPI_Datatype datatype, MPI_Request *request) 10
int MPI_File_iread_at_all_c(MPI_File fh, MPI_Offset offset, void *buf, 11
                             MPI_Count count, MPI_Datatype datatype, MPI_Request *request) 12
int MPI_File_iread_at_c(MPI_File fh, MPI_Offset offset, void *buf, 13
                        MPI_Count count, MPI_Datatype datatype, MPI_Request *request) 14
int MPI_File_iread_c(MPI_File fh, void *buf, MPI_Count count, 15
                     MPI_Datatype datatype, MPI_Request *request) 16
int MPI_File_iread_shared(MPI_File fh, void *buf, int count, 17
                           MPI_Datatype datatype, MPI_Request *request) 18
int MPI_File_iread_shared_c(MPI_File fh, void *buf, MPI_Count count, 19
                             MPI_Datatype datatype, MPI_Request *request) 20
int MPI_File_iwrite(MPI_File fh, const void *buf, int count, 21
                    MPI_Datatype datatype, MPI_Request *request) 22
int MPI_File_iwrite_all(MPI_File fh, const void *buf, int count, 23
                        MPI_Datatype datatype, MPI_Request *request) 24
int MPI_File_iwrite_all_c(MPI_File fh, const void *buf, MPI_Count count, 25
                           MPI_Datatype datatype, MPI_Request *request) 26
int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, const void *buf, 27
                       int count, MPI_Datatype datatype, MPI_Request *request) 28
int MPI_File_iwrite_at_all(MPI_File fh, MPI_Offset offset, const void *buf, 29
                            int count, MPI_Datatype datatype, MPI_Request *request) 30
int MPI_File_iwrite_at_all_c(MPI_File fh, MPI_Offset offset, const void *buf, 31
                              MPI_Count count, MPI_Datatype datatype, MPI_Request *request) 32
int MPI_File_iwrite_at_c(MPI_File fh, MPI_Offset offset, const void *buf, 33
                          MPI_Count count, MPI_Datatype datatype, MPI_Request *request) 34
int MPI_File_iwrite_c(MPI_File fh, const void *buf, MPI_Count count, 35
                      MPI_Datatype datatype, MPI_Request *request) 36
int MPI_File_iwrite_shared(MPI_File fh, const void *buf, int count, 37
                           MPI_Datatype datatype, MPI_Request *request) 38
int MPI_File_iwrite_shared_c(MPI_File fh, const void *buf, MPI_Count count, 39
                              MPI_Datatype datatype, MPI_Request *request) 40
int MPI_File_iwrite_at_c(MPI_File fh, MPI_Offset offset, const void *buf, 41
                          MPI_Count count, MPI_Datatype datatype, MPI_Request *request) 42
int MPI_File_iwrite_c(MPI_File fh, const void *buf, MPI_Count count, 43
                      MPI_Datatype datatype, MPI_Request *request) 44
int MPI_File_iwrite_shared(MPI_File fh, const void *buf, int count, 45
                            MPI_Datatype datatype, MPI_Request *request) 46
int MPI_File_iwrite_shared_c(MPI_File fh, const void *buf, MPI_Count count, 47
                              MPI_Datatype datatype, MPI_Request *request) 48
```

```
1 int MPI_File_iwrite_shared_c(MPI_File fh, const void *buf, MPI_Count count,
2     MPI_Datatype datatype, MPI_Request *request)
3
4 int MPI_File_open(MPI_Comm comm, const char *filename, int amode,
5     MPI_Info info, MPI_File *fh)
6
7 int MPI_File_preallocate(MPI_File fh, MPI_Offset size)
8
9 int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
10     MPI_Status *status)
11
12 int MPI_File_read_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
13     MPI_Status *status)
14
15 int MPI_File_read_all_begin(MPI_File fh, void *buf, int count,
16     MPI_Datatype datatype)
17
18 int MPI_File_read_all_begin_c(MPI_File fh, void *buf, MPI_Count count,
19     MPI_Datatype datatype)
20
21 int MPI_File_read_all_c(MPI_File fh, void *buf, MPI_Count count,
22     MPI_Datatype datatype, MPI_Status *status)
23
24 int MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status)
25
26 int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
27     MPI_Datatype datatype, MPI_Status *status)
28
29 int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf, int count,
30     MPI_Datatype datatype, MPI_Status *status)
31
32 int MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,
33     int count, MPI_Datatype datatype)
34
35 int MPI_File_read_at_all_begin_c(MPI_File fh, MPI_Offset offset, void *buf,
36     MPI_Count count, MPI_Datatype datatype)
37
38 int MPI_File_read_at_all_c(MPI_File fh, MPI_Offset offset, void *buf,
39     MPI_Count count, MPI_Datatype datatype, MPI_Status *status)
40
41 int MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)
42
43 int MPI_File_read_at_c(MPI_File fh, MPI_Offset offset, void *buf,
44     MPI_Count count, MPI_Datatype datatype, MPI_Status *status)
45
46 int MPI_File_read_c(MPI_File fh, void *buf, MPI_Count count,
47     MPI_Datatype datatype, MPI_Status *status)
48
49 int MPI_File_read_ordered(MPI_File fh, void *buf, int count,
50     MPI_Datatype datatype, MPI_Status *status)
51
52 int MPI_File_read_ordered_begin(MPI_File fh, void *buf, int count,
53     MPI_Datatype datatype)
54
55 int MPI_File_read_ordered_begin_c(MPI_File fh, void *buf, MPI_Count count,
56     MPI_Datatype datatype)
```



```
int MPI_File_read_ordered_c(MPI_File fh, void *buf, MPI_Count count,      1
                           MPI_Datatype datatype, MPI_Status *status)      2
                                                                    3
int MPI_File_read_ordered_end(MPI_File fh, void *buf, MPI_Status *status)  4
                                                                    5
int MPI_File_read_shared(MPI_File fh, void *buf, int count,              6
                        MPI_Datatype datatype, MPI_Status *status)        7
                                                                    8
int MPI_File_read_shared_c(MPI_File fh, void *buf, MPI_Count count,      9
                          MPI_Datatype datatype, MPI_Status *status)     10
                                                                    11
int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)            12
                                                                    13
int MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence)     14
                                                                    15
int MPI_File_set_atomicsity(MPI_File fh, int flag)                       16
                                                                    17
int MPI_File_set_info(MPI_File fh, MPI_Info info)                        18
                                                                    19
int MPI_File_set_size(MPI_File fh, MPI_Offset size)                      20
                                                                    21
int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,  22
                    MPI_Datatype filetype, const char *datarep, MPI_Info info) 23
                                                                    24
int MPI_File_sync(MPI_File fh)                                           25
                                                                    26
int MPI_File_write(MPI_File fh, const void *buf, int count,              27
                  MPI_Datatype datatype, MPI_Status *status)            28
                                                                    29
int MPI_File_write_all(MPI_File fh, const void *buf, int count,          30
                      MPI_Datatype datatype, MPI_Status *status)        31
                                                                    32
int MPI_File_write_all_begin(MPI_File fh, const void *buf, int count,    33
                             MPI_Datatype datatype)                     34
                                                                    35
int MPI_File_write_all_begin_c(MPI_File fh, const void *buf, MPI_Count count, 36
                              MPI_Datatype datatype)                     37
                                                                    38
int MPI_File_write_all_c(MPI_File fh, const void *buf, MPI_Count count,  39
                        MPI_Datatype datatype, MPI_Status *status)      40
                                                                    41
int MPI_File_write_all_end(MPI_File fh, const void *buf, MPI_Status *status) 42
                                                                    43
int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf,   44
                    int count, MPI_Datatype datatype, MPI_Status *status) 45
                                                                    46
int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf, 47
                        int count, MPI_Datatype datatype, MPI_Status *status) 48
                                                                    49
int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset,          50
                              const void *buf, int count, MPI_Datatype datatype) 51
                                                                    52
int MPI_File_write_at_all_begin_c(MPI_File fh, MPI_Offset offset,        53
                                  const void *buf, MPI_Count count, MPI_Datatype datatype) 54
                                                                    55
int MPI_File_write_at_all_c(MPI_File fh, MPI_Offset offset, const void *buf, 56
                            MPI_Count count, MPI_Datatype datatype, MPI_Status *status) 57
                                                                    58
```

```
1 int MPI_File_write_at_all_end(MPI_File fh, const void *buf, MPI_Status *status)
2
3 int MPI_File_write_at_c(MPI_File fh, MPI_Offset offset, const void *buf,
4     MPI_Count count, MPI_Datatype datatype, MPI_Status *status)
5
6 int MPI_File_write_c(MPI_File fh, const void *buf, MPI_Count count,
7     MPI_Datatype datatype, MPI_Status *status)
8
9 int MPI_File_write_ordered(MPI_File fh, const void *buf, int count,
10     MPI_Datatype datatype, MPI_Status *status)
11
12 int MPI_File_write_ordered_begin(MPI_File fh, const void *buf, int count,
13     MPI_Datatype datatype)
14
15 int MPI_File_write_ordered_begin_c(MPI_File fh, const void *buf,
16     MPI_Count count, MPI_Datatype datatype)
17
18 int MPI_File_write_ordered_c(MPI_File fh, const void *buf, MPI_Count count,
19     MPI_Datatype datatype, MPI_Status *status)
20
21 int MPI_File_write_ordered_end(MPI_File fh, const void *buf,
22     MPI_Status *status)
23
24 int MPI_File_write_shared(MPI_File fh, const void *buf, int count,
25     MPI_Datatype datatype, MPI_Status *status)
26
27 int MPI_File_write_shared_c(MPI_File fh, const void *buf, MPI_Count count,
28     MPI_Datatype datatype, MPI_Status *status)
29
30 int MPI_Register_datarep(const char *datarep,
31     MPI_Datarep_conversion_function *read_conversion_fn,
32     MPI_Datarep_conversion_function *write_conversion_fn,
33     MPI_Datarep_extent_function *dtype_file_extent_fn,
34     void *extra_state)
35
36 int MPI_Register_datarep_c(const char *datarep,
37     MPI_Datarep_conversion_function_c *read_conversion_fn,
38     MPI_Datarep_conversion_function_c *write_conversion_fn,
39     MPI_Datarep_extent_function *dtype_file_extent_fn,
40     void *extra_state)
```

A.3.13 Language Bindings C Bindings

```
39 MPI_Fint MPI_Comm_c2f(MPI_Comm comm)
40
41 MPI_Comm MPI_Comm_f2c(MPI_Fint comm)
42
43 MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)
44 MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)
45
46 MPI_Fint MPI_File_c2f(MPI_File file)
47 MPI_File MPI_File_f2c(MPI_Fint file)
```

48

MPI_Fint MPI_Group_c2f(MPI_Group group)	1
MPI_Group MPI_Group_f2c(MPI_Fint group)	2
MPI_Fint MPI_Info_c2f(MPI_Info info)	3
MPI_Info MPI_Info_f2c(MPI_Fint info)	4
MPI_Fint MPI_Message_c2f(MPI_Message message)	5
MPI_Message MPI_Message_f2c(MPI_Fint message)	6
MPI_Fint MPI_Op_c2f(MPI_Op op)	7
MPI_Op MPI_Op_f2c(MPI_Fint op)	8
MPI_Fint MPI_Request_c2f(MPI_Request request)	9
MPI_Request MPI_Request_f2c(MPI_Fint request)	10
MPI_Fint MPI_Session_c2f(MPI_Session session)	11
MPI_Session MPI_Session_f2c(MPI_Fint session)	12
int MPI_Status_c2f(const MPI_Status *c_status, MPI_Fint *f_status)	13
int MPI_Status_c2f08(const MPI_Status *c_status, MPI_F08_status *f08_status)	14
int MPI_Status_f082c(const MPI_F08_status *f08_status, MPI_Status *c_status)	15
int MPI_Status_f082f(const MPI_F08_status *f08_status, MPI_Fint *f_status)	16
int MPI_Status_f2c(const MPI_Fint *f_status, MPI_Status *c_status)	17
int MPI_Status_f2f08(const MPI_Fint *f_status, MPI_F08_status *f08_status)	18
MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)	19
int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype)	20
int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)	21
int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)	22
MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)	23
int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *datatype)	24
MPI_Fint MPI_Win_c2f(MPI_Win win)	25
MPI_Win MPI_Win_f2c(MPI_Fint win)	26
	27
A.3.14 Tools / Profiling Interface C Bindings	28
int MPI_Pcontrol(const int level, . . .)	29
	30
A.3.15 Tools / MPI Tool Information Interface C Bindings	31
int MPI_T_category_changed(int *update_number)	32
	33
	34
	35
	36
	37
	38
	39
	40
	41
	42
	43
	44
	45
	46
	47
	48

```
1 int MPI_T_category_get_categories(int cat_index, int len, int indices[])
2
3 int MPI_T_category_get_cvars(int cat_index, int len, int indices[])
4
5 int MPI_T_category_get_events(int cat_index, int len, int indices[])
6
7 int MPI_T_category_get_index(const char *name, int *cat_index)
8
9 int MPI_T_category_get_info(int cat_index, char *name, int *name_len,
10 char *desc, int *desc_len, int *num_cvars, int *num_pvars,
11 int *num_categories)
12
13 int MPI_T_category_get_num(int *num_cat)
14
15 int MPI_T_category_get_num_events(int cat_index, int *num_events)
16
17 int MPI_T_category_get_pvars(int cat_index, int len, int indices[])
18
19 int MPI_T_cvar_get_index(const char *name, int *cvar_index)
20
21 int MPI_T_cvar_get_info(int cvar_index, char *name, int *name_len,
22 int *verbosity, MPI_Datatype *datatype, MPI_T_enum *enumtype,
23 char *desc, int *desc_len, int *bind, int *scope)
24
25 int MPI_T_cvar_get_num(int *num_cvar)
26
27 int MPI_T_cvar_handle_alloc(int cvar_index, void *obj_handle,
28 MPI_T_cvar_handle *handle, int *count)
29
30 int MPI_T_cvar_handle_free(MPI_T_cvar_handle *handle)
31
32 int MPI_T_cvar_read(MPI_T_cvar_handle handle, void *buf)
33
34 int MPI_T_cvar_write(MPI_T_cvar_handle handle, const void *buf)
35
36 int MPI_T_enum_get_info(MPI_T_enum enumtype, int *num, char *name,
37 int *name_len)
38
39 int MPI_T_enum_get_item(MPI_T_enum enumtype, int index, int *value, char *name,
40 int *name_len)
41
42 int MPI_T_event_callback_get_info(MPI_T_event_registration event_registration,
43 MPI_T_cb_safety cb_safety, MPI_Info *info_used)
44
45 int MPI_T_event_callback_set_info(MPI_T_event_registration event_registration,
46 MPI_T_cb_safety cb_safety, MPI_Info info)
47
48 int MPI_T_event_copy(MPI_T_event_instance event_instance, void *buffer)
49
50 int MPI_T_event_get_index(const char *name, int *event_index)
51
52 int MPI_T_event_get_info(int event_index, char *name, int *name_len,
53 int *verbosity, MPI_Datatype array_of_datatypes[],
54 MPI_Aint array_of_displacements[], int *num_elements,
55 MPI_T_enum *enumtype, MPI_Info *info, char *desc, int *desc_len,
56 int *bind)
57
58 int MPI_T_event_get_num(int *num_events)
```

```

int MPI_T_event_get_source(MPI_T_event_instance event_instance,      1
                          int *source_index)                        2
                                                                    3
int MPI_T_event_get_timestamp(MPI_T_event_instance event_instance,  4
                              MPI_Count *event_timestamp)          5
                                                                    6
int MPI_T_event_handle_alloc(int event_index, void *obj_handle, MPI_Info info,
                              MPI_T_event_registration *event_registration) 7
                                                                    8
int MPI_T_event_handle_free(MPI_T_event_registration event_registration,
                             void *user_data, MPI_T_event_free_cb_function free_cb_function) 9
                                                                    10
int MPI_T_event_handle_get_info(MPI_T_event_registration event_registration,
                                MPI_Info *info_used)                11
                                                                    12
int MPI_T_event_handle_set_info(MPI_T_event_registration event_registration,
                                MPI_Info info)                       13
                                                                    14
int MPI_T_event_read(MPI_T_event_instance event_instance, int element_index,
                     void *buffer)                                   15
                                                                    16
int MPI_T_event_register_callback(MPI_T_event_registration event_registration,
                                  MPI_T_cb_safety cb_safety, MPI_Info info, void *user_data,
                                  MPI_T_event_cb_function event_cb_function) 17
                                                                    18
int MPI_T_event_set_dropped_handler(
    MPI_T_event_registration event_registration,
    MPI_T_event_dropped_cb_function dropped_cb_function)            19
                                                                    20
int MPI_T_finalize(void)                                           21
                                                                    22
int MPI_T_init_thread(int required, int *provided)                  23
                                                                    24
int MPI_T_pvar_get_index(const char *name, int var_class, int *pvar_index) 25
                                                                    26
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) 27
                                                                    28
int MPI_T_pvar_get_num(int *num_pvar)                               29
                                                                    30
int MPI_T_pvar_handle_alloc(MPI_T_pvar_session pe_session, int pvar_index,
                             void *obj_handle, MPI_T_pvar_handle *handle, int *count) 31
                                                                    32
int MPI_T_pvar_handle_free(MPI_T_pvar_session pe_session,
                            MPI_T_pvar_handle *handle)              33
                                                                    34
int MPI_T_pvar_read(MPI_T_pvar_session pe_session, MPI_T_pvar_handle handle,
                    void *buf)                                       35
                                                                    36
int MPI_T_pvar_readreset(MPI_T_pvar_session pe_session,
                          MPI_T_pvar_handle handle, void *buf)      37
                                                                    38
int MPI_T_pvar_reset(MPI_T_pvar_session pe_session, MPI_T_pvar_handle handle) 39
                                                                    40
                                                                    41
                                                                    42
                                                                    43
                                                                    44
                                                                    45
                                                                    46
                                                                    47
                                                                    48

```

```
1 int MPI_T_pvar_session_create(MPI_T_pvar_session *pe_session)
2
3 int MPI_T_pvar_session_free(MPI_T_pvar_session *pe_session)
4
5 int MPI_T_pvar_start(MPI_T_pvar_session pe_session, MPI_T_pvar_handle handle)
6
7 int MPI_T_pvar_stop(MPI_T_pvar_session pe_session, MPI_T_pvar_handle handle)
8
9 int MPI_T_pvar_write(MPI_T_pvar_session pe_session, MPI_T_pvar_handle handle,
10                     const void *buf)
11
12 int MPI_T_source_get_info(int source_index, char *name, int *name_len,
13                          char *desc, int *desc_len, MPI_T_source_order *ordering,
14                          MPI_Count *ticks_per_second, MPI_Count *max_ticks,
15                          MPI_Info *info)
16
17 int MPI_T_source_get_num(int *num_sources)
18
19 int MPI_T_source_get_timestamp(int source_index, MPI_Count *timestamp)
```

A.3.16 Deprecated C Bindings

```
20 int MPI_Attr_delete(MPI_Comm comm, int keyval)
21
22 int MPI_Attr_get(MPI_Comm comm, int keyval, void *attribute_val, int *flag)
23
24 int MPI_Attr_put(MPI_Comm comm, int keyval, void *attribute_val)
25
26 int MPI_DUP_FN(MPI_Comm oldcomm, int keyval, void *extra_state,
27               void *attribute_val_in, void *attribute_val_out, int *flag)
28
29 int MPI_Info_get(MPI_Info info, const char *key, int valuelen, char *value,
30                 int *flag)
31
32 int MPI_Info_get_valuelen(MPI_Info info, const char *key, int *valuelen,
33                            int *flag)
34
35 int MPI_Keyval_create(MPI_Copy_function *copy_fn,
36                      MPI_Delete_function *delete_fn, int *keyval, void *extra_state)
37
38 int MPI_Keyval_free(int *keyval)
39
40 int MPI_NULL_COPY_FN(MPI_Comm oldcomm, int keyval, void *extra_state,
41                     void *attribute_val_in, void *attribute_val_out, int *flag)
42
43 int MPI_NULL_DELETE_FN(MPI_Comm comm, int keyval, void *attribute_val,
44                       void *extra_state)
```

A.4 Fortran 2008 Bindings with the mpi_f08 Module

A.4.1 Point-to-Point Communication Fortran 2008 Bindings

```

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(buf, count, datatype, dest, tag, comm, ierror) !(_c)
  TYPE(*), DIMENSION(..), INTENT(IN) :: buf
  INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  INTEGER, INTENT(IN) :: dest, tag
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

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_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror) !(_c)
  TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
  INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  INTEGER, INTENT(IN) :: dest, tag
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Request), INTENT(OUT) :: request
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

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

MPI_Buffer_attach(buffer, size, ierror) !(_c)
  TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer
  INTEGER(KIND=MPI_COUNT_KIND), 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

```

```

1 MPI_Buffer_detach(buffer_addr, size, ierror) !(_c)
2   USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
3   TYPE(C_PTR), INTENT(OUT) :: buffer_addr
4   INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
5   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
7 MPI_Cancel(request, ierror)
8   TYPE(MPI_Request), INTENT(IN) :: request
9   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11 MPI_Get_count(status, datatype, count, ierror)
12   TYPE(MPI_Status), INTENT(IN) :: status
13   TYPE(MPI_Datatype), INTENT(IN) :: datatype
14   INTEGER, INTENT(OUT) :: count
15   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17 MPI_Get_count(status, datatype, count, ierror) !(_c)
18   TYPE(MPI_Status), INTENT(IN) :: status
19   TYPE(MPI_Datatype), INTENT(IN) :: datatype
20   INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: count
21   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23 MPI_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror)
24   TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
25   INTEGER, INTENT(IN) :: count, dest, tag
26   TYPE(MPI_Datatype), INTENT(IN) :: datatype
27   TYPE(MPI_Comm), INTENT(IN) :: comm
28   TYPE(MPI_Request), INTENT(OUT) :: request
29   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31 MPI_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror) !(_c)
32   TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
33   INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
34   TYPE(MPI_Datatype), INTENT(IN) :: datatype
35   INTEGER, INTENT(IN) :: dest, tag
36   TYPE(MPI_Comm), INTENT(IN) :: comm
37   TYPE(MPI_Request), INTENT(OUT) :: request
38   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40 MPI_Improbe(source, tag, comm, flag, message, status, ierror)
41   INTEGER, INTENT(IN) :: source, tag
42   TYPE(MPI_Comm), INTENT(IN) :: comm
43   LOGICAL, INTENT(OUT) :: flag
44   TYPE(MPI_Message), INTENT(OUT) :: message
45   TYPE(MPI_Status) :: status
46   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48 MPI_Imrecv(buf, count, datatype, message, request, ierror)
49   TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
50   INTEGER, INTENT(IN) :: count
51   TYPE(MPI_Datatype), INTENT(IN) :: datatype

```



```

TYPE(MPI_Message), INTENT(INOUT) :: message           1
TYPE(MPI_Request), INTENT(OUT) :: request             2
INTEGER, OPTIONAL, INTENT(OUT) :: ierror             3
                                                    4
MPI_Imrecv(buf, count, datatype, message, request, ierror) !(_c) 5
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf         6
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count    7
TYPE(MPI_Datatype), INTENT(IN) :: datatype          8
TYPE(MPI_Message), INTENT(INOUT) :: message         9
TYPE(MPI_Request), INTENT(OUT) :: request          10
INTEGER, OPTIONAL, INTENT(OUT) :: ierror           11
                                                    12
MPI_Iprobe(source, tag, comm, flag, status, ierror)    13
INTEGER, INTENT(IN) :: source, tag                 14
TYPE(MPI_Comm), INTENT(IN) :: comm                 15
LOGICAL, INTENT(OUT) :: flag                       16
TYPE(MPI_Status) :: status                         17
INTEGER, OPTIONAL, INTENT(OUT) :: ierror           18
                                                    19
MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror) 20
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf       21
INTEGER, INTENT(IN) :: count, source, tag          22
TYPE(MPI_Datatype), INTENT(IN) :: datatype        23
TYPE(MPI_Comm), INTENT(IN) :: comm                24
TYPE(MPI_Request), INTENT(OUT) :: request         25
INTEGER, OPTIONAL, INTENT(OUT) :: ierror          26
                                                    27
MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror) !(_c) 28
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf       29
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count  30
TYPE(MPI_Datatype), INTENT(IN) :: datatype        31
INTEGER, INTENT(IN) :: source, tag                 32
TYPE(MPI_Comm), INTENT(IN) :: comm                33
TYPE(MPI_Request), INTENT(OUT) :: request         34
INTEGER, OPTIONAL, INTENT(OUT) :: ierror          35
                                                    36
MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror) 37
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 38
INTEGER, INTENT(IN) :: count, dest, tag           39
TYPE(MPI_Datatype), INTENT(IN) :: datatype        40
TYPE(MPI_Comm), INTENT(IN) :: comm                41
TYPE(MPI_Request), INTENT(OUT) :: request         42
INTEGER, OPTIONAL, INTENT(OUT) :: ierror          43
                                                    44
MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror) !(_c) 45
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 46
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count  47
TYPE(MPI_Datatype), INTENT(IN) :: datatype        48
INTEGER, INTENT(IN) :: dest, tag
TYPE(MPI_Comm), INTENT(IN) :: comm

```

```

1      TYPE(MPI_Request), INTENT(OUT) :: request
2      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
4      MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror)
5          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
6          INTEGER, INTENT(IN) :: count, dest, tag
7          TYPE(MPI_Datatype), INTENT(IN) :: datatype
8          TYPE(MPI_Comm), INTENT(IN) :: comm
9          TYPE(MPI_Request), INTENT(OUT) :: request
10         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12      MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror) !(_c)
13         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
14         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
15         TYPE(MPI_Datatype), INTENT(IN) :: datatype
16         INTEGER, INTENT(IN) :: dest, tag
17         TYPE(MPI_Comm), INTENT(IN) :: comm
18         TYPE(MPI_Request), INTENT(OUT) :: request
19         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21      MPI_Isendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount,
22                    recvtype, source, recvtag, comm, request, ierror)
23         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
24         INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source, recvtag
25         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
26         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
27         TYPE(MPI_Comm), INTENT(IN) :: comm
28         TYPE(MPI_Request), INTENT(OUT) :: request
29         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31      MPI_Isendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount,
32                    recvtype, source, recvtag, comm, request, ierror) !(_c)
33         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
34         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
35         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
36         INTEGER, INTENT(IN) :: dest, sendtag, source, recvtag
37         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
38         TYPE(MPI_Comm), INTENT(IN) :: comm
39         TYPE(MPI_Request), INTENT(OUT) :: request
40         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42      MPI_Isendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
43                            comm, request, ierror)
44         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
45         INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag
46         TYPE(MPI_Datatype), INTENT(IN) :: datatype
47         TYPE(MPI_Comm), INTENT(IN) :: comm
48         TYPE(MPI_Request), INTENT(OUT) :: request
49         INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

MPI_Isendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
                      comm, request, ierror) !(_c)
  TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
  INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  INTEGER, INTENT(IN) :: dest, sendtag, source, recvtag
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Request), INTENT(OUT) :: request
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

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_Issend(buf, count, datatype, dest, tag, comm, request, ierror) !(_c)
  TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
  INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  INTEGER, INTENT(IN) :: dest, tag
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Request), INTENT(OUT) :: request
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

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_Mrecv(buf, count, datatype, message, status, ierror) !(_c)
  TYPE(*), DIMENSION(..) :: buf
  INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Message), INTENT(INOUT) :: message
  TYPE(MPI_Status) :: status
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```
1 MPI_Probe(source, tag, comm, status, ierror)
2     INTEGER, INTENT(IN) :: source, tag
3     TYPE(MPI_Comm), INTENT(IN) :: comm
4     TYPE(MPI_Status) :: status
5     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
7 MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)
8     TYPE(*), DIMENSION(..) :: buf
9     INTEGER, INTENT(IN) :: count, source, tag
10    TYPE(MPI_Datatype), INTENT(IN) :: datatype
11    TYPE(MPI_Comm), INTENT(IN) :: comm
12    TYPE(MPI_Status) :: status
13    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15 MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror) !(_c)
16    TYPE(*), DIMENSION(..) :: buf
17    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
18    TYPE(MPI_Datatype), INTENT(IN) :: datatype
19    INTEGER, INTENT(IN) :: source, tag
20    TYPE(MPI_Comm), INTENT(IN) :: comm
21    TYPE(MPI_Status) :: status
22    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
24 MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror)
25    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
26    INTEGER, INTENT(IN) :: count, source, tag
27    TYPE(MPI_Datatype), INTENT(IN) :: datatype
28    TYPE(MPI_Comm), INTENT(IN) :: comm
29    TYPE(MPI_Request), INTENT(OUT) :: request
30    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32 MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror) !(_c)
33    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
34    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
35    TYPE(MPI_Datatype), INTENT(IN) :: datatype
36    INTEGER, INTENT(IN) :: source, tag
37    TYPE(MPI_Comm), INTENT(IN) :: comm
38    TYPE(MPI_Request), INTENT(OUT) :: request
39    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41 MPI_Request_free(request, ierror)
42    TYPE(MPI_Request), INTENT(INOUT) :: request
43    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45 MPI_Request_get_status(request, flag, status, ierror)
46    TYPE(MPI_Request), INTENT(IN) :: request
47    LOGICAL, INTENT(OUT) :: flag
48    TYPE(MPI_Status) :: status
49    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
50
51 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(buf, count, datatype, dest, tag, comm, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN) :: buf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
INTEGER, INTENT(IN) :: dest, tag
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

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_Rsend_init(buf, count, datatype, dest, tag, comm, request, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
INTEGER, INTENT(IN) :: dest, tag
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

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

MPI_Send(buf, count, datatype, dest, tag, comm, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN) :: buf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
INTEGER, INTENT(IN) :: dest, tag
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

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

```

```

1      TYPE(MPI_Comm), INTENT(IN) :: comm
2      TYPE(MPI_Request), INTENT(OUT) :: request
3      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
5      MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror) !(_c)
6      TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
7      INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
8      TYPE(MPI_Datatype), INTENT(IN) :: datatype
9      INTEGER, INTENT(IN) :: dest, tag
10     TYPE(MPI_Comm), INTENT(IN) :: comm
11     TYPE(MPI_Request), INTENT(OUT) :: request
12     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14     MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount,
15                 recvtype, source, recvtag, comm, status, ierror)
16     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
17     INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source, recvtag
18     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
19     TYPE(*), DIMENSION(..) :: recvbuf
20     TYPE(MPI_Comm), INTENT(IN) :: comm
21     TYPE(MPI_Status) :: status
22     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
24     MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount,
25                 recvtype, source, recvtag, comm, status, ierror) !(_c)
26     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
27     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
28     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
29     INTEGER, INTENT(IN) :: dest, sendtag, source, recvtag
30     TYPE(*), DIMENSION(..) :: recvbuf
31     TYPE(MPI_Comm), INTENT(IN) :: comm
32     TYPE(MPI_Status) :: status
33     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35     MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
36                          comm, status, ierror)
37     TYPE(*), DIMENSION(..) :: buf
38     INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag
39     TYPE(MPI_Datatype), INTENT(IN) :: datatype
40     TYPE(MPI_Comm), INTENT(IN) :: comm
41     TYPE(MPI_Status) :: status
42     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44     MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
45                          comm, status, ierror) !(_c)
46     TYPE(*), DIMENSION(..) :: buf
47     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
48     TYPE(MPI_Datatype), INTENT(IN) :: datatype
49     INTEGER, INTENT(IN) :: dest, sendtag, source, recvtag
50     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(buf, count, datatype, dest, tag, comm, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN) :: buf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
INTEGER, INTENT(IN) :: dest, tag
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
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_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
INTEGER, INTENT(IN) :: dest, tag
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
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)

```

```
1     TYPE(MPI_Status), INTENT(IN) :: status
2     LOGICAL, INTENT(OUT) :: flag
3     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
5     MPI_Testall(count, array_of_requests, flag, array_of_statuses, ierror)
6         INTEGER, INTENT(IN) :: count
7         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
8         LOGICAL, INTENT(OUT) :: flag
9         TYPE(MPI_Status) :: array_of_statuses(*)
10        INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12    MPI_Testany(count, array_of_requests, index, flag, status, ierror)
13        INTEGER, INTENT(IN) :: count
14        TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
15        INTEGER, INTENT(OUT) :: index
16        LOGICAL, INTENT(OUT) :: flag
17        TYPE(MPI_Status) :: status
18        INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20    MPI_Testsome(incount, array_of_requests, outcount, array_of_indices,
21                array_of_statuses, ierror)
22        INTEGER, INTENT(IN) :: incount
23        TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)
24        INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
25        TYPE(MPI_Status) :: array_of_statuses(*)
26        INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28    MPI_Wait(request, status, ierror)
29        TYPE(MPI_Request), INTENT(INOUT) :: request
30        TYPE(MPI_Status) :: status
31        INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33    MPI_Waitall(count, array_of_requests, array_of_statuses, ierror)
34        INTEGER, INTENT(IN) :: count
35        TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
36        TYPE(MPI_Status) :: array_of_statuses(*)
37        INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39    MPI_Waitany(count, array_of_requests, index, status, ierror)
40        INTEGER, INTENT(IN) :: count
41        TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
42        INTEGER, INTENT(OUT) :: index
43        TYPE(MPI_Status) :: status
44        INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46    MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices,
47                array_of_statuses, ierror)
48        INTEGER, INTENT(IN) :: incount
49        TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)
50        INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
51        TYPE(MPI_Status) :: array_of_statuses(*)
```



```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror
1
2
3
A.4.2 Partitioned Communication Fortran 2008 Bindings
4
MPI_Parrived(request, partition, flag, ierror)
5
TYPE(MPI_Request), INTENT(IN) :: request
6
INTEGER, INTENT(IN) :: partition
7
LOGICAL, INTENT(OUT) :: flag
8
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10
MPI_Pready(partition, request, ierror)
11
INTEGER, INTENT(IN) :: partition
12
TYPE(MPI_Request), INTENT(IN) :: request
13
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
MPI_Pready_list(length, array_of_partitions, request, ierror)
16
INTEGER, INTENT(IN) :: length, array_of_partitions(length)
17
TYPE(MPI_Request), INTENT(IN) :: request
18
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20
MPI_Pready_range(partition_low, partition_high, request, ierror)
21
INTEGER, INTENT(IN) :: partition_low, partition_high
22
TYPE(MPI_Request), INTENT(IN) :: request
23
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
25
MPI_Precv_init(buf, partitions, count, datatype, source, tag, comm, info,
26
               request, ierror)
27
TYPE(*), DIMENSION(..), INTENT(IN) :: buf
28
INTEGER, INTENT(IN) :: partitions, source, tag
29
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
30
TYPE(MPI_Datatype), INTENT(IN) :: datatype
31
TYPE(MPI_Comm), INTENT(IN) :: comm
32
TYPE(MPI_Info), INTENT(IN) :: info
33
TYPE(MPI_Request), INTENT(OUT) :: request
34
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
MPI_Psend_init(buf, partitions, count, datatype, dest, tag, comm, info,
37
               request, ierror)
38
TYPE(*), DIMENSION(..), INTENT(IN) :: buf
39
INTEGER, INTENT(IN) :: partitions, dest, tag
40
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
41
TYPE(MPI_Datatype), INTENT(IN) :: datatype
42
TYPE(MPI_Comm), INTENT(IN) :: comm
43
TYPE(MPI_Info), INTENT(IN) :: info
44
TYPE(MPI_Request), INTENT(OUT) :: request
45
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
48

```

A.4.3 Datatypes Fortran 2008 Bindings

```

1  INTEGER(KIND=MPI_ADDRESS_KIND) MPI_Aint_add(base, disp)
2      INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: base, disp
3
4  INTEGER(KIND=MPI_ADDRESS_KIND) MPI_Aint_diff(addr1, addr2)
5      INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: addr1, addr2
6
7  MPI_Get_address(location, address, ierror)
8      TYPE(*), DIMENSION(..), ASYNCHRONOUS :: location
9      INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: address
10     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12 MPI_Get_elements(status, datatype, count, ierror)
13     TYPE(MPI_Status), INTENT(IN) :: status
14     TYPE(MPI_Datatype), INTENT(IN) :: datatype
15     INTEGER, INTENT(OUT) :: count
16     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18 MPI_Get_elements(status, datatype, count, ierror) !(_c)
19     TYPE(MPI_Status), INTENT(IN) :: status
20     TYPE(MPI_Datatype), INTENT(IN) :: datatype
21     INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: count
22     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
24 MPI_Get_elements_x(status, datatype, count, ierror)
25     TYPE(MPI_Status), INTENT(IN) :: status
26     TYPE(MPI_Datatype), INTENT(IN) :: datatype
27     INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: count
28     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30 MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror)
31     TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
32     INTEGER, INTENT(IN) :: incount, outsize
33     TYPE(MPI_Datatype), INTENT(IN) :: datatype
34     TYPE(*), DIMENSION(..) :: outbuf
35     INTEGER, INTENT(INOUT) :: position
36     TYPE(MPI_Comm), INTENT(IN) :: comm
37     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39 MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror)
40     !(_c)
41     TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
42     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: incount, outsize
43     TYPE(MPI_Datatype), INTENT(IN) :: datatype
44     TYPE(*), DIMENSION(..) :: outbuf
45     INTEGER(KIND=MPI_COUNT_KIND), INTENT(INOUT) :: position
46     TYPE(MPI_Comm), INTENT(IN) :: comm
47     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
49 MPI_Pack_external(datatype, inbuf, incount, datatype, outbuf, outsize, position,
50     ierror)

```

```

CHARACTER(LEN=*), INTENT(IN) :: datarep 1
TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf 2
INTEGER, INTENT(IN) :: incount 3
TYPE(MPI_Datatype), INTENT(IN) :: datatype 4
TYPE(*), DIMENSION(..) :: outbuf 5
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: outsize 6
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position 7
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 8
MPI_Pack_external(datarep, inbuf, incount, datatype, outbuf, outsize, position, 9
ierror) !(_c) 10
CHARACTER(LEN=*), INTENT(IN) :: datarep 11
TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf 12
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: incount, outsize 13
TYPE(MPI_Datatype), INTENT(IN) :: datatype 14
TYPE(*), DIMENSION(..) :: outbuf 15
INTEGER(KIND=MPI_COUNT_KIND), INTENT(INOUT) :: position 16
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 17
MPI_Pack_external_size(datarep, incount, datatype, size, ierror) 18
CHARACTER(LEN=*), INTENT(IN) :: datarep 19
INTEGER, INTENT(IN) :: incount 20
TYPE(MPI_Datatype), INTENT(IN) :: datatype 21
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size 22
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 23
MPI_Pack_external_size(datarep, incount, datatype, size, ierror) !(_c) 24
CHARACTER(LEN=*), INTENT(IN) :: datarep 25
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: incount 26
TYPE(MPI_Datatype), INTENT(IN) :: datatype 27
INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size 28
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 29
MPI_Pack_size(incount, datatype, comm, size, ierror) 30
INTEGER, INTENT(IN) :: incount 31
TYPE(MPI_Datatype), INTENT(IN) :: datatype 32
TYPE(MPI_Comm), INTENT(IN) :: comm 33
INTEGER, INTENT(OUT) :: size 34
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 35
MPI_Pack_size(incount, datatype, comm, size, ierror) !(_c) 36
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: incount 37
TYPE(MPI_Datatype), INTENT(IN) :: datatype 38
TYPE(MPI_Comm), INTENT(IN) :: comm 39
INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size 40
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 41
MPI_Type_commit(datatype, ierror) 42
TYPE(MPI_Datatype), INTENT(INOUT) :: datatype 43
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 44

```

```

1 MPI_Type_contiguous(count, oldtype, newtype, ierror)
2     INTEGER, INTENT(IN) :: count
3     TYPE(MPI_Datatype), INTENT(IN) :: oldtype
4     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
5     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
7 MPI_Type_contiguous(count, oldtype, newtype, ierror) !(_c)
8     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
9     TYPE(MPI_Datatype), INTENT(IN) :: oldtype
10    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
11    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13 MPI_Type_create_darray(size, rank, ndims, array_of_gsizes, array_of_distrib,
14     array_of_dargs, array_of_psize, order, oldtype, newtype, ierror)
15     INTEGER, INTENT(IN) :: size, rank, ndims, array_of_gsize(ndims),
16     array_of_distrib(ndims), array_of_dargs(ndims),
17     array_of_psize(ndims), order
18     TYPE(MPI_Datatype), INTENT(IN) :: oldtype
19     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
20     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22 MPI_Type_create_darray(size, rank, ndims, array_of_gsize, array_of_distrib,
23     array_of_dargs, array_of_psize, order, oldtype, newtype, ierror)
24     !(_c)
25     INTEGER, INTENT(IN) :: size, rank, ndims, array_of_distrib(ndims),
26     array_of_dargs(ndims), array_of_psize(ndims), order
27     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: array_of_gsize(ndims)
28     TYPE(MPI_Datatype), INTENT(IN) :: oldtype
29     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
30     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32 MPI_Type_create_hindexed(count, array_of_blocklength, array_of_displacement,
33     oldtype, newtype, ierror)
34     INTEGER, INTENT(IN) :: count, array_of_blocklength(count)
35     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: array_of_displacement(count)
36     TYPE(MPI_Datatype), INTENT(IN) :: oldtype
37     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
38     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40 MPI_Type_create_hindexed(count, array_of_blocklength, array_of_displacement,
41     oldtype, newtype, ierror) !(_c)
42     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count,
43     array_of_blocklength(count), array_of_displacement(count)
44     TYPE(MPI_Datatype), INTENT(IN) :: oldtype
45     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
46     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48 MPI_Type_create_hindexed_block(count, blocklength, array_of_displacement,
49     oldtype, newtype, ierror)
50     INTEGER, INTENT(IN) :: count, blocklength

```

```

INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: array_of_displacements(count) 1
TYPE(MPI_Datatype), INTENT(IN) :: oldtype 2
TYPE(MPI_Datatype), INTENT(OUT) :: newtype 3
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4
5
MPI_Type_create_hindexed_block(count, blocklength, array_of_displacements, 6
    oldtype, newtype, ierror) !(_c) 7
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count, blocklength, 8
    array_of_displacements(count) 9
TYPE(MPI_Datatype), INTENT(IN) :: oldtype 10
TYPE(MPI_Datatype), INTENT(OUT) :: newtype 11
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 12
13
MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype, ierror) 14
INTEGER, INTENT(IN) :: count, blocklength 15
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: stride 16
TYPE(MPI_Datatype), INTENT(IN) :: oldtype 17
TYPE(MPI_Datatype), INTENT(OUT) :: newtype 18
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 19
20
MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype, ierror) 21
    !(_c) 22
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count, blocklength, stride 23
TYPE(MPI_Datatype), INTENT(IN) :: oldtype 24
TYPE(MPI_Datatype), INTENT(OUT) :: newtype 25
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 26
27
MPI_Type_create_indexed_block(count, blocklength, array_of_displacements, 28
    oldtype, newtype, ierror) 29
INTEGER, INTENT(IN) :: count, blocklength, array_of_displacements(count) 30
TYPE(MPI_Datatype), INTENT(IN) :: oldtype 31
TYPE(MPI_Datatype), INTENT(OUT) :: newtype 32
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 33
34
MPI_Type_create_indexed_block(count, blocklength, array_of_displacements, 35
    oldtype, newtype, ierror) !(_c) 36
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count, blocklength, 37
    array_of_displacements(count) 38
TYPE(MPI_Datatype), INTENT(IN) :: oldtype 39
TYPE(MPI_Datatype), INTENT(OUT) :: newtype 40
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 41
42
MPI_Type_create_resized(oldtype, lb, extent, newtype, ierror) 43
TYPE(MPI_Datatype), INTENT(IN) :: oldtype 44
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: lb, extent 45
TYPE(MPI_Datatype), INTENT(OUT) :: newtype 46
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 47
48
MPI_Type_create_resized(oldtype, lb, extent, newtype, ierror) !(_c) 49
TYPE(MPI_Datatype), INTENT(IN) :: oldtype 50
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: lb, extent 51

```

```

1     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
2     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
4 MPI_Type_create_struct(count, array_of_blocklengths, array_of_displacements,
5                       array_of_types, newtype, ierror)
6     INTEGER, INTENT(IN) :: count, array_of_blocklengths(count)
7     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: array_of_displacements(count)
8     TYPE(MPI_Datatype), INTENT(IN) :: array_of_types(count)
9     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
10    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12 MPI_Type_create_struct(count, array_of_blocklengths, array_of_displacements,
13                       array_of_types, newtype, ierror) !(_c)
14    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count,
15    array_of_blocklengths(count), array_of_displacements(count)
16    TYPE(MPI_Datatype), INTENT(IN) :: array_of_types(count)
17    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
18    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20 MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,
21                          array_of_starts, order, oldtype, newtype, ierror)
22    INTEGER, INTENT(IN) :: ndims, array_of_sizes(ndims),
23    array_of_subsizes(ndims), array_of_starts(ndims), order
24    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
25    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
26    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28 MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,
29                          array_of_starts, order, oldtype, newtype, ierror) !(_c)
30    INTEGER, INTENT(IN) :: ndims, order
31    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: array_of_sizes(ndims),
32    array_of_subsizes(ndims), array_of_starts(ndims)
33    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
34    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
35    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37 MPI_Type_dup(oldtype, newtype, ierror)
38    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
39    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
40    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42 MPI_Type_free(datatype, ierror)
43    TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
44    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46 MPI_Type_get_contents(datatype, max_integers, max_addresses, max_datatypes,
47                      array_of_integers, array_of_addresses, array_of_datatypes,
48                      ierror)
49    TYPE(MPI_Datatype), INTENT(IN) :: datatype
50    INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes
51    INTEGER, INTENT(OUT) :: array_of_integers(max_integers)

```

```

INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::                                1
    array_of_addresses(max_addresses)                                        2
TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes)        3
INTEGER, OPTIONAL, INTENT(OUT) :: ierror                                    4
                                                                 5
MPI_Type_get_contents(datatype, max_integers, max_addresses, max_large_counts, 6
    max_datatypes, array_of_integers, array_of_addresses,
    array_of_large_counts, array_of_datatypes, ierror) !(_c)                7
TYPE(MPI_Datatype), INTENT(IN) :: datatype                                  8
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: max_integers, max_addresses,    9
    max_large_counts, max_datatypes                                       10
INTEGER, INTENT(OUT) :: array_of_integers(max_integers)                   11
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::                               12
    array_of_addresses(max_addresses)                                       13
INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) ::                               14
    array_of_large_counts(max_large_counts)                                  15
TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes)      16
INTEGER, OPTIONAL, INTENT(OUT) :: ierror                                    17
                                                                 18
MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes, 19
    combiner, ierror)                                                       20
TYPE(MPI_Datatype), INTENT(IN) :: datatype                                  21
INTEGER, INTENT(OUT) :: num_integers, num_addresses, num_datatypes,        22
    combiner                                                                  23
INTEGER, OPTIONAL, INTENT(OUT) :: ierror                                    24
                                                                 25
MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_large_counts, 26
    num_datatypes, combiner, ierror) !(_c)                                   27
TYPE(MPI_Datatype), INTENT(IN) :: datatype                                  28
INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: num_integers, num_addresses,   29
    num_large_counts, num_datatypes                                          30
INTEGER, INTENT(OUT) :: combiner                                             31
INTEGER, OPTIONAL, INTENT(OUT) :: ierror                                    32
                                                                 33
MPI_Type_get_extent(datatype, lb, extent, ierror)                           34
TYPE(MPI_Datatype), INTENT(IN) :: datatype                                  35
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: lb, extent                   36
INTEGER, OPTIONAL, INTENT(OUT) :: ierror                                    37
                                                                 38
MPI_Type_get_extent(datatype, lb, extent, ierror) !(_c)                     39
TYPE(MPI_Datatype), INTENT(IN) :: datatype                                  40
INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: lb, extent                     41
INTEGER, OPTIONAL, INTENT(OUT) :: ierror                                    42
                                                                 43
MPI_Type_get_extent_x(datatype, lb, extent, ierror)                          44
TYPE(MPI_Datatype), INTENT(IN) :: datatype                                  45
INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: lb, extent                     46
INTEGER, OPTIONAL, INTENT(OUT) :: ierror                                    47
                                                                 48
MPI_Type_get_true_extent(datatype, true_lb, true_extent, ierror)
TYPE(MPI_Datatype), INTENT(IN) :: datatype

```

```
1     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: true_lb, true_extent
2     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
4 MPI_Type_get_true_extent(datatype, true_lb, true_extent, ierror) !(_c)
5     TYPE(MPI_Datatype), INTENT(IN) :: datatype
6     INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: true_lb, true_extent
7     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
9 MPI_Type_get_true_extent_x(datatype, true_lb, true_extent, ierror)
10    TYPE(MPI_Datatype), INTENT(IN) :: datatype
11    INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: true_lb, true_extent
12    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14 MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements, oldtype,
15                 newtype, ierror)
16    INTEGER, INTENT(IN) :: count, array_of_blocklengths(count),
17    array_of_displacements(count)
18    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
19    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
20    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22 MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements, oldtype,
23                 newtype, ierror) !(_c)
24    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count,
25    array_of_blocklengths(count), array_of_displacements(count)
26    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
27    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
28    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30 MPI_Type_size(datatype, size, ierror)
31    TYPE(MPI_Datatype), INTENT(IN) :: datatype
32    INTEGER, INTENT(OUT) :: size
33    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35 MPI_Type_size(datatype, size, ierror) !(_c)
36    TYPE(MPI_Datatype), INTENT(IN) :: datatype
37    INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
38    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40 MPI_Type_size_x(datatype, size, ierror)
41    TYPE(MPI_Datatype), INTENT(IN) :: datatype
42    INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
43    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45 MPI_Type_vector(count, blocklength, stride, oldtype, newtype, ierror)
46    INTEGER, INTENT(IN) :: count, blocklength, stride
47    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
48    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
49    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
50
51 MPI_Type_vector(count, blocklength, stride, oldtype, newtype, ierror) !(_c)
52    INTEGER(KIND=MPI_COUNT_KIND), 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(inbuf, insize, position, outbuf, outcount, datatype, comm, ierror)
!(_c)
TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: insize, outcount
INTEGER(KIND=MPI_COUNT_KIND), 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, outcount,
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) :: outcount
TYPE(MPI_Datatype), INTENT(IN) :: datatype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outcount,
datatype, ierror) !(_c)
CHARACTER(LEN=*), INTENT(IN) :: datarep
TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: insize, outcount
INTEGER(KIND=MPI_COUNT_KIND), INTENT(INOUT) :: position
TYPE(*), DIMENSION(..) :: outbuf
TYPE(MPI_Datatype), INTENT(IN) :: datatype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

A.4.4 Collective Communication Fortran 2008 Bindings

MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcnt, recvtype, comm,
ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf

```

```

1     INTEGER, INTENT(IN) :: sendcount, recvcount
2     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
3     TYPE(*), DIMENSION(..) :: recvbuf
4     TYPE(MPI_Comm), INTENT(IN) :: comm
5     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
7     MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm,
8         ierror) !(_c)
9     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
10    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
11    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
12    TYPE(*), DIMENSION(..) :: recvbuf
13    TYPE(MPI_Comm), INTENT(IN) :: comm
14    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16    MPI_Allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
17        comm, info, request, ierror)
18    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
19    INTEGER, INTENT(IN) :: sendcount, recvcount
20    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
21    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
22    TYPE(MPI_Comm), INTENT(IN) :: comm
23    TYPE(MPI_Info), INTENT(IN) :: info
24    TYPE(MPI_Request), INTENT(OUT) :: request
25    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27    MPI_Allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
28        comm, info, request, ierror) !(_c)
29    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
30    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
31    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
32    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
33    TYPE(MPI_Comm), INTENT(IN) :: comm
34    TYPE(MPI_Info), INTENT(IN) :: info
35    TYPE(MPI_Request), INTENT(OUT) :: request
36    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38    MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
39        recvtype, comm, ierror)
40    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
41    INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
42    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
43    TYPE(*), DIMENSION(..) :: recvbuf
44    TYPE(MPI_Comm), INTENT(IN) :: comm
45    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47    MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
48        recvtype, comm, ierror) !(_c)
49    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
50    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcounts(*)

```

```

TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Allgatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcnts, displs,
    recvtype, comm, info, request, ierror)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
INTEGER, INTENT(IN) :: sendcount
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcnts(*), displs(*)
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_init(sendbuf, sendcount, sendtype, recvbuf, recvcnts, displs,
    recvtype, comm, info, request, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcnts(*)
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
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(sendbuf, recvbuf, count, datatype, op, comm, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER(KIND=MPI_COUNT_KIND), 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

```

```

1 MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
2     ierror)
3     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
4     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
5     INTEGER, INTENT(IN) :: count
6     TYPE(MPI_Datatype), INTENT(IN) :: datatype
7     TYPE(MPI_Op), INTENT(IN) :: op
8     TYPE(MPI_Comm), INTENT(IN) :: comm
9     TYPE(MPI_Info), INTENT(IN) :: info
10    TYPE(MPI_Request), INTENT(OUT) :: request
11    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13 MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
14     ierror) !(_c)
15    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
16    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
17    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
18    TYPE(MPI_Datatype), INTENT(IN) :: datatype
19    TYPE(MPI_Op), INTENT(IN) :: op
20    TYPE(MPI_Comm), INTENT(IN) :: comm
21    TYPE(MPI_Info), INTENT(IN) :: info
22    TYPE(MPI_Request), INTENT(OUT) :: request
23    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
25 MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm,
26     ierror)
27    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
28    INTEGER, INTENT(IN) :: sendcount, recvcount
29    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
30    TYPE(*), DIMENSION(..) :: recvbuf
31    TYPE(MPI_Comm), INTENT(IN) :: comm
32    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34 MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm,
35     ierror) !(_c)
36    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
37    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
38    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
39    TYPE(*), DIMENSION(..) :: recvbuf
40    TYPE(MPI_Comm), INTENT(IN) :: comm
41    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43 MPI_Alltoall_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
44     comm, info, request, ierror)
45    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
46    INTEGER, INTENT(IN) :: sendcount, recvcount
47    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
48    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
49    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_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                  comm, info, request, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
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
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_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
              rdispls, recvtype, comm, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*), recvcounts(*)
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..) :: recvbuf
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
INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
recvcounts(*), rdispls(*)
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Info), INTENT(IN) :: info
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Alltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
                  rdispls, recvtype, comm, info, request, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf

```

```

1     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*),
2         recvcounts(*)
3     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
4         rdispls(*)
5     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
6     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
7     TYPE(MPI_Comm), INTENT(IN) :: comm
8     TYPE(MPI_Info), INTENT(IN) :: info
9     TYPE(MPI_Request), INTENT(OUT) :: request
10    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12    MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
13        rdispls, recvtypes, comm, ierror)
14    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
15    INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*), rdispls(*)
16    TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
17    TYPE(*), DIMENSION(..) :: recvbuf
18    TYPE(MPI_Comm), INTENT(IN) :: comm
19    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21    MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
22        rdispls, recvtypes, comm, ierror) !(_c)
23    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
24    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*), recvcounts(*)
25    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
26    TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
27    TYPE(*), DIMENSION(..) :: recvbuf
28    TYPE(MPI_Comm), INTENT(IN) :: comm
29    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31    MPI_Alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
32        recvcounts, rdispls, recvtypes, comm, info, request, ierror)
33    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
34    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
35        recvcounts(*), rdispls(*)
36    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*), recvtypes(*)
37    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
38    TYPE(MPI_Comm), INTENT(IN) :: comm
39    TYPE(MPI_Info), INTENT(IN) :: info
40    TYPE(MPI_Request), INTENT(OUT) :: request
41    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43    MPI_Alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
44        recvcounts, rdispls, recvtypes, comm, info, request, ierror)
45        !(_c)
46    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
47    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*),
48        recvcounts(*)

```

```

INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),      1
    rdispls(*)                                                                2
TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*), recvtypes(*)   3
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf                             4
TYPE(MPI_Comm), INTENT(IN) :: comm                                          5
TYPE(MPI_Info), INTENT(IN) :: info                                          6
TYPE(MPI_Request), INTENT(OUT) :: request                                    7
INTEGER, OPTIONAL, INTENT(OUT) :: ierror                                     8
                                                                              9
MPI_Barrier(comm, ierror)                                                    10
    TYPE(MPI_Comm), INTENT(IN) :: comm                                       11
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror                                  12
                                                                              13
MPI_Barrier_init(comm, info, request, ierror)                                14
    TYPE(MPI_Comm), INTENT(IN) :: comm                                       15
    TYPE(MPI_Info), INTENT(IN) :: info                                       16
    TYPE(MPI_Request), INTENT(OUT) :: request                                  17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror                                  18
                                                                              19
MPI_Bcast(buffer, count, datatype, root, comm, ierror)                       19
    TYPE(*), DIMENSION(..) :: buffer                                         20
    INTEGER, INTENT(IN) :: count, root                                        21
    TYPE(MPI_Datatype), INTENT(IN) :: datatype                               22
    TYPE(MPI_Comm), INTENT(IN) :: comm                                       23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror                                  24
                                                                              25
MPI_Bcast(buffer, count, datatype, root, comm, ierror) !(_c)                25
    TYPE(*), DIMENSION(..) :: buffer                                         26
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count                         27
    TYPE(MPI_Datatype), INTENT(IN) :: datatype                               28
    INTEGER, INTENT(IN) :: root                                              29
    TYPE(MPI_Comm), INTENT(IN) :: comm                                       30
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror                                  31
                                                                              32
MPI_Bcast_init(buffer, count, datatype, root, comm, info, request, ierror)   33
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer                          34
    INTEGER, INTENT(IN) :: count, root                                        35
    TYPE(MPI_Datatype), INTENT(IN) :: datatype                               36
    TYPE(MPI_Comm), INTENT(IN) :: comm                                       37
    TYPE(MPI_Info), INTENT(IN) :: info                                       38
    TYPE(MPI_Request), INTENT(OUT) :: request                                  39
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror                                  40
                                                                              41
MPI_Bcast_init(buffer, count, datatype, root, comm, info, request, ierror)   41
    !(_c)                                                                      42
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer                          43
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count                         44
    TYPE(MPI_Datatype), INTENT(IN) :: datatype                               45
    INTEGER, INTENT(IN) :: root                                              46
    TYPE(MPI_Comm), INTENT(IN) :: comm                                       47
                                                                              48

```

```

1     TYPE(MPI_Info), INTENT(IN) :: info
2     TYPE(MPI_Request), INTENT(OUT) :: request
3     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
5     MPI_Exscan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
6     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
7     TYPE(*), DIMENSION(..) :: recvbuf
8     INTEGER, INTENT(IN) :: count
9     TYPE(MPI_Datatype), INTENT(IN) :: datatype
10    TYPE(MPI_Op), INTENT(IN) :: op
11    TYPE(MPI_Comm), INTENT(IN) :: comm
12    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14    MPI_Exscan(sendbuf, recvbuf, count, datatype, op, comm, ierror) !(_c)
15    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
16    TYPE(*), DIMENSION(..) :: recvbuf
17    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
18    TYPE(MPI_Datatype), INTENT(IN) :: datatype
19    TYPE(MPI_Op), INTENT(IN) :: op
20    TYPE(MPI_Comm), INTENT(IN) :: comm
21    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23    MPI_Exscan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
24                    ierror)
25    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
26    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
27    INTEGER, INTENT(IN) :: count
28    TYPE(MPI_Datatype), INTENT(IN) :: datatype
29    TYPE(MPI_Op), INTENT(IN) :: op
30    TYPE(MPI_Comm), INTENT(IN) :: comm
31    TYPE(MPI_Info), INTENT(IN) :: info
32    TYPE(MPI_Request), INTENT(OUT) :: request
33    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35    MPI_Exscan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
36                    ierror) !(_c)
37    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
38    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
39    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
40    TYPE(MPI_Datatype), INTENT(IN) :: datatype
41    TYPE(MPI_Op), INTENT(IN) :: op
42    TYPE(MPI_Comm), INTENT(IN) :: comm
43    TYPE(MPI_Info), INTENT(IN) :: info
44    TYPE(MPI_Request), INTENT(OUT) :: request
45    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47    MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcnt, recvtype, root,
48              comm, ierror)
49    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
50    INTEGER, INTENT(IN) :: sendcount, recvcnt, root

```



```

TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..) :: recvbuf
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root,
           comm, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: root
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
INTEGER, INTENT(IN) :: sendcount, recvcount, root
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Info), INTENT(IN) :: info
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Gather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
               root, comm, info, request, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER, INTENT(IN) :: root
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_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
INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*), root
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..) :: recvbuf
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
            recvtype, root, comm, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf

```

```

1     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcounts(*)
2     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
3     TYPE(*), DIMENSION(..) :: recvbuf
4     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
5     INTEGER, INTENT(IN) :: root
6     TYPE(MPI_Comm), INTENT(IN) :: comm
7     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
9     MPI_Gatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
10                    recvtype, root, comm, info, request, ierror)
11     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
12     INTEGER, INTENT(IN) :: sendcount, root
13     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
14     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
15     INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
16     TYPE(MPI_Comm), INTENT(IN) :: comm
17     TYPE(MPI_Info), INTENT(IN) :: info
18     TYPE(MPI_Request), INTENT(OUT) :: request
19     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21     MPI_Gatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
22                    recvtype, root, comm, info, request, ierror) !(_c)
23     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
24     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
25     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
26     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
27     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
28     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
29     INTEGER, INTENT(IN) :: root
30     TYPE(MPI_Comm), INTENT(IN) :: comm
31     TYPE(MPI_Info), INTENT(IN) :: info
32     TYPE(MPI_Request), INTENT(OUT) :: request
33     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35     MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
36                  comm, request, ierror)
37     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
38     INTEGER, INTENT(IN) :: sendcount, recvcount
39     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
40     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
41     TYPE(MPI_Comm), INTENT(IN) :: comm
42     TYPE(MPI_Request), INTENT(OUT) :: request
43     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45     MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
46                  comm, request, ierror) !(_c)
47     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
48     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
49     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype

```

```

TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcnts, displs,
                recvtype, comm, request, ierror)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
INTEGER, INTENT(IN) :: sendcount
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcnts(*), displs(*)
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcnts, displs,
                recvtype, comm, request, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcnts(*)
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
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_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
                !(_c)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER(KIND=MPI_COUNT_KIND), 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

```

```

1 MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm,
2             request, ierror)
3     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
4     INTEGER, INTENT(IN) :: sendcount, recvcount
5     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
6     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
7     TYPE(MPI_Comm), INTENT(IN) :: comm
8     TYPE(MPI_Request), INTENT(OUT) :: request
9     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11 MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm,
12             request, ierror) !(_c)
13     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
14     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
15     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
16     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
17     TYPE(MPI_Comm), INTENT(IN) :: comm
18     TYPE(MPI_Request), INTENT(OUT) :: request
19     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21 MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
22             rdispls, recvtype, comm, request, ierror)
23     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
24     INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
25             recvcounts(*), rdispls(*)
26     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
27     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
28     TYPE(MPI_Comm), INTENT(IN) :: comm
29     TYPE(MPI_Request), INTENT(OUT) :: request
30     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32 MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
33             rdispls, recvtype, comm, request, ierror) !(_c)
34     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
35     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*),
36             recvcounts(*)
37     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
38             rdispls(*)
39     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
40     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
41     TYPE(MPI_Comm), INTENT(IN) :: comm
42     TYPE(MPI_Request), INTENT(OUT) :: request
43     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45 MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
46             rdispls, recvtypes, comm, request, ierror)
47     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
48     INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
49             recvcounts(*), rdispls(*)

```

```

TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*), recvtypes(*)
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
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) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*),
               recvcounts(*)
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
               rdispls(*)
TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*), recvtypes(*)
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
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_Ibcast(buffer, count, datatype, root, comm, request, ierror) !(_c)
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
INTEGER, INTENT(IN) :: root
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

```

```

1 MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror) !(_c)
2   TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
3   TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
4   INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
5   TYPE(MPI_Datatype), INTENT(IN) :: datatype
6   TYPE(MPI_Op), INTENT(IN) :: op
7   TYPE(MPI_Comm), INTENT(IN) :: comm
8   TYPE(MPI_Request), INTENT(OUT) :: request
9   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11 MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root,
12             comm, request, ierror)
13   TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
14   INTEGER, INTENT(IN) :: sendcount, recvcount, root
15   TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
16   TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
17   TYPE(MPI_Comm), INTENT(IN) :: comm
18   TYPE(MPI_Request), INTENT(OUT) :: request
19   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21 MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root,
22            comm, request, ierror) !(_c)
23   TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
24   INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
25   TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
26   TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
27   INTEGER, INTENT(IN) :: root
28   TYPE(MPI_Comm), INTENT(IN) :: comm
29   TYPE(MPI_Request), INTENT(OUT) :: request
30   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32 MPI_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
33             recvtype, root, comm, request, ierror)
34   TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
35   INTEGER, INTENT(IN) :: sendcount, root
36   TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
37   TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
38   INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
39   TYPE(MPI_Comm), INTENT(IN) :: comm
40   TYPE(MPI_Request), INTENT(OUT) :: request
41   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43 MPI_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
44             recvtype, root, comm, request, ierror) !(_c)
45   TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
46   INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
47   TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
48   TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
49   INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)

```

```

INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)           1
INTEGER, INTENT(IN) :: root                                                    2
TYPE(MPI_Comm), INTENT(IN) :: comm                                             3
TYPE(MPI_Request), INTENT(OUT) :: request                                      4
INTEGER, OPTIONAL, INTENT(OUT) :: ierror                                       5
                                                                                   6
MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request, ierror) 7
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf                  8
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf                               9
INTEGER, INTENT(IN) :: count, root                                           10
TYPE(MPI_Datatype), INTENT(IN) :: datatype                                    11
TYPE(MPI_Op), INTENT(IN) :: op                                               12
TYPE(MPI_Comm), INTENT(IN) :: comm                                           13
TYPE(MPI_Request), INTENT(OUT) :: request                                     14
INTEGER, OPTIONAL, INTENT(OUT) :: ierror                                       15
                                                                                   16
MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request, ierror) 17
      !(_c)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf                  18
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf                               19
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count                             20
TYPE(MPI_Datatype), INTENT(IN) :: datatype                                    21
TYPE(MPI_Op), INTENT(IN) :: op                                               22
INTEGER, INTENT(IN) :: root                                                  23
TYPE(MPI_Comm), INTENT(IN) :: comm                                           24
TYPE(MPI_Request), INTENT(OUT) :: request                                     25
INTEGER, OPTIONAL, INTENT(OUT) :: ierror                                       26
                                                                                   27
MPI_Ireduce_scatter(sendbuf, recvbuf, recvcnt, datatype, op, comm, request,     28
      ierror)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf                  29
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf                               30
INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcnt(*)                               31
TYPE(MPI_Datatype), INTENT(IN) :: datatype                                    32
TYPE(MPI_Op), INTENT(IN) :: op                                               33
TYPE(MPI_Comm), INTENT(IN) :: comm                                           34
TYPE(MPI_Request), INTENT(OUT) :: request                                     35
INTEGER, OPTIONAL, INTENT(OUT) :: ierror                                       36
                                                                                   37
MPI_Ireduce_scatter(sendbuf, recvbuf, recvcnt, datatype, op, comm, request,     38
      ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf                  39
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf                               40
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcnt(*)         41
TYPE(MPI_Datatype), INTENT(IN) :: datatype                                    42
TYPE(MPI_Op), INTENT(IN) :: op                                               43
TYPE(MPI_Comm), INTENT(IN) :: comm                                           44
TYPE(MPI_Request), INTENT(OUT) :: request                                     45
INTEGER, OPTIONAL, INTENT(OUT) :: ierror                                       46
                                                                                   47
                                                                                   48

```

```

1 MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
2     request, ierror)
3     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
4     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
5     INTEGER, INTENT(IN) :: recvcount
6     TYPE(MPI_Datatype), INTENT(IN) :: datatype
7     TYPE(MPI_Op), INTENT(IN) :: op
8     TYPE(MPI_Comm), INTENT(IN) :: comm
9     TYPE(MPI_Request), INTENT(OUT) :: request
10    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12 MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
13     request, ierror) !(_c)
14    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
15    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
16    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
17    TYPE(MPI_Datatype), INTENT(IN) :: datatype
18    TYPE(MPI_Op), INTENT(IN) :: op
19    TYPE(MPI_Comm), INTENT(IN) :: comm
20    TYPE(MPI_Request), INTENT(OUT) :: request
21    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23 MPI_Iscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
24    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
25    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
26    INTEGER, INTENT(IN) :: count
27    TYPE(MPI_Datatype), INTENT(IN) :: datatype
28    TYPE(MPI_Op), INTENT(IN) :: op
29    TYPE(MPI_Comm), INTENT(IN) :: comm
30    TYPE(MPI_Request), INTENT(OUT) :: request
31    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33 MPI_Iscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror) !(_c)
34    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
35    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
36    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
37    TYPE(MPI_Datatype), INTENT(IN) :: datatype
38    TYPE(MPI_Op), INTENT(IN) :: op
39    TYPE(MPI_Comm), INTENT(IN) :: comm
40    TYPE(MPI_Request), INTENT(OUT) :: request
41    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43 MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root,
44     comm, request, ierror)
45    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
46    INTEGER, INTENT(IN) :: sendcount, recvcount, root
47    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
48    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
49    TYPE(MPI_Comm), INTENT(IN) :: comm

```



```

TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Isscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root,
              comm, request, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER, INTENT(IN) :: root
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Isscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
              recvtype, root, comm, request, ierror)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER, INTENT(IN) :: recvcount, root
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Isscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
              recvtype, root, comm, request, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*)
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
INTEGER, INTENT(IN) :: root
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_create_c(user_fn, commute, op, ierror) !(_c)

```

```

1      PROCEDURE(MPI_User_function_c) :: user_fn
2      LOGICAL, INTENT(IN) :: commute
3      TYPE(MPI_Op), INTENT(OUT) :: op
4      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
6      MPI_Op_free(op, ierror)
7          TYPE(MPI_Op), INTENT(INOUT) :: op
8          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10     MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
11         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
12         TYPE(*), DIMENSION(..) :: recvbuf
13         INTEGER, INTENT(IN) :: count, root
14         TYPE(MPI_Datatype), INTENT(IN) :: datatype
15         TYPE(MPI_Op), INTENT(IN) :: op
16         TYPE(MPI_Comm), INTENT(IN) :: comm
17         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19     MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror) !(_c)
20         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
21         TYPE(*), DIMENSION(..) :: recvbuf
22         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
23         TYPE(MPI_Datatype), INTENT(IN) :: datatype
24         TYPE(MPI_Op), INTENT(IN) :: op
25         INTEGER, INTENT(IN) :: root
26         TYPE(MPI_Comm), INTENT(IN) :: comm
27         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29     MPI_Reduce_init(sendbuf, recvbuf, count, datatype, op, root, comm, info,
30                     request, ierror)
31         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
32         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
33         INTEGER, INTENT(IN) :: count, root
34         TYPE(MPI_Datatype), INTENT(IN) :: datatype
35         TYPE(MPI_Op), INTENT(IN) :: op
36         TYPE(MPI_Comm), INTENT(IN) :: comm
37         TYPE(MPI_Info), INTENT(IN) :: info
38         TYPE(MPI_Request), INTENT(OUT) :: request
39         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41     MPI_Reduce_init(sendbuf, recvbuf, count, datatype, op, root, comm, info,
42                     request, ierror) !(_c)
43         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
44         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
45         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
46         TYPE(MPI_Datatype), INTENT(IN) :: datatype
47         TYPE(MPI_Op), INTENT(IN) :: op
48         INTEGER, INTENT(IN) :: root
49         TYPE(MPI_Comm), INTENT(IN) :: comm
50         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_local(inbuf, inoutbuf, count, datatype, op, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
TYPE(*), DIMENSION(..) :: inoutbuf
INTEGER(KIND=MPI_COUNT_KIND), 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, recvcnt, datatype, op, comm, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: recvcnt(*)
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(sendbuf, recvbuf, recvcnt, datatype, op, comm, ierror)
!(_c)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcnt(*)
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, recvcnt, datatype, op, comm,
ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: recvcnt
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, recvcnt, datatype, op, comm,
ierror) !(_c)

```

```

1     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
2     TYPE(*), DIMENSION(..) :: recvbuf
3     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcnt
4     TYPE(MPI_Datatype), INTENT(IN) :: datatype
5     TYPE(MPI_Op), INTENT(IN) :: op
6     TYPE(MPI_Comm), INTENT(IN) :: comm
7     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
9     MPI_Reduce_scatter_block_init(sendbuf, recvbuf, recvcnt, datatype, op, comm,
10        info, request, ierror)
11     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
12     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
13     INTEGER, INTENT(IN) :: recvcnt
14     TYPE(MPI_Datatype), INTENT(IN) :: datatype
15     TYPE(MPI_Op), INTENT(IN) :: op
16     TYPE(MPI_Comm), INTENT(IN) :: comm
17     TYPE(MPI_Info), INTENT(IN) :: info
18     TYPE(MPI_Request), INTENT(OUT) :: request
19     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21     MPI_Reduce_scatter_block_init(sendbuf, recvbuf, recvcnt, datatype, op, comm,
22        info, request, ierror) !(_c)
23     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
24     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
25     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcnt
26     TYPE(MPI_Datatype), INTENT(IN) :: datatype
27     TYPE(MPI_Op), INTENT(IN) :: op
28     TYPE(MPI_Comm), INTENT(IN) :: comm
29     TYPE(MPI_Info), INTENT(IN) :: info
30     TYPE(MPI_Request), INTENT(OUT) :: request
31     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33     MPI_Reduce_scatter_init(sendbuf, recvbuf, recvcnts, datatype, op, comm, info,
34        request, ierror)
35     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
36     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
37     INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcnts(*)
38     TYPE(MPI_Datatype), INTENT(IN) :: datatype
39     TYPE(MPI_Op), INTENT(IN) :: op
40     TYPE(MPI_Comm), INTENT(IN) :: comm
41     TYPE(MPI_Info), INTENT(IN) :: info
42     TYPE(MPI_Request), INTENT(OUT) :: request
43     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45     MPI_Reduce_scatter_init(sendbuf, recvbuf, recvcnts, datatype, op, comm, info,
46        request, ierror) !(_c)
47     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
48     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
49     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcnts(*)

```

```

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(sendbuf, recvbuf, count, datatype, op, comm, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER(KIND=MPI_COUNT_KIND), 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_Scan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
              ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER(KIND=MPI_COUNT_KIND), 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

```

```

1 MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root,
2           comm, ierror)
3     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
4     INTEGER, INTENT(IN) :: sendcount, recvcount, root
5     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
6     TYPE(*), DIMENSION(..) :: recvbuf
7     TYPE(MPI_Comm), INTENT(IN) :: comm
8     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10 MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root,
11            comm, ierror) !(_c)
12     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
13     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
14     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
15     TYPE(*), DIMENSION(..) :: recvbuf
16     INTEGER, INTENT(IN) :: root
17     TYPE(MPI_Comm), INTENT(IN) :: comm
18     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20 MPI_Scatter_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
21                root, comm, info, request, ierror)
22     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
23     INTEGER, INTENT(IN) :: sendcount, recvcount, root
24     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
25     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
26     TYPE(MPI_Comm), INTENT(IN) :: comm
27     TYPE(MPI_Info), INTENT(IN) :: info
28     TYPE(MPI_Request), INTENT(OUT) :: request
29     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31 MPI_Scatter_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
32                root, comm, info, request, ierror) !(_c)
33     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
34     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
35     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
36     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
37     INTEGER, INTENT(IN) :: root
38     TYPE(MPI_Comm), INTENT(IN) :: comm
39     TYPE(MPI_Info), INTENT(IN) :: info
40     TYPE(MPI_Request), INTENT(OUT) :: request
41     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43 MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
44             recvtype, root, comm, ierror)
45     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
46     INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root
47     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
48     TYPE(*), DIMENSION(..) :: recvbuf
49     TYPE(MPI_Comm), INTENT(IN) :: comm

```

```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1
MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, 2
            recvtype, root, comm, ierror) !(_c) 3
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf 4
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*), recvcount 5
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*) 6
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 7
TYPE(*), DIMENSION(..) :: recvbuf 8
INTEGER, INTENT(IN) :: root 9
TYPE(MPI_Comm), INTENT(IN) :: comm 10
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 11
MPI_Scatterv_init(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, 12
                recvtype, root, comm, info, request, ierror) 13
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 14
INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*) 15
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 16
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf 17
INTEGER, INTENT(IN) :: recvcount, root 18
TYPE(MPI_Comm), INTENT(IN) :: comm 19
TYPE(MPI_Info), INTENT(IN) :: info 20
TYPE(MPI_Request), INTENT(OUT) :: request 21
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22
MPI_Scatterv_init(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, 23
                recvtype, root, comm, info, request, ierror) !(_c) 24
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 25
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*) 26
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*) 27
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 28
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf 29
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount 30
INTEGER, INTENT(IN) :: root 31
TYPE(MPI_Comm), INTENT(IN) :: comm 32
TYPE(MPI_Info), INTENT(IN) :: info 33
TYPE(MPI_Request), INTENT(OUT) :: request 34
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 35
A.4.5 Groups, Contexts, Communicators, and Caching Fortran 2008 Bindings 36
MPI_Comm_compare(comm1, comm2, result, ierror) 37
TYPE(MPI_Comm), INTENT(IN) :: comm1, comm2 38
INTEGER, INTENT(OUT) :: result 39
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 40
MPI_Comm_create(comm, group, newcomm, ierror) 41
TYPE(MPI_Comm), INTENT(IN) :: comm 42
TYPE(MPI_Group), INTENT(IN) :: group 43

```

```

1     TYPE(MPI_Comm), INTENT(OUT) :: newcomm
2     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
4 MPI_Comm_create_from_group(group, stringtag, info, errhandler, newcomm, ierror)
5     TYPE(MPI_Group), INTENT(IN) :: group
6     CHARACTER(LEN=*), INTENT(IN) :: stringtag
7     TYPE(MPI_Info), INTENT(IN) :: info
8     TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
9     TYPE(MPI_Comm), INTENT(OUT) :: newcomm
10    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12 MPI_Comm_create_group(comm, group, tag, newcomm, ierror)
13    TYPE(MPI_Comm), INTENT(IN) :: comm
14    TYPE(MPI_Group), INTENT(IN) :: group
15    INTEGER, INTENT(IN) :: tag
16    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
17    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19 MPI_Comm_create_keyval(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval,
20     extra_state, ierror)
21    PROCEDURE(MPI_Comm_copy_attr_function) :: comm_copy_attr_fn
22    PROCEDURE(MPI_Comm_delete_attr_function) :: comm_delete_attr_fn
23    INTEGER, INTENT(OUT) :: comm_keyval
24    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
25    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27 MPI_Comm_delete_attr(comm, comm_keyval, ierror)
28    TYPE(MPI_Comm), INTENT(IN) :: comm
29    INTEGER, INTENT(IN) :: comm_keyval
30    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32 MPI_Comm_dup(comm, newcomm, ierror)
33    TYPE(MPI_Comm), INTENT(IN) :: comm
34    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
35    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37 MPI_COMM_DUP_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,
38     attribute_val_out, flag, ierror)
39    TYPE(MPI_Comm) :: oldcomm
40    INTEGER :: comm_keyval, ierror
41    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
42     attribute_val_out
43    LOGICAL :: flag
44
45 MPI_Comm_dup_with_info(comm, info, newcomm, ierror)
46    TYPE(MPI_Comm), INTENT(IN) :: comm
47    TYPE(MPI_Info), INTENT(IN) :: info
48    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
49    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
50
51 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_NULL_COPY_FN(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

MPI_COMM_NULL_DELETE_FN(comm, comm_keyval, attribute_val, extra_state, ierror)

```

```
1     TYPE(MPI_Comm) :: comm
2     INTEGER :: comm_keyval, ierror
3     INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
4
5     MPI_Comm_rank(comm, rank, ierror)
6     TYPE(MPI_Comm), INTENT(IN) :: comm
7     INTEGER, INTENT(OUT) :: rank
8     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10    MPI_Comm_remote_group(comm, group, ierror)
11    TYPE(MPI_Comm), INTENT(IN) :: comm
12    TYPE(MPI_Group), INTENT(OUT) :: group
13    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15    MPI_Comm_remote_size(comm, size, ierror)
16    TYPE(MPI_Comm), INTENT(IN) :: comm
17    INTEGER, INTENT(OUT) :: size
18    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20    MPI_Comm_set_attr(comm, comm_keyval, attribute_val, ierror)
21    TYPE(MPI_Comm), INTENT(IN) :: comm
22    INTEGER, INTENT(IN) :: comm_keyval
23    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
24    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26    MPI_Comm_set_info(comm, info, ierror)
27    TYPE(MPI_Comm), INTENT(IN) :: comm
28    TYPE(MPI_Info), INTENT(IN) :: info
29    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31    MPI_Comm_set_name(comm, comm_name, ierror)
32    TYPE(MPI_Comm), INTENT(IN) :: comm
33    CHARACTER(LEN=*), INTENT(IN) :: comm_name
34    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36    MPI_Comm_size(comm, size, ierror)
37    TYPE(MPI_Comm), INTENT(IN) :: comm
38    INTEGER, INTENT(OUT) :: size
39    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41    MPI_Comm_split(comm, color, key, newcomm, ierror)
42    TYPE(MPI_Comm), INTENT(IN) :: comm
43    INTEGER, INTENT(IN) :: color, key
44    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
45    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47    MPI_Comm_split_type(comm, split_type, key, info, newcomm, ierror)
48    TYPE(MPI_Comm), INTENT(IN) :: comm
49    INTEGER, INTENT(IN) :: split_type, key
50    TYPE(MPI_Info), INTENT(IN) :: info
51    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
52    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

MPI_Comm_test_inter(comm, flag, ierror)	1
TYPE(MPI_Comm), INTENT(IN) :: comm	2
LOGICAL, INTENT(OUT) :: flag	3
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	4
MPI_Group_compare(group1, group2, result, ierror)	5
TYPE(MPI_Group), INTENT(IN) :: group1, group2	6
INTEGER, INTENT(OUT) :: result	7
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	8
MPI_Group_difference(group1, group2, newgroup, ierror)	9
TYPE(MPI_Group), INTENT(IN) :: group1, group2	10
TYPE(MPI_Group), INTENT(OUT) :: newgroup	11
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	12
MPI_Group_excl(group, n, ranks, newgroup, ierror)	13
TYPE(MPI_Group), INTENT(IN) :: group	14
INTEGER, INTENT(IN) :: n, ranks(n)	15
TYPE(MPI_Group), INTENT(OUT) :: newgroup	16
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	17
MPI_Group_free(group, ierror)	18
TYPE(MPI_Group), INTENT(INOUT) :: group	19
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	20
MPI_Group_from_session_pset(session, pset_name, newgroup, ierror)	21
TYPE(MPI_Session), INTENT(IN) :: session	22
CHARACTER(LEN=*), INTENT(IN) :: pset_name	23
TYPE(MPI_Group), INTENT(OUT) :: newgroup	24
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	25
MPI_Group_incl(group, n, ranks, newgroup, ierror)	26
TYPE(MPI_Group), INTENT(IN) :: group	27
INTEGER, INTENT(IN) :: n, ranks(n)	28
TYPE(MPI_Group), INTENT(OUT) :: newgroup	29
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	30
MPI_Group_intersection(group1, group2, newgroup, ierror)	31
TYPE(MPI_Group), INTENT(IN) :: group1, group2	32
TYPE(MPI_Group), INTENT(OUT) :: newgroup	33
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	34
MPI_Group_range_excl(group, n, ranges, newgroup, ierror)	35
TYPE(MPI_Group), INTENT(IN) :: group	36
INTEGER, INTENT(IN) :: n, ranges(3, n)	37
TYPE(MPI_Group), INTENT(OUT) :: newgroup	38
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	39
MPI_Group_range_incl(group, n, ranges, newgroup, ierror)	40
TYPE(MPI_Group), INTENT(IN) :: group	41
INTEGER, INTENT(IN) :: n, ranges(3, n)	42
TYPE(MPI_Group), INTENT(OUT) :: newgroup	43
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	44
MPI_Group_range_incl(group, n, ranges, newgroup, ierror)	45
TYPE(MPI_Group), INTENT(IN) :: group	46
INTEGER, INTENT(IN) :: n, ranges(3, n)	47
TYPE(MPI_Group), INTENT(OUT) :: newgroup	48

```

1     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
2
3 MPI_Group_rank(group, rank, ierror)
4     TYPE(MPI_Group), INTENT(IN) :: group
5     INTEGER, INTENT(OUT) :: rank
6     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8 MPI_Group_size(group, size, ierror)
9     TYPE(MPI_Group), INTENT(IN) :: group
10    INTEGER, INTENT(OUT) :: size
11    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13 MPI_Group_translate_ranks(group1, n, ranks1, group2, ranks2, ierror)
14    TYPE(MPI_Group), INTENT(IN) :: group1, group2
15    INTEGER, INTENT(IN) :: n, ranks1(n)
16    INTEGER, INTENT(OUT) :: ranks2(n)
17    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19 MPI_Group_union(group1, group2, newgroup, ierror)
20    TYPE(MPI_Group), INTENT(IN) :: group1, group2
21    TYPE(MPI_Group), INTENT(OUT) :: newgroup
22    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
24 MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader, tag,
25                     newintercomm, ierror)
26    TYPE(MPI_Comm), INTENT(IN) :: local_comm, peer_comm
27    INTEGER, INTENT(IN) :: local_leader, remote_leader, tag
28    TYPE(MPI_Comm), INTENT(OUT) :: newintercomm
29    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31 MPI_Intercomm_create_from_groups(local_group, local_leader, remote_group,
32                                 remote_leader, stringtag, info, errhandler, newintercomm, ierror)
33    TYPE(MPI_Group), INTENT(IN) :: local_group, remote_group
34    INTEGER, INTENT(IN) :: local_leader, remote_leader
35    CHARACTER(LEN=*), INTENT(IN) :: stringtag
36    TYPE(MPI_Info), INTENT(IN) :: info
37    TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
38    TYPE(MPI_Comm), INTENT(OUT) :: newintercomm
39    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41 MPI_Intercomm_merge(intercomm, high, newintracomm, ierror)
42    TYPE(MPI_Comm), INTENT(IN) :: intercomm
43    LOGICAL, INTENT(IN) :: high
44    TYPE(MPI_Comm), INTENT(OUT) :: newintracomm
45    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47 MPI_Type_create_keyval(type_copy_attr_fn, type_delete_attr_fn, type_keyval,
48                       extra_state, ierror)
49    PROCEDURE(MPI_Type_copy_attr_function) :: type_copy_attr_fn
50    PROCEDURE(MPI_Type_delete_attr_function) :: type_delete_attr_fn
51    INTEGER, INTENT(OUT) :: type_keyval

```

```

INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state      1
INTEGER, OPTIONAL, INTENT(OUT) :: ierror                      2
                                                                3
MPI_Type_delete_attr(datatype, type_keyval, ierror)           4
  TYPE(MPI_Datatype), INTENT(IN) :: datatype                  5
  INTEGER, INTENT(IN) :: type_keyval                          6
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror                    7
                                                                8
MPI_TYPE_DUP_FN(oldtype, type_keyval, extra_state, attribute_val_in,
                attribute_val_out, flag, ierror)              9
  TYPE(MPI_Datatype) :: oldtype                               10
  INTEGER :: type_keyval, ierror                              11
  INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                attribute_val_out                             12
  LOGICAL :: flag                                             13
                                                                14
MPI_Type_free_keyval(type_keyval, ierror)                     15
  INTEGER, INTENT(INOUT) :: type_keyval                       16
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror                    17
                                                                18
MPI_Type_get_attr(datatype, type_keyval, attribute_val, flag, ierror) 19
  TYPE(MPI_Datatype), INTENT(IN) :: datatype                  20
  INTEGER, INTENT(IN) :: type_keyval                          21
  INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val 22
  LOGICAL, INTENT(OUT) :: flag                                23
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror                    24
                                                                25
MPI_Type_get_name(datatype, type_name, resultlen, ierror)     26
  TYPE(MPI_Datatype), INTENT(IN) :: datatype                  27
  CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: type_name 28
  INTEGER, INTENT(OUT) :: resultlen                           29
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror                    30
                                                                31
MPI_TYPE_NULL_COPY_FN(oldtype, type_keyval, extra_state, attribute_val_in,
                    attribute_val_out, flag, ierror)           32
  TYPE(MPI_Datatype) :: oldtype                               33
  INTEGER :: type_keyval, ierror                              34
  INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                    attribute_val_out                         35
  LOGICAL :: flag                                             36
                                                                37
MPI_TYPE_NULL_DELETE_FN(datatype, type_keyval, attribute_val, extra_state,
                       ierror)                                 38
  TYPE(MPI_Datatype) :: datatype                              39
  INTEGER :: type_keyval                                      40
  INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state 41
  INTEGER, INTENT(OUT) :: ierror                              42
                                                                43
MPI_Type_set_attr(datatype, type_keyval, attribute_val, ierror) 44
  TYPE(MPI_Datatype), INTENT(IN) :: datatype                  45
  INTEGER, INTENT(IN) :: type_keyval                          46
  INTEGER, INTENT(IN) :: attribute_val                         47
  INTEGER, INTENT(OUT) :: ierror                              48

```

```

1     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
2     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
4     MPI_Type_set_name(datatype, type_name, ierror)
5     TYPE(MPI_Datatype), INTENT(IN) :: datatype
6     CHARACTER(LEN=*), INTENT(IN) :: type_name
7     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
9     MPI_Win_create_keyval(win_copy_attr_fn, win_delete_attr_fn, win_keyval,
10        extra_state, ierror)
11    PROCEDURE(MPI_Win_copy_attr_function) :: win_copy_attr_fn
12    PROCEDURE(MPI_Win_delete_attr_function) :: win_delete_attr_fn
13    INTEGER, INTENT(OUT) :: win_keyval
14    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
15    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17    MPI_Win_delete_attr(win, win_keyval, ierror)
18    TYPE(MPI_Win), INTENT(IN) :: win
19    INTEGER, INTENT(IN) :: win_keyval
20    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22    MPI_WIN_DUP_FN(oldwin, win_keyval, extra_state, attribute_val_in,
23        attribute_val_out, flag, ierror)
24    TYPE(MPI_Win) :: oldwin
25    INTEGER :: win_keyval, ierror
26    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
27        attribute_val_out
28    LOGICAL :: flag
29
30    MPI_Win_free_keyval(win_keyval, ierror)
31    INTEGER, INTENT(INOUT) :: win_keyval
32    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34    MPI_Win_get_attr(win, win_keyval, attribute_val, flag, ierror)
35    TYPE(MPI_Win), INTENT(IN) :: win
36    INTEGER, INTENT(IN) :: win_keyval
37    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
38    LOGICAL, INTENT(OUT) :: flag
39    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41    MPI_Win_get_name(win, win_name, resultlen, ierror)
42    TYPE(MPI_Win), INTENT(IN) :: win
43    CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: win_name
44    INTEGER, INTENT(OUT) :: resultlen
45    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47    MPI_WIN_NULL_COPY_FN(oldwin, win_keyval, extra_state, attribute_val_in,
48        attribute_val_out, flag, ierror)
49    TYPE(MPI_Win) :: oldwin
50    INTEGER :: 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)
TYPE(MPI_Win) :: win
INTEGER :: win_keyval, ierror
INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state

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.4.6 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

```

```

1     INTEGER, INTENT(IN) :: coords(*)
2     INTEGER, INTENT(OUT) :: rank
3     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
5     MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror)
6         TYPE(MPI_Comm), INTENT(IN) :: comm
7         INTEGER, INTENT(IN) :: direction, disp
8         INTEGER, INTENT(OUT) :: rank_source, rank_dest
9         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11    MPI_Cart_sub(comm, remain_dims, newcomm, ierror)
12        TYPE(MPI_Comm), INTENT(IN) :: comm
13        LOGICAL, INTENT(IN) :: remain_dims(*)
14        TYPE(MPI_Comm), INTENT(OUT) :: newcomm
15        INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17    MPI_Cartdim_get(comm, ndims, ierror)
18        TYPE(MPI_Comm), INTENT(IN) :: comm
19        INTEGER, INTENT(OUT) :: ndims
20        INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22    MPI_Dims_create(nnodes, ndims, dims, ierror)
23        INTEGER, INTENT(IN) :: nnodes, ndims
24        INTEGER, INTENT(INOUT) :: dims(ndims)
25        INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27    MPI_Dist_graph_create(comm_old, n, sources, degrees, destinations, weights,
28        info, reorder, comm_dist_graph, ierror)
29        TYPE(MPI_Comm), INTENT(IN) :: comm_old
30        INTEGER, INTENT(IN) :: n, sources(n), degrees(n), destinations(*),
31        weights(*)
32        TYPE(MPI_Info), INTENT(IN) :: info
33        LOGICAL, INTENT(IN) :: reorder
34        TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
35        INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37    MPI_Dist_graph_create_adjacent(comm_old, indegree, sources, sourceweights,
38        outdegree, destinations, destweights, info, reorder,
39        comm_dist_graph, ierror)
40        TYPE(MPI_Comm), INTENT(IN) :: comm_old
41        INTEGER, INTENT(IN) :: indegree, sources(indegree), sourceweights(*),
42        outdegree, destinations(outdegree), destweights(*)
43        TYPE(MPI_Info), INTENT(IN) :: info
44        LOGICAL, INTENT(IN) :: reorder
45        TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
46        INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48    MPI_Dist_graph_neighbors(comm, maxindegree, sources, sourceweights,
49        maxoutdegree, destinations, destweights, ierror)
50        TYPE(MPI_Comm), INTENT(IN) :: comm
51        INTEGER, INTENT(IN) :: maxindegree, maxoutdegree

```



```

    INTEGER, INTENT(OUT) :: sources(maxindegree), destinations(maxoutdegree) 1
    INTEGER :: sourceweights(*), destweights(*) 2
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror 3
 4
MPI_Dist_graph_neighbors_count(comm, indegree, outdegree, weighted, ierror) 5
    TYPE(MPI_Comm), INTENT(IN) :: comm 6
    INTEGER, INTENT(OUT) :: indegree, outdegree 7
    LOGICAL, INTENT(OUT) :: weighted 8
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror 9
 10
MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph, ierror) 11
    TYPE(MPI_Comm), INTENT(IN) :: comm_old 12
    INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*) 13
    LOGICAL, INTENT(IN) :: reorder 14
    TYPE(MPI_Comm), INTENT(OUT) :: comm_graph 15
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror 16
 17
MPI_Graph_get(comm, maxindex, maxedges, index, edges, ierror) 18
    TYPE(MPI_Comm), INTENT(IN) :: comm 19
    INTEGER, INTENT(IN) :: maxindex, maxedges 20
    INTEGER, INTENT(OUT) :: index(maxindex), edges(maxedges) 21
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22
 23
MPI_Graph_map(comm, nnodes, index, edges, newrank, ierror) 24
    TYPE(MPI_Comm), INTENT(IN) :: comm 25
    INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*) 26
    INTEGER, INTENT(OUT) :: newrank 27
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror 28
 29
MPI_Graph_neighbors(comm, rank, maxneighbors, neighbors, ierror) 30
    TYPE(MPI_Comm), INTENT(IN) :: comm 31
    INTEGER, INTENT(IN) :: rank, maxneighbors 32
    INTEGER, INTENT(OUT) :: neighbors(maxneighbors) 33
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror 34
 35
MPI_Graph_neighbors_count(comm, rank, nneighbors, ierror) 36
    TYPE(MPI_Comm), INTENT(IN) :: comm 37
    INTEGER, INTENT(IN) :: rank 38
    INTEGER, INTENT(OUT) :: nneighbors 39
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror 40
 41
MPI_Graphdims_get(comm, nnodes, nedges, ierror) 42
    TYPE(MPI_Comm), INTENT(IN) :: comm 43
    INTEGER, INTENT(OUT) :: nnodes, nedges 44
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror 45
 46
MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
    recvtype, comm, request, ierror) 47
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 48
    INTEGER, INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype

```

```

1     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
2     TYPE(MPI_Comm), INTENT(IN) :: comm
3     TYPE(MPI_Request), INTENT(OUT) :: request
4     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
6 MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
7     recvtype, comm, request, ierror) !(_c)
8     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
9     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
10    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
11    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
12    TYPE(MPI_Comm), INTENT(IN) :: comm
13    TYPE(MPI_Request), INTENT(OUT) :: request
14    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16 MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
17     displs, recvtype, comm, request, ierror)
18    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
19    INTEGER, INTENT(IN) :: sendcount
20    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
21    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
22    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
23    TYPE(MPI_Comm), INTENT(IN) :: comm
24    TYPE(MPI_Request), INTENT(OUT) :: request
25    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27 MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
28     displs, recvtype, comm, request, ierror) !(_c)
29    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
30    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
31    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
32    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
33    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
34    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
35    TYPE(MPI_Comm), INTENT(IN) :: comm
36    TYPE(MPI_Request), INTENT(OUT) :: request
37    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39 MPI_Ineighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
40     recvtype, comm, request, ierror)
41    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
42    INTEGER, INTENT(IN) :: sendcount, recvcount
43    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
44    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
45    TYPE(MPI_Comm), INTENT(IN) :: comm
46    TYPE(MPI_Request), INTENT(OUT) :: request
47    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
49 MPI_Ineighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
50     recvtype, comm, request, ierror) !(_c)

```

```

TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf           1
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount      2
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype                 3
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf                     4
TYPE(MPI_Comm), INTENT(IN) :: comm                                  5
TYPE(MPI_Request), INTENT(OUT) :: request                          6
INTEGER, OPTIONAL, INTENT(OUT) :: ierror                          7
                                                                    8
MPI_Ineighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,  9
                        recvcnts, rdispls, recvtype, comm, request, ierror) 10
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf           11
INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),      12
                        recvcnts(*), rdispls(*)                       13
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype                 14
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf                     15
TYPE(MPI_Comm), INTENT(IN) :: comm                                  16
TYPE(MPI_Request), INTENT(OUT) :: request                          17
INTEGER, OPTIONAL, INTENT(OUT) :: ierror                          18
                                                                    19
MPI_Ineighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,  20
                        recvcnts, rdispls, recvtype, comm, request, ierror) !(_c) 21
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf           22
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*), 23
                        recvcnts(*)                                   24
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*), 25
                        rdispls(*)                                   26
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype                 27
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf                     28
TYPE(MPI_Comm), INTENT(IN) :: comm                                  29
TYPE(MPI_Request), INTENT(OUT) :: request                          30
INTEGER, OPTIONAL, INTENT(OUT) :: ierror                          31
                                                                    32
MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, 33
                        recvcnts, rdispls, recvtypes, comm, request, ierror) 34
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf           35
INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcnts(*)      36
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*), 37
                        rdispls(*)                                   38
TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*), recvtypes(*) 39
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf                     40
TYPE(MPI_Comm), INTENT(IN) :: comm                                  41
TYPE(MPI_Request), INTENT(OUT) :: request                          42
INTEGER, OPTIONAL, INTENT(OUT) :: ierror                          43
                                                                    44
MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, 45
                        recvcnts, rdispls, recvtypes, comm, request, ierror) !(_c) 46
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf           47
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*), 48
                        recvcnts(*)

```

```

1     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
2         rdispls(*)
3     TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*), recvtypes(*)
4     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
5     TYPE(MPI_Comm), INTENT(IN) :: comm
6     TYPE(MPI_Request), INTENT(OUT) :: request
7     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
9     MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
10        recvtype, comm, ierror)
11     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
12     INTEGER, INTENT(IN) :: sendcount, recvcount
13     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
14     TYPE(*), DIMENSION(..) :: recvbuf
15     TYPE(MPI_Comm), INTENT(IN) :: comm
16     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18     MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
19        recvtype, comm, ierror) !(_c)
20     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
21     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
22     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
23     TYPE(*), DIMENSION(..) :: recvbuf
24     TYPE(MPI_Comm), INTENT(IN) :: comm
25     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27     MPI_Neighbor_allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
28        recvtype, comm, info, request, ierror)
29     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
30     INTEGER, INTENT(IN) :: sendcount, recvcount
31     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
32     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
33     TYPE(MPI_Comm), INTENT(IN) :: comm
34     TYPE(MPI_Info), INTENT(IN) :: info
35     TYPE(MPI_Request), INTENT(OUT) :: request
36     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38     MPI_Neighbor_allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
39        recvtype, comm, info, request, ierror) !(_c)
40     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
41     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
42     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
43     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
44     TYPE(MPI_Comm), INTENT(IN) :: comm
45     TYPE(MPI_Info), INTENT(IN) :: info
46     TYPE(MPI_Request), INTENT(OUT) :: request
47     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
49     MPI_Neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
50        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(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
    displs, recvtype, comm, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcounts(*)
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
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
INTEGER, INTENT(IN) :: sendcount, displs(*)
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
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_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
    displs, recvtype, comm, info, request, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
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, recvcoun,
    recvtype, comm, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
INTEGER, INTENT(IN) :: sendcount, recvcoun
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..) :: recvbuf
TYPE(MPI_Comm), INTENT(IN) :: comm

```

```

1      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
2
3      MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
4          recvtype, comm, ierror) !(_c)
5      TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
6      INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
7      TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
8      TYPE(*), DIMENSION(..) :: recvbuf
9      TYPE(MPI_Comm), INTENT(IN) :: comm
10     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12     MPI_Neighbor_alltoall_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
13         recvtype, comm, info, request, ierror)
14     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
15     INTEGER, INTENT(IN) :: sendcount, recvcount
16     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
17     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
18     TYPE(MPI_Comm), INTENT(IN) :: comm
19     TYPE(MPI_Info), INTENT(IN) :: info
20     TYPE(MPI_Request), INTENT(OUT) :: request
21     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23     MPI_Neighbor_alltoall_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
24         recvtype, comm, info, request, ierror) !(_c)
25     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
26     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
27     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
28     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
29     TYPE(MPI_Comm), INTENT(IN) :: comm
30     TYPE(MPI_Info), INTENT(IN) :: info
31     TYPE(MPI_Request), INTENT(OUT) :: request
32     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34     MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
35         recvcounts, rdispls, recvtype, comm, ierror)
36     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
37     INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*), rdispls(*)
38     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
39     TYPE(*), DIMENSION(..) :: recvbuf
40     TYPE(MPI_Comm), INTENT(IN) :: comm
41     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43     MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
44         recvcounts, rdispls, recvtype, comm, ierror) !(_c)
45     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
46     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*), recvcounts(*)
47     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
48     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
49     TYPE(*), DIMENSION(..) :: recvbuf
50     TYPE(MPI_Comm), INTENT(IN) :: comm

```

```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror
1
MPI_Neighbor_alltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
MPI_Neighbor_alltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
    recvcounts, rdispls, recvtype, comm, info, request, ierror)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
    recvcounts(*), rdispls(*)
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Info), INTENT(IN) :: info
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Neighbor_alltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
    recvcounts, rdispls, recvtype, comm, info, request, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*),
    recvcounts(*)
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
    rdispls(*)
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Info), INTENT(IN) :: info
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
    recvcounts, rdispls, recvtypes, comm, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
INTEGER, INTENT(IN) :: sendcounts(*), recvcounts(*)
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
TYPE(*), DIMENSION(..) :: recvbuf
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
    recvcounts, rdispls, recvtypes, comm, ierror) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*), recvcounts(*)
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
TYPE(*), DIMENSION(..) :: recvbuf
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)

```

```

1     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
2     INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
3     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
4         rdispls(*)
5     TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*), recvtypes(*)
6     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
7     TYPE(MPI_Comm), INTENT(IN) :: comm
8     TYPE(MPI_Info), INTENT(IN) :: info
9     TYPE(MPI_Request), INTENT(OUT) :: request
10    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12    MPI_Neighbor_alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
13        recvcounts, rdispls, recvtypes, comm, info, request, ierror)
14        !(_c)
15    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
16    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*),
17        recvcounts(*)
18    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
19        rdispls(*)
20    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*), recvtypes(*)
21    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
22    TYPE(MPI_Comm), INTENT(IN) :: comm
23    TYPE(MPI_Info), INTENT(IN) :: info
24    TYPE(MPI_Request), INTENT(OUT) :: request
25    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27    MPI_Topo_test(comm, status, ierror)
28    TYPE(MPI_Comm), INTENT(IN) :: comm
29    INTEGER, INTENT(OUT) :: status
30    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32    A.4.7 MPI Environmental Management Fortran 2008 Bindings
33
34    DOUBLE PRECISION MPI_Wtick()
35
36    DOUBLE PRECISION MPI_Wtime()
37
38    MPI_Add_error_class(errorclass, ierror)
39    INTEGER, INTENT(OUT) :: errorclass
40    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42    MPI_Add_error_code(errorclass, errorcode, ierror)
43    INTEGER, INTENT(IN) :: errorclass
44    INTEGER, INTENT(OUT) :: errorcode
45    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47    MPI_Add_error_string(errorcode, string, ierror)
48    INTEGER, INTENT(IN) :: errorcode
49    CHARACTER(LEN=*), INTENT(IN) :: string
50    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```



```

MPI_Alloc_mem(size, info, baseptr, ierror)                                1
  USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR                          2
  INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size                    3
  TYPE(MPI_Info), INTENT(IN) :: info                                    4
  TYPE(C_PTR), INTENT(OUT) :: baseptr                                  5
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror                             6
                                                                 7
MPI_Comm_call_errhandler(comm, errorcode, ierror)                       8
  TYPE(MPI_Comm), INTENT(IN) :: comm                                   9
  INTEGER, INTENT(IN) :: errorcode                                    10
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror                             11
                                                                 12
MPI_Comm_create_errhandler(comm_errhandler_fn, errhandler, ierror)     13
  PROCEDURE(MPI_Comm_errhandler_function) :: comm_errhandler_fn      14
  TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler                    15
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror                             16
                                                                 17
MPI_Comm_get_errhandler(comm, errhandler, ierror)                      18
  TYPE(MPI_Comm), INTENT(IN) :: comm                                   19
  TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler                    20
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror                             21
                                                                 22
MPI_Comm_set_errhandler(comm, errhandler, ierror)                      23
  TYPE(MPI_Comm), INTENT(IN) :: comm                                   24
  TYPE(MPI_Errhandler), INTENT(IN) :: errhandler                     25
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror                             26
                                                                 27
MPI_Errhandler_free(errhandler, ierror)                                  28
  TYPE(MPI_Errhandler), INTENT(INOUT) :: errhandler                  29
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror                             30
                                                                 31
MPI_Error_class(errorcode, errorclass, ierror)                          32
  INTEGER, INTENT(IN) :: errorcode                                    33
  INTEGER, INTENT(OUT) :: errorclass                                  34
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror                             35
                                                                 36
MPI_Error_string(errorcode, string, resultlen, ierror)                  37
  INTEGER, INTENT(IN) :: errorcode                                    38
  CHARACTER(LEN=MPI_MAX_ERROR_STRING), INTENT(OUT) :: string        39
  INTEGER, INTENT(OUT) :: resultlen                                  40
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror                             41
                                                                 42
MPI_File_call_errhandler(fh, errorcode, ierror)                         43
  TYPE(MPI_File), INTENT(IN) :: fh                                    44
  INTEGER, INTENT(IN) :: errorcode                                    45
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror                             46
                                                                 47
MPI_File_create_errhandler(file_errhandler_fn, errhandler, ierror)     48
  PROCEDURE(MPI_File_errhandler_function) :: file_errhandler_fn
  TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```
1 MPI_File_get_errhandler(file, errhandler, ierror)
2   TYPE(MPI_File), INTENT(IN) :: file
3   TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
4   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
6 MPI_File_set_errhandler(file, errhandler, ierror)
7   TYPE(MPI_File), INTENT(IN) :: file
8   TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
9   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11 MPI_Free_mem(base, ierror)
12   TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: base
13   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15 MPI_Get_library_version(version, resultlen, ierror)
16   CHARACTER(LEN=MPI_MAX_LIBRARY_VERSION_STRING), INTENT(OUT) :: version
17   INTEGER, INTENT(OUT) :: resultlen
18   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20 MPI_Get_processor_name(name, resultlen, ierror)
21   CHARACTER(LEN=MPI_MAX_PROCESSOR_NAME), INTENT(OUT) :: name
22   INTEGER, INTENT(OUT) :: resultlen
23   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
25 MPI_Get_version(version, subversion, ierror)
26   INTEGER, INTENT(OUT) :: version, subversion
27   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29 MPI_Session_call_errhandler(session, errorcode, ierror)
30   TYPE(MPI_Session), INTENT(IN) :: session
31   INTEGER, INTENT(IN) :: errorcode
32   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34 MPI_Session_create_errhandler(session_errhandler_fn, errhandler, ierror)
35   PROCEDURE(MPI_Session_errhandler_function) :: session_errhandler_fn
36   TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
37   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39 MPI_Session_get_errhandler(session, errhandler, ierror)
40   TYPE(MPI_Session), INTENT(IN) :: session
41   TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
42   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44 MPI_Session_set_errhandler(session, errhandler, ierror)
45   TYPE(MPI_Session), INTENT(IN) :: session
46   TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
47   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
49 MPI_Win_call_errhandler(win, errorcode, ierror)
50   TYPE(MPI_Win), INTENT(IN) :: win
51   INTEGER, INTENT(IN) :: errorcode
52   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

MPI_Win_create_errhandler(win_errhandler_fn, errhandler, ierror)	1
PROCEDURE(MPI_Win_errhandler_function) :: win_errhandler_fn	2
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler	3
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	4
	5
MPI_Win_get_errhandler(win, errhandler, ierror)	6
TYPE(MPI_Win), INTENT(IN) :: win	7
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler	8
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	9
	10
MPI_Win_set_errhandler(win, errhandler, ierror)	11
TYPE(MPI_Win), INTENT(IN) :: win	12
TYPE(MPI_Errhandler), INTENT(IN) :: errhandler	13
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	14
	15
A.4.8 The Info Object Fortran 2008 Bindings	16
	17
MPI_Info_create(info, ierror)	18
TYPE(MPI_Info), INTENT(OUT) :: info	19
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	20
	21
MPI_Info_create_env(info, ierror)	22
TYPE(MPI_Info), INTENT(OUT) :: info	23
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	24
	25
MPI_Info_delete(info, key, ierror)	26
TYPE(MPI_Info), INTENT(IN) :: info	27
CHARACTER(LEN=*), INTENT(IN) :: key	28
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	29
	30
MPI_Info_dup(info, newinfo, ierror)	31
TYPE(MPI_Info), INTENT(IN) :: info	32
TYPE(MPI_Info), INTENT(OUT) :: newinfo	33
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	34
	35
MPI_Info_free(info, ierror)	36
TYPE(MPI_Info), INTENT(INOUT) :: info	37
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	38
	39
MPI_Info_get_nkeys(info, nkeys, ierror)	40
TYPE(MPI_Info), INTENT(IN) :: info	41
INTEGER, INTENT(OUT) :: nkeys	42
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	43
	44
MPI_Info_get_nthkey(info, n, key, ierror)	45
TYPE(MPI_Info), INTENT(IN) :: info	46
INTEGER, INTENT(IN) :: n	47
CHARACTER(LEN=*), INTENT(OUT) :: key	48
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	
MPI_Info_get_string(info, key, buflen, value, flag, ierror)	
TYPE(MPI_Info), INTENT(IN) :: info	

```

1     CHARACTER(LEN=*), INTENT(IN) :: key
2     INTEGER, INTENT(INOUT) :: buflen
3     CHARACTER(LEN=*), INTENT(OUT) :: value
4     LOGICAL, INTENT(OUT) :: flag
5     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
7     MPI_Info_set(info, key, value, ierror)
8     TYPE(MPI_Info), INTENT(IN) :: info
9     CHARACTER(LEN=*), INTENT(IN) :: key, value
10    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12    A.4.9 Process Creation and Management Fortran 2008 Bindings
13
14    MPI_Abort(comm, errorcode, ierror)
15        TYPE(MPI_Comm), INTENT(IN) :: comm
16        INTEGER, INTENT(IN) :: errorcode
17        INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19    MPI_Close_port(port_name, ierror)
20        CHARACTER(LEN=*), INTENT(IN) :: port_name
21        INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23    MPI_Comm_accept(port_name, info, root, comm, newcomm, ierror)
24        CHARACTER(LEN=*), INTENT(IN) :: port_name
25        TYPE(MPI_Info), INTENT(IN) :: info
26        INTEGER, INTENT(IN) :: root
27        TYPE(MPI_Comm), INTENT(IN) :: comm
28        TYPE(MPI_Comm), INTENT(OUT) :: newcomm
29        INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31    MPI_Comm_connect(port_name, info, root, comm, newcomm, ierror)
32        CHARACTER(LEN=*), INTENT(IN) :: port_name
33        TYPE(MPI_Info), INTENT(IN) :: info
34        INTEGER, INTENT(IN) :: root
35        TYPE(MPI_Comm), INTENT(IN) :: comm
36        TYPE(MPI_Comm), INTENT(OUT) :: newcomm
37        INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39    MPI_Comm_disconnect(comm, ierror)
40        TYPE(MPI_Comm), INTENT(INOUT) :: comm
41        INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43    MPI_Comm_get_parent(parent, ierror)
44        TYPE(MPI_Comm), INTENT(OUT) :: parent
45        INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47    MPI_Comm_join(fd, intercomm, ierror)
48        INTEGER, INTENT(IN) :: fd
49        TYPE(MPI_Comm), INTENT(OUT) :: intercomm
50        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_Finalize(ierror)
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Finalized(flag, ierror)
LOGICAL, INTENT(OUT) :: flag
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Init(ierror)
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_Initialized(flag, ierror)
LOGICAL, INTENT(OUT) :: flag
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Is_thread_main(flag, ierror)
LOGICAL, INTENT(OUT) :: flag
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)

```

```
1     TYPE(MPI_Info), INTENT(IN) :: info
2     CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name
3     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
5     MPI_Publish_name(service_name, info, port_name, ierror)
6     CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name
7     TYPE(MPI_Info), INTENT(IN) :: info
8     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10    MPI_Query_thread(provided, ierror)
11    INTEGER, INTENT(OUT) :: provided
12    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14    MPI_Session_finalize(session, ierror)
15    TYPE(MPI_Session), INTENT(INOUT) :: session
16    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18    MPI_Session_get_info(session, info_used, ierror)
19    TYPE(MPI_Session), INTENT(IN) :: session
20    TYPE(MPI_Info), INTENT(OUT) :: info_used
21    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23    MPI_Session_get_nth_pset(session, info, n, pset_len, pset_name, ierror)
24    TYPE(MPI_Session), INTENT(IN) :: session
25    TYPE(MPI_Info), INTENT(IN) :: info
26    INTEGER, INTENT(IN) :: n
27    INTEGER, INTENT(INOUT) :: pset_len
28    CHARACTER(LEN=*), INTENT(OUT) :: pset_name
29    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31    MPI_Session_get_num_psets(session, info, npset_names, ierror)
32    TYPE(MPI_Session), INTENT(IN) :: session
33    TYPE(MPI_Info), INTENT(IN) :: info
34    INTEGER, INTENT(OUT) :: npset_names
35    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37    MPI_Session_get_pset_info(session, pset_name, info, ierror)
38    TYPE(MPI_Session), INTENT(IN) :: session
39    CHARACTER(LEN=*), INTENT(IN) :: pset_name
40    TYPE(MPI_Info), INTENT(OUT) :: info
41    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43    MPI_Session_init(info, errhandler, session, ierror)
44    TYPE(MPI_Info), INTENT(IN) :: info
45    TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
46    TYPE(MPI_Session), INTENT(OUT) :: session
47    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
49    MPI_Unpublish_name(service_name, info, port_name, ierror)
50    CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name
51    TYPE(MPI_Info), INTENT(IN) :: info
52    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

A.4.10 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_Accumulate(origin_addr, origin_count, origin_datatype, target_rank,
               target_disp, target_count, target_datatype, op, win, ierror)
    !(_c)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_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_Compare_and_swap(origin_addr, compare_addr, result_addr, datatype,
                     target_rank, target_disp, win, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr,
    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

```

```

1     TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
2     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
3     TYPE(MPI_Win), INTENT(IN) :: win
4     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
6 MPI_Get(origin_addr, origin_count, origin_datatype, target_rank, target_disp,
7         target_count, target_datatype, win, ierror) !(_c)
8     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: origin_addr
9     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
10    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
11    INTEGER, INTENT(IN) :: target_rank
12    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
13    TYPE(MPI_Win), INTENT(IN) :: win
14    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16 MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
17                   result_count, result_datatype, target_rank, target_disp,
18                   target_count, target_datatype, op, win, ierror)
19    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
20    INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
21                   target_count
22    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
23                   target_datatype
24    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: result_addr
25    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
26    TYPE(MPI_Op), INTENT(IN) :: op
27    TYPE(MPI_Win), INTENT(IN) :: win
28    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30 MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
31                   result_count, result_datatype, target_rank, target_disp,
32                   target_count, target_datatype, op, win, ierror) !(_c)
33    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
34    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, result_count,
35                   target_count
36    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
37                   target_datatype
38    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: result_addr
39    INTEGER, INTENT(IN) :: target_rank
40    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
41    TYPE(MPI_Op), INTENT(IN) :: op
42    TYPE(MPI_Win), INTENT(IN) :: win
43    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45 MPI_Put(origin_addr, origin_count, origin_datatype, target_rank, target_disp,
46         target_count, target_datatype, win, ierror)
47    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
48    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
49    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype

```



```

INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp      1
TYPE(MPI_Win), INTENT(IN) :: win                               2
INTEGER, OPTIONAL, INTENT(OUT) :: ierror                       3
                                                                4
MPI_Put(origin_addr, origin_count, origin_datatype, target_rank, target_disp,
        target_count, target_datatype, win, ierror) !(_c)      5
                                                                6
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr 7
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count 8
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype 9
INTEGER, INTENT(IN) :: target_rank                             10
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp     11
TYPE(MPI_Win), INTENT(IN) :: win                               12
INTEGER, OPTIONAL, INTENT(OUT) :: ierror                       13
                                                                14
MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
        target_disp, target_count, target_datatype, op, win, request,
        ierror)                                               15
                                                                16
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr 17
INTEGER, INTENT(IN) :: origin_count, target_rank, target_count 18
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype 19
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp     20
TYPE(MPI_Op), INTENT(IN) :: op                                 21
TYPE(MPI_Win), INTENT(IN) :: win                               22
TYPE(MPI_Request), INTENT(OUT) :: request                     23
INTEGER, OPTIONAL, INTENT(OUT) :: ierror                       24
                                                                25
MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
        target_disp, target_count, target_datatype, op, win, request,
        ierror) !(_c)                                         26
                                                                27
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr 28
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count 29
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype 30
INTEGER, INTENT(IN) :: target_rank                             31
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp     32
TYPE(MPI_Op), INTENT(IN) :: op                                 33
TYPE(MPI_Win), INTENT(IN) :: win                               34
TYPE(MPI_Request), INTENT(OUT) :: request                     35
INTEGER, OPTIONAL, INTENT(OUT) :: ierror                       36
                                                                37
MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank, target_disp,
        target_count, target_datatype, win, request, ierror) 38
                                                                39
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: origin_addr          40
INTEGER, INTENT(IN) :: origin_count, target_rank, target_count 41
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype 42
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp     43
TYPE(MPI_Win), INTENT(IN) :: win                               44
TYPE(MPI_Request), INTENT(OUT) :: request                     45
INTEGER, OPTIONAL, INTENT(OUT) :: ierror                       46
                                                                47
                                                                48

```

```

1 MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank, target_disp,
2         target_count, target_datatype, win, request, ierror) !(_c)
3     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: origin_addr
4     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
5     TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
6     INTEGER, INTENT(IN) :: target_rank
7     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
8     TYPE(MPI_Win), INTENT(IN) :: win
9     TYPE(MPI_Request), INTENT(OUT) :: request
10    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12 MPI_Rget_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
13                    result_count, result_datatype, target_rank, target_disp,
14                    target_count, target_datatype, op, win, request, ierror)
15    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
16    INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
17                    target_count
18    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
19                    target_datatype
20    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: result_addr
21    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
22    TYPE(MPI_Op), INTENT(IN) :: op
23    TYPE(MPI_Win), INTENT(IN) :: win
24    TYPE(MPI_Request), INTENT(OUT) :: request
25    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27 MPI_Rget_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
28                    result_count, result_datatype, target_rank, target_disp,
29                    target_count, target_datatype, op, win, request, ierror) !(_c)
30    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
31    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, result_count,
32                    target_count
33    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
34                    target_datatype
35    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: result_addr
36    INTEGER, INTENT(IN) :: target_rank
37    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
38    TYPE(MPI_Op), INTENT(IN) :: op
39    TYPE(MPI_Win), INTENT(IN) :: win
40    TYPE(MPI_Request), INTENT(OUT) :: request
41    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43 MPI_Rput(origin_addr, origin_count, origin_datatype, target_rank, target_disp,
44          target_count, target_datatype, win, request, ierror)
45    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
46    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
47    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
48    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
49    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) !(_c)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
INTEGER, INTENT(IN) :: target_rank
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(size, disp_unit, info, comm, baseptr, win, ierror) !(_c)
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size, 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_allocate_shared(size, disp_unit, info, comm, baseptr, win, ierror)
         !(_c)
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size, disp_unit
TYPE(MPI_Info), INTENT(IN) :: info
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(C_PTR), INTENT(OUT) :: baseptr

```

```

1     TYPE(MPI_Win), INTENT(OUT) :: win
2     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
4 MPI_Win_attach(win, base, size, ierror)
5     TYPE(MPI_Win), INTENT(IN) :: win
6     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: base
7     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
8     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10 MPI_Win_complete(win, ierror)
11     TYPE(MPI_Win), INTENT(IN) :: win
12     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14 MPI_Win_create(base, size, disp_unit, info, comm, win, ierror)
15     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: base
16     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
17     INTEGER, INTENT(IN) :: disp_unit
18     TYPE(MPI_Info), INTENT(IN) :: info
19     TYPE(MPI_Comm), INTENT(IN) :: comm
20     TYPE(MPI_Win), INTENT(OUT) :: win
21     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23 MPI_Win_create(base, size, disp_unit, info, comm, win, ierror) !(_c)
24     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: base
25     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size, disp_unit
26     TYPE(MPI_Info), INTENT(IN) :: info
27     TYPE(MPI_Comm), INTENT(IN) :: comm
28     TYPE(MPI_Win), INTENT(OUT) :: win
29     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31 MPI_Win_create_dynamic(info, comm, win, ierror)
32     TYPE(MPI_Info), INTENT(IN) :: info
33     TYPE(MPI_Comm), INTENT(IN) :: comm
34     TYPE(MPI_Win), INTENT(OUT) :: win
35     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37 MPI_Win_detach(win, base, ierror)
38     TYPE(MPI_Win), INTENT(IN) :: win
39     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: base
40     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42 MPI_Win_fence(assert, win, ierror)
43     INTEGER, INTENT(IN) :: assert
44     TYPE(MPI_Win), INTENT(IN) :: win
45     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47 MPI_Win_flush(rank, win, ierror)
48     INTEGER, INTENT(IN) :: rank
49     TYPE(MPI_Win), INTENT(IN) :: win
50     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
51
52 MPI_Win_flush_all(win, ierror)

```

TYPE(MPI_Win), INTENT(IN) :: win	1
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	2
	3
MPI_Win_flush_local(rank, win, ierror)	4
INTEGER, INTENT(IN) :: rank	5
TYPE(MPI_Win), INTENT(IN) :: win	6
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	7
	8
MPI_Win_flush_local_all(win, ierror)	9
TYPE(MPI_Win), INTENT(IN) :: win	10
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	11
	12
MPI_Win_free(win, ierror)	13
TYPE(MPI_Win), INTENT(INOUT) :: win	14
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	15
	16
MPI_Win_get_group(win, group, ierror)	17
TYPE(MPI_Win), INTENT(IN) :: win	18
TYPE(MPI_Group), INTENT(OUT) :: group	19
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	20
	21
MPI_Win_get_info(win, info_used, ierror)	22
TYPE(MPI_Win), INTENT(IN) :: win	23
TYPE(MPI_Info), INTENT(OUT) :: info_used	24
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	25
	26
MPI_Win_lock(lock_type, rank, assert, win, ierror)	27
INTEGER, INTENT(IN) :: lock_type, rank, assert	28
TYPE(MPI_Win), INTENT(IN) :: win	29
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	30
	31
MPI_Win_lock_all(assert, win, ierror)	32
INTEGER, INTENT(IN) :: assert	33
TYPE(MPI_Win), INTENT(IN) :: win	34
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	35
	36
MPI_Win_post(group, assert, win, ierror)	37
TYPE(MPI_Group), INTENT(IN) :: group	38
INTEGER, INTENT(IN) :: assert	39
TYPE(MPI_Win), INTENT(IN) :: win	40
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	41
	42
MPI_Win_set_info(win, info, ierror)	43
TYPE(MPI_Win), INTENT(IN) :: win	44
TYPE(MPI_Info), INTENT(IN) :: info	45
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	46
	47
MPI_Win_shared_query(win, rank, size, disp_unit, baseptr, ierror)	48
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	

```

1      INTEGER, INTENT(OUT) :: disp_unit
2      TYPE(C_PTR), INTENT(OUT) :: baseptr
3      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
5      MPI_Win_shared_query(win, rank, size, disp_unit, baseptr, ierror) !(_c)
6      USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
7      TYPE(MPI_Win), INTENT(IN) :: win
8      INTEGER, INTENT(IN) :: rank
9      INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size, disp_unit
10     TYPE(C_PTR), INTENT(OUT) :: baseptr
11     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13     MPI_Win_start(group, assert, win, ierror)
14     TYPE(MPI_Group), INTENT(IN) :: group
15     INTEGER, INTENT(IN) :: assert
16     TYPE(MPI_Win), INTENT(IN) :: win
17     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19     MPI_Win_sync(win, ierror)
20     TYPE(MPI_Win), INTENT(IN) :: win
21     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23     MPI_Win_test(win, flag, ierror)
24     TYPE(MPI_Win), INTENT(IN) :: win
25     LOGICAL, INTENT(OUT) :: flag
26     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28     MPI_Win_unlock(rank, win, ierror)
29     INTEGER, INTENT(IN) :: rank
30     TYPE(MPI_Win), INTENT(IN) :: win
31     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33     MPI_Win_unlock_all(win, ierror)
34     TYPE(MPI_Win), INTENT(IN) :: win
35     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37     MPI_Win_wait(win, ierror)
38     TYPE(MPI_Win), INTENT(IN) :: win
39     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41     A.4.11 External Interfaces Fortran 2008 Bindings
42
43     MPI_Grequest_complete(request, ierror)
44     TYPE(MPI_Request), INTENT(IN) :: request
45     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47     MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request, ierror)
48     PROCEDURE(MPI_Grequest_query_function) :: query_fn
49     PROCEDURE(MPI_Grequest_free_function) :: free_fn
50     PROCEDURE(MPI_Grequest_cancel_function) :: cancel_fn
51     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state

```

```

TYPE(MPI_Request), INTENT(OUT) :: request      1
INTEGER, OPTIONAL, INTENT(OUT) :: ierror      2
MPI_Status_set_cancelled(status, flag, ierror) 3
TYPE(MPI_Status), INTENT(INOUT) :: status     4
LOGICAL, INTENT(IN) :: flag                   5
INTEGER, OPTIONAL, INTENT(OUT) :: ierror      6
MPI_Status_set_elements(status, datatype, count, ierror) 7
TYPE(MPI_Status), INTENT(INOUT) :: status     8
TYPE(MPI_Datatype), INTENT(IN) :: datatype    9
INTEGER, INTENT(IN) :: count                 10
INTEGER, OPTIONAL, INTENT(OUT) :: ierror     11
MPI_Status_set_elements(status, datatype, count, ierror) !(_c) 12
TYPE(MPI_Status), INTENT(INOUT) :: status     13
TYPE(MPI_Datatype), INTENT(IN) :: datatype    14
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count 15
INTEGER, OPTIONAL, INTENT(OUT) :: ierror     16
MPI_Status_set_elements_x(status, datatype, count, ierror) 17
TYPE(MPI_Status), INTENT(INOUT) :: status     18
TYPE(MPI_Datatype), INTENT(IN) :: datatype    19
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count 20
INTEGER, OPTIONAL, INTENT(OUT) :: ierror     21
A.4.12 I/O Fortran 2008 Bindings 22
MPI_CONVERSION_FN_NULL(userbuf, datatype, count, filebuf, position, 23
    extra_state, ierror) 24
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR 25
TYPE(C_PTR), VALUE :: userbuf, filebuf 26
TYPE(MPI_Datatype) :: datatype 27
INTEGER :: count, ierror 28
INTEGER(KIND=MPI_OFFSET_KIND) :: position 29
INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state 30
MPI_CONVERSION_FN_NULL_C(userbuf, datatype, count, filebuf, position, 31
    extra_state, ierror) !(_c) 32
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR 33
TYPE(C_PTR), VALUE :: userbuf, filebuf 34
TYPE(MPI_Datatype) :: datatype 35
INTEGER(KIND=MPI_COUNT_KIND) :: count 36
INTEGER(KIND=MPI_OFFSET_KIND) :: position 37
INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state 38
INTEGER :: ierror 39
MPI_File_close(fh, ierror) 40
TYPE(MPI_File), INTENT(INOUT) :: fh 41
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 42

```

```
1 MPI_File_delete(filename, info, ierror)
2   CHARACTER(LEN=*), INTENT(IN) :: filename
3   TYPE(MPI_Info), INTENT(IN) :: info
4   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
6 MPI_File_get_amode(fh, amode, ierror)
7   TYPE(MPI_File), INTENT(IN) :: fh
8   INTEGER, INTENT(OUT) :: amode
9   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11 MPI_File_get_atomicity(fh, flag, ierror)
12   TYPE(MPI_File), INTENT(IN) :: fh
13   LOGICAL, INTENT(OUT) :: flag
14   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16 MPI_File_get_byte_offset(fh, offset, disp, ierror)
17   TYPE(MPI_File), INTENT(IN) :: fh
18   INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
19   INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp
20   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22 MPI_File_get_group(fh, group, ierror)
23   TYPE(MPI_File), INTENT(IN) :: fh
24   TYPE(MPI_Group), INTENT(OUT) :: group
25   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27 MPI_File_get_info(fh, info_used, ierror)
28   TYPE(MPI_File), INTENT(IN) :: fh
29   TYPE(MPI_Info), INTENT(OUT) :: info_used
30   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32 MPI_File_get_position(fh, offset, ierror)
33   TYPE(MPI_File), INTENT(IN) :: fh
34   INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset
35   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37 MPI_File_get_position_shared(fh, offset, ierror)
38   TYPE(MPI_File), INTENT(IN) :: fh
39   INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset
40   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42 MPI_File_get_size(fh, size, ierror)
43   TYPE(MPI_File), INTENT(IN) :: fh
44   INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: size
45   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47 MPI_File_get_type_extent(fh, datatype, extent, ierror)
48   TYPE(MPI_File), INTENT(IN) :: fh
49   TYPE(MPI_Datatype), INTENT(IN) :: datatype
50   INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: extent
51   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```



```

MPI_File_get_type_extent(fh, datatype, extent, ierror) !(_c)           1
  TYPE(MPI_File), INTENT(IN) :: fh                                   2
  TYPE(MPI_Datatype), INTENT(IN) :: datatype                       3
  INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: extent             4
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror                         5
                                                                    6
MPI_File_get_view(fh, disp, etype, filetype, datarep, ierror)       7
  TYPE(MPI_File), INTENT(IN) :: fh                                 8
  INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp              9
  TYPE(MPI_Datatype), INTENT(OUT) :: etype, filetype             10
  CHARACTER(LEN=*), INTENT(OUT) :: datarep                       11
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror                       12
                                                                    13
MPI_File_iread(fh, buf, count, datatype, request, ierror)          14
  TYPE(MPI_File), INTENT(IN) :: fh                                 15
  TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf                    16
  INTEGER, INTENT(IN) :: count                                     17
  TYPE(MPI_Datatype), INTENT(IN) :: datatype                       18
  TYPE(MPI_Request), INTENT(OUT) :: request                       19
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror                       20
                                                                    21
MPI_File_iread(fh, buf, count, datatype, request, ierror) !(_c)   22
  TYPE(MPI_File), INTENT(IN) :: fh                                 23
  TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf                    24
  INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count              25
  TYPE(MPI_Datatype), INTENT(IN) :: datatype                       26
  TYPE(MPI_Request), INTENT(OUT) :: request                       27
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror                       28
                                                                    29
MPI_File_iread_all(fh, buf, count, datatype, request, ierror)     30
  TYPE(MPI_File), INTENT(IN) :: fh                                 31
  TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf                    32
  INTEGER, INTENT(IN) :: count                                     33
  TYPE(MPI_Datatype), INTENT(IN) :: datatype                       34
  TYPE(MPI_Request), INTENT(OUT) :: request                       35
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror                       36
                                                                    37
MPI_File_iread_all(fh, buf, count, datatype, request, ierror) !(_c) 38
  TYPE(MPI_File), INTENT(IN) :: fh                                 39
  TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf                    40
  INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count              41
  TYPE(MPI_Datatype), INTENT(IN) :: datatype                       42
  TYPE(MPI_Request), INTENT(OUT) :: request                       43
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror                       44
                                                                    45
MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror) 46
  TYPE(MPI_File), INTENT(IN) :: fh                                 47
  INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset            48
  TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf                    49
  INTEGER, INTENT(IN) :: count                                     50

```

```

1     TYPE(MPI_Datatype), INTENT(IN) :: datatype
2     TYPE(MPI_Request), INTENT(OUT) :: request
3     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
5 MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror) !(_c)
6     TYPE(MPI_File), INTENT(IN) :: fh
7     INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
8     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
9     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
10    TYPE(MPI_Datatype), INTENT(IN) :: datatype
11    TYPE(MPI_Request), INTENT(OUT) :: request
12    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14 MPI_File_iread_at_all(fh, offset, buf, count, datatype, request, ierror)
15    TYPE(MPI_File), INTENT(IN) :: fh
16    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
17    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
18    INTEGER, INTENT(IN) :: count
19    TYPE(MPI_Datatype), INTENT(IN) :: datatype
20    TYPE(MPI_Request), INTENT(OUT) :: request
21    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23 MPI_File_iread_at_all(fh, offset, buf, count, datatype, request, ierror) !(_c)
24    TYPE(MPI_File), INTENT(IN) :: fh
25    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
26    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
27    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
28    TYPE(MPI_Datatype), INTENT(IN) :: datatype
29    TYPE(MPI_Request), INTENT(OUT) :: request
30    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32 MPI_File_iread_shared(fh, buf, count, datatype, request, ierror)
33    TYPE(MPI_File), INTENT(IN) :: fh
34    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
35    INTEGER, INTENT(IN) :: count
36    TYPE(MPI_Datatype), INTENT(IN) :: datatype
37    TYPE(MPI_Request), INTENT(OUT) :: request
38    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40 MPI_File_iread_shared(fh, buf, count, datatype, request, ierror) !(_c)
41    TYPE(MPI_File), INTENT(IN) :: fh
42    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
43    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
44    TYPE(MPI_Datatype), INTENT(IN) :: datatype
45    TYPE(MPI_Request), INTENT(OUT) :: request
46    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48 MPI_File_iwrite(fh, buf, count, datatype, request, ierror)
49    TYPE(MPI_File), INTENT(IN) :: fh
50    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_iread(fh, buf, count, datatype, request, ierror) !(_c)
TYPE(MPI_File), INTENT(IN) :: fh
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
INTEGER(KIND=MPI_COUNT_KIND), 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(..), 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_iread_all(fh, buf, count, datatype, request, ierror) !(_c)
TYPE(MPI_File), INTENT(IN) :: fh
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
INTEGER(KIND=MPI_COUNT_KIND), 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(..), 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_iread_at(fh, offset, buf, count, datatype, request, ierror) !(_c)
TYPE(MPI_File), INTENT(IN) :: fh
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
INTEGER(KIND=MPI_COUNT_KIND), 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

```

```

1     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
2     INTEGER, INTENT(IN) :: count
3     TYPE(MPI_Datatype), INTENT(IN) :: datatype
4     TYPE(MPI_Request), INTENT(OUT) :: request
5     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
7     MPI_File_iwrite_at_all(fh, offset, buf, count, datatype, request, ierror) !(_c)
8     TYPE(MPI_File), INTENT(IN) :: fh
9     INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
10    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
11    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
12    TYPE(MPI_Datatype), INTENT(IN) :: datatype
13    TYPE(MPI_Request), INTENT(OUT) :: request
14    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16    MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror)
17    TYPE(MPI_File), INTENT(IN) :: fh
18    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
19    INTEGER, INTENT(IN) :: count
20    TYPE(MPI_Datatype), INTENT(IN) :: datatype
21    TYPE(MPI_Request), INTENT(OUT) :: request
22    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
24    MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror) !(_c)
25    TYPE(MPI_File), INTENT(IN) :: fh
26    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
27    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
28    TYPE(MPI_Datatype), INTENT(IN) :: datatype
29    TYPE(MPI_Request), INTENT(OUT) :: request
30    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32    MPI_File_open(comm, filename, amode, info, fh, ierror)
33    TYPE(MPI_Comm), INTENT(IN) :: comm
34    CHARACTER(LEN=*), INTENT(IN) :: filename
35    INTEGER, INTENT(IN) :: amode
36    TYPE(MPI_Info), INTENT(IN) :: info
37    TYPE(MPI_File), INTENT(OUT) :: fh
38    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40    MPI_File_preallocate(fh, size, ierror)
41    TYPE(MPI_File), INTENT(IN) :: fh
42    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size
43    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45    MPI_File_read(fh, buf, count, datatype, status, ierror)
46    TYPE(MPI_File), INTENT(IN) :: fh
47    TYPE(*), DIMENSION(..) :: buf
48    INTEGER, INTENT(IN) :: count
49    TYPE(MPI_Datatype), INTENT(IN) :: datatype
50    TYPE(MPI_Status) :: status

```

```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1
MPI_File_read(fh, buf, count, datatype, status, ierror) !(_c) 2
TYPE(MPI_File), INTENT(IN) :: fh 3
TYPE(*), DIMENSION(..) :: buf 4
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count 5
TYPE(MPI_Datatype), INTENT(IN) :: datatype 6
TYPE(MPI_Status) :: status 7
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 8
MPI_File_read_all(fh, buf, count, datatype, status, ierror) 9
TYPE(MPI_File), INTENT(IN) :: fh 10
TYPE(*), DIMENSION(..) :: buf 11
INTEGER, INTENT(IN) :: count 12
TYPE(MPI_Datatype), INTENT(IN) :: datatype 13
TYPE(MPI_Status) :: status 14
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 15
MPI_File_read_all(fh, buf, count, datatype, status, ierror) !(_c) 16
TYPE(MPI_File), INTENT(IN) :: fh 17
TYPE(*), DIMENSION(..) :: buf 18
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count 19
TYPE(MPI_Datatype), INTENT(IN) :: datatype 20
TYPE(MPI_Status) :: status 21
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22
MPI_File_read_all_begin(fh, buf, count, datatype, ierror) 23
TYPE(MPI_File), INTENT(IN) :: fh 24
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf 25
INTEGER, INTENT(IN) :: count 26
TYPE(MPI_Datatype), INTENT(IN) :: datatype 27
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 28
MPI_File_read_all_begin(fh, buf, count, datatype, ierror) !(_c) 29
TYPE(MPI_File), INTENT(IN) :: fh 30
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf 31
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count 32
TYPE(MPI_Datatype), INTENT(IN) :: datatype 33
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 34
MPI_File_read_all_end(fh, buf, status, ierror) 35
TYPE(MPI_File), INTENT(IN) :: fh 36
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf 37
TYPE(MPI_Status) :: status 38
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 39
MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror) 40
TYPE(MPI_File), INTENT(IN) :: fh 41
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset 42
TYPE(*), DIMENSION(..) :: buf 43
INTEGER, INTENT(IN) :: count 44

```

```

1      TYPE(MPI_Datatype), INTENT(IN) :: datatype
2      TYPE(MPI_Status) :: status
3      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
5      MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror) !(_c)
6      TYPE(MPI_File), INTENT(IN) :: fh
7      INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
8      TYPE(*), DIMENSION(..) :: buf
9      INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
10     TYPE(MPI_Datatype), INTENT(IN) :: datatype
11     TYPE(MPI_Status) :: status
12     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14     MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror)
15     TYPE(MPI_File), INTENT(IN) :: fh
16     INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
17     TYPE(*), DIMENSION(..) :: buf
18     INTEGER, INTENT(IN) :: count
19     TYPE(MPI_Datatype), INTENT(IN) :: datatype
20     TYPE(MPI_Status) :: status
21     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23     MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror) !(_c)
24     TYPE(MPI_File), INTENT(IN) :: fh
25     INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
26     TYPE(*), DIMENSION(..) :: buf
27     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
28     TYPE(MPI_Datatype), INTENT(IN) :: datatype
29     TYPE(MPI_Status) :: status
30     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32     MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror)
33     TYPE(MPI_File), INTENT(IN) :: fh
34     INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
35     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
36     INTEGER, INTENT(IN) :: count
37     TYPE(MPI_Datatype), INTENT(IN) :: datatype
38     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40     MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror) !(_c)
41     TYPE(MPI_File), INTENT(IN) :: fh
42     INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
43     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
44     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
45     TYPE(MPI_Datatype), INTENT(IN) :: datatype
46     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48     MPI_File_read_at_all_end(fh, buf, status, ierror)
49     TYPE(MPI_File), INTENT(IN) :: fh
50     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(fh, buf, count, datatype, status, ierror) !(_c)
TYPE(MPI_File), INTENT(IN) :: fh
TYPE(*), DIMENSION(..) :: buf
INTEGER(KIND=MPI_COUNT_KIND), 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_begin(fh, buf, count, datatype, ierror) !(_c)
TYPE(MPI_File), INTENT(IN) :: fh
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
INTEGER(KIND=MPI_COUNT_KIND), 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_read_shared(fh, buf, count, datatype, status, ierror) !(_c)
TYPE(MPI_File), INTENT(IN) :: fh
TYPE(*), DIMENSION(..) :: buf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count

```

```
1     TYPE(MPI_Datatype), INTENT(IN) :: datatype
2     TYPE(MPI_Status) :: status
3     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
5 MPI_File_seek(fh, offset, whence, ierror)
6     TYPE(MPI_File), INTENT(IN) :: fh
7     INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
8     INTEGER, INTENT(IN) :: whence
9     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11 MPI_File_seek_shared(fh, offset, whence, ierror)
12     TYPE(MPI_File), INTENT(IN) :: fh
13     INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
14     INTEGER, INTENT(IN) :: whence
15     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17 MPI_File_set_atomicity(fh, flag, ierror)
18     TYPE(MPI_File), INTENT(IN) :: fh
19     LOGICAL, INTENT(IN) :: flag
20     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22 MPI_File_set_info(fh, info, ierror)
23     TYPE(MPI_File), INTENT(IN) :: fh
24     TYPE(MPI_Info), INTENT(IN) :: info
25     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27 MPI_File_set_size(fh, size, ierror)
28     TYPE(MPI_File), INTENT(IN) :: fh
29     INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size
30     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32 MPI_File_set_view(fh, disp, etype, filetype, datarep, info, ierror)
33     TYPE(MPI_File), INTENT(IN) :: fh
34     INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: disp
35     TYPE(MPI_Datatype), INTENT(IN) :: etype, filetype
36     CHARACTER(LEN=*), INTENT(IN) :: datarep
37     TYPE(MPI_Info), INTENT(IN) :: info
38     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40 MPI_File_sync(fh, ierror)
41     TYPE(MPI_File), INTENT(IN) :: fh
42     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44 MPI_File_write(fh, buf, count, datatype, status, ierror)
45     TYPE(MPI_File), INTENT(IN) :: fh
46     TYPE(*), DIMENSION(..), INTENT(IN) :: buf
47     INTEGER, INTENT(IN) :: count
48     TYPE(MPI_Datatype), INTENT(IN) :: datatype
49     TYPE(MPI_Status) :: status
50     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
51
52 MPI_File_write(fh, buf, count, datatype, status, ierror) !(_c)
```



```

TYPE(MPI_File), INTENT(IN) :: fh
TYPE(*), DIMENSION(..), INTENT(IN) :: buf
INTEGER(KIND=MPI_COUNT_KIND), 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(fh, buf, count, datatype, status, ierror) !(_c)
TYPE(MPI_File), INTENT(IN) :: fh
TYPE(*), DIMENSION(..), INTENT(IN) :: buf
INTEGER(KIND=MPI_COUNT_KIND), 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_begin(fh, buf, count, datatype, ierror) !(_c)
TYPE(MPI_File), INTENT(IN) :: fh
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
INTEGER(KIND=MPI_COUNT_KIND), 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

```

```

1     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
2
3 MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror) !(_c)
4     TYPE(MPI_File), INTENT(IN) :: fh
5     INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
6     TYPE(*), DIMENSION(..), INTENT(IN) :: buf
7     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
8     TYPE(MPI_Datatype), INTENT(IN) :: datatype
9     TYPE(MPI_Status) :: status
10    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12 MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)
13    TYPE(MPI_File), INTENT(IN) :: fh
14    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
15    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
16    INTEGER, INTENT(IN) :: count
17    TYPE(MPI_Datatype), INTENT(IN) :: datatype
18    TYPE(MPI_Status) :: status
19    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21 MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror) !(_c)
22    TYPE(MPI_File), INTENT(IN) :: fh
23    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
24    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
25    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
26    TYPE(MPI_Datatype), INTENT(IN) :: datatype
27    TYPE(MPI_Status) :: status
28    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30 MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror)
31    TYPE(MPI_File), INTENT(IN) :: fh
32    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
33    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
34    INTEGER, INTENT(IN) :: count
35    TYPE(MPI_Datatype), INTENT(IN) :: datatype
36    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38 MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror) !(_c)
39    TYPE(MPI_File), INTENT(IN) :: fh
40    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
41    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
42    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
43    TYPE(MPI_Datatype), INTENT(IN) :: datatype
44    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46 MPI_File_write_at_all_end(fh, buf, status, ierror)
47    TYPE(MPI_File), INTENT(IN) :: fh
48    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
49    TYPE(MPI_Status) :: status
50    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

MPI_File_write_ordered(fh, buf, count, datatype, status, ierror) 1
    TYPE(MPI_File), INTENT(IN) :: fh 2
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf 3
    INTEGER, INTENT(IN) :: count 4
    TYPE(MPI_Datatype), INTENT(IN) :: datatype 5
    TYPE(MPI_Status) :: status 6
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror 7
8
MPI_File_write_ordered(fh, buf, count, datatype, status, ierror) !(_c) 9
    TYPE(MPI_File), INTENT(IN) :: fh 10
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf 11
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count 12
    TYPE(MPI_Datatype), INTENT(IN) :: datatype 13
    TYPE(MPI_Status) :: status 14
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror 15
16
MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror) 17
    TYPE(MPI_File), INTENT(IN) :: fh 18
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 19
    INTEGER, INTENT(IN) :: count 20
    TYPE(MPI_Datatype), INTENT(IN) :: datatype 21
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22
23
MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror) !(_c) 24
    TYPE(MPI_File), INTENT(IN) :: fh 25
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 26
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count 27
    TYPE(MPI_Datatype), INTENT(IN) :: datatype 28
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror 29
30
MPI_File_write_ordered_end(fh, buf, status, ierror) 31
    TYPE(MPI_File), INTENT(IN) :: fh 32
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 33
    TYPE(MPI_Status) :: status 34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror 35
36
MPI_File_write_shared(fh, buf, count, datatype, status, ierror) 37
    TYPE(MPI_File), INTENT(IN) :: fh 38
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf 39
    INTEGER, INTENT(IN) :: count 40
    TYPE(MPI_Datatype), INTENT(IN) :: datatype 41
    TYPE(MPI_Status) :: status 42
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror 43
44
MPI_File_write_shared(fh, buf, count, datatype, status, ierror) !(_c) 45
    TYPE(MPI_File), INTENT(IN) :: fh 46
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf 47
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count 48
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status

```

```

1      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
2
3      MPI_Register_datarep(datarep, read_conversion_fn, write_conversion_fn,
4          dtype_file_extent_fn, extra_state, ierror)
5          CHARACTER(LEN=*), INTENT(IN) :: datarep
6          PROCEDURE(MPI_Datarep_conversion_function) :: read_conversion_fn,
7              write_conversion_fn
8          PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn
9          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
10         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12     MPI_Register_datarep_c(datarep, read_conversion_fn, write_conversion_fn,
13         dtype_file_extent_fn, extra_state, ierror) !(_c)
14         CHARACTER(LEN=*), INTENT(IN) :: datarep
15         PROCEDURE(MPI_Datarep_conversion_function_c) :: read_conversion_fn,
16             write_conversion_fn
17         PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn
18         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
19         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21     A.4.13 Language Bindings Fortran 2008 Bindings
22
23     MPI_F_sync_reg(buf)
24         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
25
26     MPI_Status_f082f(f08_status, f_status, ierror)
27         TYPE(MPI_Status), INTENT(IN) :: f08_status
28         INTEGER, INTENT(OUT) :: f_status(MPI_STATUS_SIZE)
29         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31     MPI_Status_f2f08(f_status, f08_status, ierror)
32         INTEGER, INTENT(IN) :: f_status(MPI_STATUS_SIZE)
33         TYPE(MPI_Status), INTENT(OUT) :: f08_status
34         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36     MPI_Type_create_f90_complex(p, r, newtype, ierror)
37         INTEGER, INTENT(IN) :: p, r
38         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
39         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41     MPI_Type_create_f90_integer(r, newtype, ierror)
42         INTEGER, INTENT(IN) :: r
43         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
44         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46     MPI_Type_create_f90_real(p, r, newtype, ierror)
47         INTEGER, INTENT(IN) :: p, r
48         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
49         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
50
51     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.4.14 Tools / Profiling Interface Fortran 2008 Bindings

```

MPI_Pcontrol(level)
  INTEGER, INTENT(IN) :: level

```

A.4.15 Deprecated Fortran 2008 Bindings

```

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=valuelen), INTENT(OUT) :: value
  LOGICAL, INTENT(OUT) :: flag
  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_Sizeof(x, size, ierror)
  TYPE(*), DIMENSION(..) :: x
  INTEGER, INTENT(OUT) :: size
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

A.5 Fortran Bindings with `mpif.h` or the `mpi` Module

A.5.1 Point-to-Point Communication Fortran Bindings

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

<type> BUF(*)

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

MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)

<type> BUF(*)

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

MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR)

<type> BUFFER(*)

INTEGER SIZE, IERROR

MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)

<type> BUFFER_ADDR(*)

INTEGER SIZE, IERROR

MPI_CANCEL(REQUEST, IERROR)

INTEGER REQUEST, IERROR

MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)

INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR

MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)

<type> BUF(*)

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

MPI_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_ISENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF, REVCOUNT,	1
RECVTYPE, SOURCE, RECVTAG, COMM, REQUEST, IERROR)	2
<type> SENDBUF(*), RECVBUF(*)	3
INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, REVCOUNT, RECVTYPE, SOURCE,	4
RECVTAG, COMM, REQUEST, IERROR	5
	6
MPI_ISENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,	7
COMM, REQUEST, IERROR)	8
<type> BUF(*)	9
INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, REQUEST,	10
IERROR	11
	12
MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	13
<type> BUF(*)	14
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR	15
	16
MPI_MPROBE(SOURCE, TAG, COMM, MESSAGE, STATUS, IERROR)	17
INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR	18
	19
MPI_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR)	20
<type> BUF(*)	21
INTEGER COUNT, DATATYPE, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR	22
	23
MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)	24
INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR	25
	26
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)	27
<type> BUF(*)	28
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR	29
	30
MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)	31
<type> BUF(*)	32
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR	33
	34
MPI_REQUEST_FREE(REQUEST, IERROR)	35
INTEGER REQUEST, IERROR	36
	37
MPI_REQUEST_GET_STATUS(REQUEST, FLAG, STATUS, IERROR)	38
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR	39
LOGICAL FLAG	40
	41
MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)	42
<type> BUF(*)	43
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR	44
	45
MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	46
<type> BUF(*)	47
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR	48
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)	
<type> BUF(*)	
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR	
MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	

```

1     <type> BUF(*)
2     INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
3
4     MPI_SENDBUF(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF, REVCOUNT,
5               RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
6     <type> SENDBUF(*), RECVBUF(*)
7     INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, REVCOUNT, RECVTYPE, SOURCE,
8               RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
9
10    MPI_SENDBUF_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
11                      COMM, STATUS, IERROR)
12    <type> BUF(*)
13    INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM,
14          STATUS(MPI_STATUS_SIZE), IERROR
15
16    MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
17    <type> BUF(*)
18    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
19
20    MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
21    <type> BUF(*)
22    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
23
24    MPI_START(REQUEST, IERROR)
25    INTEGER REQUEST, IERROR
26
27    MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)
28    INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR
29
30    MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
31    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
32    LOGICAL FLAG
33
34    MPI_TEST_CANCELLED(STATUS, FLAG, IERROR)
35    INTEGER STATUS(MPI_STATUS_SIZE), IERROR
36    LOGICAL FLAG
37
38    MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR)
39    INTEGER COUNT, ARRAY_OF_REQUESTS(*), ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *),
40          IERROR
41    LOGICAL FLAG
42
43    MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)
44    INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE), IERROR
45    LOGICAL FLAG
46
47    MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
48               ARRAY_OF_STATUSES, IERROR)
49    INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
50          ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *), IERROR
51
52    MPI_WAIT(REQUEST, STATUS, IERROR)
53    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

```



```

MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)      1
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *),  2
    IERROR                                                                3
                                                                4
MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)         5
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE), IERROR  6
                                                                7
MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,  8
    ARRAY_OF_STATUSES, IERROR)                                         9
    INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),     10
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *), IERROR                         11

```

A.5.2 Partitioned Communication Fortran Bindings

```

MPI_PARRIVED(REQUEST, PARTITION, FLAG, IERROR)                        12
    INTEGER REQUEST, PARTITION, IERROR                                  13
    LOGICAL FLAG                                                       14
                                                                15
MPI_PREADY(PARTITION, REQUEST, IERROR)                                16
    INTEGER PARTITION, REQUEST, IERROR                                  17
                                                                18
MPI_PREADY_LIST(LENGTH, ARRAY_OF_PARTITIONS, REQUEST, IERROR)        19
    INTEGER LENGTH, ARRAY_OF_PARTITIONS(*), REQUEST, IERROR           20
                                                                21
MPI_PREADY_RANGE(PARTITION_LOW, PARTITION_HIGH, REQUEST, IERROR)     22
    INTEGER PARTITION_LOW, PARTITION_HIGH, REQUEST, IERROR            23
                                                                24
MPI_PRECV_INIT(BUF, PARTITIONS, COUNT, DATATYPE, SOURCE, TAG, COMM, INFO,  25
    REQUEST, IERROR)                                                   26
    <type> BUF(*)                                                       27
    INTEGER PARTITIONS, DATATYPE, SOURCE, TAG, COMM, INFO, REQUEST, IERROR  28
    INTEGER(KIND=MPI_COUNT_KIND) COUNT                                  29
                                                                30
MPI_PSEND_INIT(BUF, PARTITIONS, COUNT, DATATYPE, DEST, TAG, COMM, INFO,  31
    REQUEST, IERROR)                                                   32
    <type> BUF(*)                                                       33
    INTEGER PARTITIONS, DATATYPE, DEST, TAG, COMM, INFO, REQUEST, IERROR  34
    INTEGER(KIND=MPI_COUNT_KIND) COUNT                                  35

```

A.5.3 Datatypes Fortran Bindings

```

INTEGER(KIND=MPI_ADDRESS_KIND) MPI_AINT_ADD(BASE, DISP)               36
    INTEGER(KIND=MPI_ADDRESS_KIND) BASE, DISP                          37
                                                                38
INTEGER(KIND=MPI_ADDRESS_KIND) MPI_AINT_DIFF(ADDR1, ADDR2)           39
    INTEGER(KIND=MPI_ADDRESS_KIND) ADDR1, ADDR2                        40
                                                                41
MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)                              42
    <type> LOCATION(*)                                                 43
    INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS                             44
    INTEGER IERROR                                                      45

```

```

1 MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
2     INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
3
4 MPI_GET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)
5     INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR
6     INTEGER(KIND=MPI_COUNT_KIND) COUNT
7
8 MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)
9     <type> INBUF(*), OUTBUF(*)
10    INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR
11
12 MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION,
13     IERROR)
14    CHARACTER*(*) DATAREP
15    <type> INBUF(*), OUTBUF(*)
16    INTEGER INCOUNT, DATATYPE, IERROR
17    INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, POSITION
18
19 MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR)
20    CHARACTER*(*) DATAREP
21    INTEGER INCOUNT, DATATYPE, IERROR
22    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
23
24 MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)
25    INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR
26
27 MPI_TYPE_COMMIT(DATATYPE, IERROR)
28    INTEGER DATATYPE, IERROR
29
30 MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
31    INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
32
33 MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS, ARRAY_OF_GSIZES, ARRAY_OF_DISTRIBS,
34     ARRAY_OF_DARGS, ARRAY_OF_PSIZEs, ORDER, OLDTYPE, NEWTYPE, IERROR)
35    INTEGER SIZE, RANK, NDIMS, ARRAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*),
36     ARRAY_OF_DARGS(*), ARRAY_OF_PSIZEs(*), ORDER, OLDTYPE, NEWTYPE,
37     IERROR
38
39 MPI_TYPE_CREATE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
40     OLDTYPE, NEWTYPE, IERROR)
41    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), OLDTYPE, NEWTYPE, IERROR
42    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
43
44 MPI_TYPE_CREATE_HINDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,
45     OLDTYPE, NEWTYPE, IERROR)
46    INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR
47    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
48
49 MPI_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
50    INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR
51    INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE

```

```

MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,      1
                              OLDTYPE, NEWTYPE, IERROR)                        2
    INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDTYPE, NEWTYPE,   3
    IERROR                                                                    4
MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)                5
    INTEGER OLDTYPE, NEWTYPE, IERROR                                           6
    INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT                                   7
MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,  9
                      ARRAY_OF_TYPES, NEWTYPE, IERROR)                       10
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_TYPES(*), NEWTYPE, IERROR 11
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)                 12
MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES, ARRAY_OF_SUBSIZES,           13
                        ARRAY_OF_STARTS, ORDER, OLDTYPE, NEWTYPE, IERROR)     14
    INTEGER NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*), ARRAY_OF_STARTS(*), 15
    ORDER, OLDTYPE, NEWTYPE, IERROR                                           16
MPI_TYPE_DUP(OLDTYPE, NEWTYPE, IERROR)                                        17
    INTEGER OLDTYPE, NEWTYPE, IERROR                                           18
MPI_TYPE_FREE(DATATYPE, IERROR)                                             19
    INTEGER DATATYPE, IERROR                                                   20
MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,   21
                    ARRAY_OF_INTEGERS, ARRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES, 22
                    IERROR)                                                   23
    INTEGER DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,             24
    ARRAY_OF_INTEGERS(*), ARRAY_OF_DATATYPES(*), IERROR                     25
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_ADDRESSES(*)                     26
MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES,   27
                    COMBINER, IERROR)                                         28
    INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER,   29
    IERROR                                                                    30
MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR)                           31
    INTEGER DATATYPE, IERROR                                                   32
    INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT                                   33
MPI_TYPE_GET_EXTENT_X(DATATYPE, LB, EXTENT, IERROR)                         34
    INTEGER DATATYPE, IERROR                                                   35
    INTEGER(KIND=MPI_COUNT_KIND) LB, EXTENT                                    36
MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)            37
    INTEGER DATATYPE, IERROR                                                   38
    INTEGER(KIND=MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT                       39
MPI_TYPE_GET_TRUE_EXTENT_X(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)          40
    INTEGER DATATYPE, IERROR                                                   41
    INTEGER(KIND=MPI_COUNT_KIND) TRUE_LB, TRUE_EXTENT                         42

```

```

1 MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS, OLDDTYPE,
2     NEWTYPE, IERROR)
3     INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
4     OLDDTYPE, NEWTYPE, IERROR
5
6 MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
7     INTEGER DATATYPE, SIZE, IERROR
8
9 MPI_TYPE_SIZE_X(DATATYPE, SIZE, IERROR)
10    INTEGER DATATYPE, IERROR
11    INTEGER(KIND=MPI_COUNT_KIND) SIZE
12
13 MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDDTYPE, NEWTYPE, IERROR)
14    INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDDTYPE, NEWTYPE, IERROR
15
16 MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM, IERROR)
17    <type> INBUF(*), OUTBUF(*)
18    INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR
19
20 MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT,
21    DATATYPE, IERROR)
22    CHARACTER*(*) DATAREP
23    <type> INBUF(*), OUTBUF(*)
24    INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION
25    INTEGER OUTCOUNT, DATATYPE, IERROR
26
27 A.5.4 Collective Communication Fortran Bindings
28
29 MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUFFER, RECVCOUNT, RECVTYPE, COMM,
30    IERROR)
31    <type> SENDBUF(*), RECVBUFFER(*)
32    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
33
34 MPI_ALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUFFER, RECVCOUNT, RECVTYPE,
35    COMM, INFO, REQUEST, IERROR)
36    <type> SENDBUF(*), RECVBUFFER(*)
37    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
38    IERROR
39
40 MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUFFER, RECVCOUNTS, DISPLS,
41    RECVTYPE, COMM, IERROR)
42    <type> SENDBUF(*), RECVBUFFER(*)
43    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
44    IERROR
45
46 MPI_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUFFER, RECVCOUNTS, DISPLS,
47    RECVTYPE, COMM, INFO, REQUEST, IERROR)
48    <type> SENDBUF(*), RECVBUFFER(*)
49    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
50    INFO, REQUEST, IERROR
51
52 MPI_ALLREDUCE(SENDBUF, RECVBUFFER, COUNT, DATATYPE, OP, COMM, IERROR)

```

<type> SENDBUF(*), RECVBUF(*)	1
INTEGER COUNT, DATATYPE, OP, COMM, IERROR	2
MPI_ALLREDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,	3
IERROR)	4
<type> SENDBUF(*), RECVBUF(*)	5
INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR	6
MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM,	7
IERROR)	8
<type> SENDBUF(*), RECVBUF(*)	9
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR	10
MPI_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,	11
COMM, INFO, REQUEST, IERROR)	12
<type> SENDBUF(*), RECVBUF(*)	13
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,	14
IERROR	15
MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,	16
RDISPLS, RECVTYPE, COMM, IERROR)	17
<type> SENDBUF(*), RECVBUF(*)	18
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),	19
RECVTYPE, COMM, IERROR	20
MPI_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,	21
RDISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)	22
<type> SENDBUF(*), RECVBUF(*)	23
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),	24
RECVTYPE, COMM, INFO, REQUEST, IERROR	25
MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS,	26
RDISPLS, RECVTYPES, COMM, IERROR)	27
<type> SENDBUF(*), RECVBUF(*)	28
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*), RDISPLS(*),	29
RECVTYPES(*), COMM, IERROR	30
MPI_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,	31
RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST, IERROR)	32
<type> SENDBUF(*), RECVBUF(*)	33
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*), RDISPLS(*),	34
RECVTYPES(*), COMM, INFO, REQUEST, IERROR	35
MPI_BARRIER(COMM, IERROR)	36
INTEGER COMM, IERROR	37
MPI_BARRIER_INIT(COMM, INFO, REQUEST, IERROR)	38
INTEGER COMM, INFO, REQUEST, IERROR	39
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)	40
<type> BUFFER(*)	41
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR	42
	43
	44
	45
	46
	47
	48

```
1 MPI_BCAST_INIT(BUFFER, COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR)
2   <type> BUFFER(*)
3   INTEGER COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR
4
5 MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
6   <type> SENDBUF(*), RECVBUF(*)
7   INTEGER COUNT, DATATYPE, OP, COMM, IERROR
8
9 MPI_EXSCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,
10  IERROR)
11  <type> SENDBUF(*), RECVBUF(*)
12  INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
13
14 MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT,
15  COMM, IERROR)
16  <type> SENDBUF(*), RECVBUF(*)
17  INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
18
19 MPI_GATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
20  ROOT, COMM, INFO, REQUEST, IERROR)
21  <type> SENDBUF(*), RECVBUF(*)
22  INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO,
23  REQUEST, IERROR
24
25 MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
26  RECVTYPE, ROOT, COMM, IERROR)
27  <type> SENDBUF(*), RECVBUF(*)
28  INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
29  COMM, IERROR
30
31 MPI_GATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
32  RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)
33  <type> SENDBUF(*), RECVBUF(*)
34  INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
35  COMM, INFO, REQUEST, IERROR
36
37 MPI_IALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
38  COMM, REQUEST, IERROR)
39  <type> SENDBUF(*), RECVBUF(*)
40  INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
41
42 MPI_IALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
43  RECVTYPE, COMM, REQUEST, IERROR)
44  <type> SENDBUF(*), RECVBUF(*)
45  INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
46  REQUEST, IERROR
47
48 MPI_IALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)
49  <type> SENDBUF(*), RECVBUF(*)
50  INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR
51
52 MPI_IALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM,
```

REQUEST, IERROR)	1
<type> SENDBUF(*), RECVBUF(*)	2
INTEGER SENDCOUNT, SENDTYPE, REVCOUNT, RECVTYPE, COMM, REQUEST, IERROR	3
MPI_IALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, REVCOUNTS,	4
RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)	5
<type> SENDBUF(*), RECVBUF(*)	6
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, REVCOUNTS(*), RDISPLS(*),	7
RECVTYPE, COMM, REQUEST, IERROR	8
	9
MPI_IALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, REVCOUNTS,	10
RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)	11
<type> SENDBUF(*), RECVBUF(*)	12
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), REVCOUNTS(*), RDISPLS(*),	13
RECVTYPES(*), COMM, REQUEST, IERROR	14
	15
MPI_IBARRIER(COMM, REQUEST, IERROR)	16
INTEGER COMM, REQUEST, IERROR	17
MPI_IBCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR)	18
<type> BUFFER(*)	19
INTEGER COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR	20
	21
MPI_IEXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)	22
<type> SENDBUF(*), RECVBUF(*)	23
INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR	24
	25
MPI_IGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, REVCOUNT, RECVTYPE, ROOT,	26
COMM, REQUEST, IERROR)	27
<type> SENDBUF(*), RECVBUF(*)	28
INTEGER SENDCOUNT, SENDTYPE, REVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,	29
IERROR	30
	31
MPI_IGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, REVCOUNTS, DISPLS,	32
RECVTYPE, ROOT, COMM, REQUEST, IERROR)	33
<type> SENDBUF(*), RECVBUF(*)	34
INTEGER SENDCOUNT, SENDTYPE, REVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,	35
COMM, REQUEST, IERROR	36
	37
MPI_IREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, IERROR)	38
<type> SENDBUF(*), RECVBUF(*)	39
INTEGER COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, IERROR	40
	41
MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, REVCOUNTS, DATATYPE, OP, COMM, REQUEST,	42
IERROR)	43
<type> SENDBUF(*), RECVBUF(*)	44
INTEGER REVCOUNTS(*), DATATYPE, OP, COMM, REQUEST, IERROR	45
	46
MPI_IREDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, REVCOUNT, DATATYPE, OP, COMM,	47
REQUEST, IERROR)	48
<type> SENDBUF(*), RECVBUF(*)	
INTEGER REVCOUNT, DATATYPE, OP, COMM, REQUEST, IERROR	

```
1 MPI_ISCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)
2   <type> SENDBUF(*), RECVBUF(*)
3   INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR
4
5 MPI_ISCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT,
6             COMM, REQUEST, IERROR)
7   <type> SENDBUF(*), RECVBUF(*)
8   INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,
9             IERROR
10
11 MPI_ISCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
12             RECVTYPE, ROOT, COMM, REQUEST, IERROR)
13   <type> SENDBUF(*), RECVBUF(*)
14   INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
15             COMM, REQUEST, IERROR
16
17 MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR)
18   INTEGER OP, IERROR
19   LOGICAL COMMUTE
20
21 MPI_OP_CREATE(USER_FN, COMMUTE, OP, IERROR)
22   EXTERNAL USER_FN
23   LOGICAL COMMUTE
24   INTEGER OP, IERROR
25
26 MPI_OP_FREE(OP, IERROR)
27   INTEGER OP, IERROR
28
29 MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
30   <type> SENDBUF(*), RECVBUF(*)
31   INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
32
33 MPI_REDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, INFO,
34             REQUEST, IERROR)
35   <type> SENDBUF(*), RECVBUF(*)
36   INTEGER COUNT, DATATYPE, OP, ROOT, COMM, INFO, REQUEST, IERROR
37
38 MPI_REDUCE_LOCAL(INBUF, INOUTBUF, COUNT, DATATYPE, OP, IERROR)
39   <type> INBUF(*), INOUTBUF(*)
40   INTEGER COUNT, DATATYPE, OP, IERROR
41
42 MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, IERROR)
43   <type> SENDBUF(*), RECVBUF(*)
44   INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR
45
46 MPI_REDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
47             IERROR)
48   <type> SENDBUF(*), RECVBUF(*)
49   INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR
50
51 MPI_REDUCE_SCATTER_BLOCK_INIT(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
52             INFO, REQUEST, IERROR)
53   <type> SENDBUF(*), RECVBUF(*)
54   INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR, INFO, REQUEST, IERROR
```



```

INTEGER RECVCOUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR 1
MPI_REDUCE_SCATTER_INIT(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, INFO, 2
    REQUEST, IERROR) 3
    <type> SENDBUF(*), RECVBUF(*) 4
    INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, INFO, REQUEST, IERROR 5
MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR) 6
    <type> SENDBUF(*), RECVBUF(*) 7
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR 8
MPI_SCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST, 9
    IERROR) 10
    <type> SENDBUF(*), RECVBUF(*) 11
    INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR 12
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, 13
    COMM, IERROR) 14
    <type> SENDBUF(*), RECVBUF(*) 15
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR 16
MPI_SCATTER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 17
    ROOT, COMM, INFO, REQUEST, IERROR) 18
    <type> SENDBUF(*), RECVBUF(*) 19
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO, 20
    REQUEST, IERROR 21
MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, 22
    RECVTYPE, ROOT, COMM, IERROR) 23
    <type> SENDBUF(*), RECVBUF(*) 24
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, 25
    COMM, IERROR 26
MPI_SCATTERV_INIT(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, 27
    RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR) 28
    <type> SENDBUF(*), RECVBUF(*) 29
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, 30
    COMM, INFO, REQUEST, IERROR 31
A.5.5 Groups, Contexts, Communicators, and Caching Fortran Bindings 32
MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR) 33
    INTEGER COMM1, COMM2, RESULT, IERROR 34
MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR) 35
    INTEGER COMM, GROUP, NEWCOMM, IERROR 36
MPI_COMM_CREATE_FROM_GROUP(GROUP, STRINGTAG, INFO, ERRHANDLER, NEWCOMM, IERROR) 37
    INTEGER GROUP, INFO, ERRHANDLER, NEWCOMM, IERROR 38
    CHARACTER*(*) STRINGTAG 39
MPI_COMM_CREATE_GROUP(COMM, GROUP, TAG, NEWCOMM, IERROR) 40

```

```

1      INTEGER COMM, GROUP, TAG, NEWCOMM, IERROR
2
3      MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL,
4          EXTRA_STATE, IERROR)
5          EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN
6          INTEGER COMM_KEYVAL, IERROR
7          INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
8
9      MPI_COMM_DELETE_ATTR(COMM, COMM_KEYVAL, IERROR)
10         INTEGER COMM, COMM_KEYVAL, IERROR
11
12     MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
13         INTEGER COMM, NEWCOMM, IERROR
14
15     MPI_COMM_DUP_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
16         ATTRIBUTE_VAL_OUT, FLAG, IERROR)
17         INTEGER OLDCOMM, COMM_KEYVAL, IERROR
18         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
19         ATTRIBUTE_VAL_OUT
20         LOGICAL FLAG
21
22     MPI_COMM_DUP_WITH_INFO(COMM, INFO, NEWCOMM, IERROR)
23         INTEGER COMM, INFO, NEWCOMM, IERROR
24
25     MPI_COMM_FREE(COMM, IERROR)
26         INTEGER COMM, IERROR
27
28     MPI_COMM_FREE_KEYVAL(COMM_KEYVAL, IERROR)
29         INTEGER COMM_KEYVAL, IERROR
30
31     MPI_COMM_GET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
32         INTEGER COMM, COMM_KEYVAL, IERROR
33         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
34         LOGICAL FLAG
35
36     MPI_COMM_GET_INFO(COMM, INFO_USED, IERROR)
37         INTEGER COMM, INFO_USED, IERROR
38
39     MPI_COMM_GET_NAME(COMM, COMM_NAME, RESULTLEN, IERROR)
40         INTEGER COMM, RESULTLEN, IERROR
41         CHARACTER*(*) COMM_NAME
42
43     MPI_COMM_GROUP(COMM, GROUP, IERROR)
44         INTEGER COMM, GROUP, IERROR
45
46     MPI_COMM_IDUP(COMM, NEWCOMM, REQUEST, IERROR)
47         INTEGER COMM, NEWCOMM, REQUEST, IERROR
48
49     MPI_COMM_IDUP_WITH_INFO(COMM, INFO, NEWCOMM, REQUEST, IERROR)
50         INTEGER COMM, INFO, NEWCOMM, REQUEST, IERROR
51
52     MPI_COMM_NULL_COPY_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
53         ATTRIBUTE_VAL_OUT, FLAG, IERROR)
54         INTEGER OLDCOMM, COMM_KEYVAL, IERROR

```

INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	1
ATTRIBUTE_VAL_OUT	2
LOGICAL FLAG	3
MPI_COMM_NULL_DELETE_FN(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)	4
INTEGER COMM, COMM_KEYVAL, IERROR	5
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	6
MPI_COMM_RANK(COMM, RANK, IERROR)	7
INTEGER COMM, RANK, IERROR	8
MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR)	9
INTEGER COMM, GROUP, IERROR	10
MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR)	11
INTEGER COMM, SIZE, IERROR	12
MPI_COMM_SET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, IERROR)	13
INTEGER COMM, COMM_KEYVAL, IERROR	14
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL	15
MPI_COMM_SET_INFO(COMM, INFO, IERROR)	16
INTEGER COMM, INFO, IERROR	17
MPI_COMM_SET_NAME(COMM, COMM_NAME, IERROR)	18
INTEGER COMM, IERROR	19
CHARACTER*(*) COMM_NAME	20
MPI_COMM_SIZE(COMM, SIZE, IERROR)	21
INTEGER COMM, SIZE, IERROR	22
MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)	23
INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR	24
MPI_COMM_SPLIT_TYPE(COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR)	25
INTEGER COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR	26
MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)	27
INTEGER COMM, IERROR	28
LOGICAL FLAG	29
MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR)	30
INTEGER GROUP1, GROUP2, RESULT, IERROR	31
MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)	32
INTEGER GROUP1, GROUP2, NEWGROUP, IERROR	33
MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)	34
INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR	35
MPI_GROUP_FREE(GROUP, IERROR)	36
INTEGER GROUP, IERROR	37
MPI_GROUP_FROM_SESSION_PSET(SESSION, PSET_NAME, NEWGROUP, IERROR)	38
INTEGER SESSION, NEWGROUP, IERROR	39
	40
	41
	42
	43
	44
	45
	46
	47
	48

```

1     CHARACTER*(*) PSET_NAME
2
3     MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
4         INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
5
6     MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR)
7         INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
8
9     MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR)
10        INTEGER GROUP, N, RANGES(3, *), NEWGROUP, IERROR
11
12    MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)
13        INTEGER GROUP, N, RANGES(3, *), NEWGROUP, IERROR
14
15    MPI_GROUP_RANK(GROUP, RANK, IERROR)
16        INTEGER GROUP, RANK, IERROR
17
18    MPI_GROUP_SIZE(GROUP, SIZE, IERROR)
19        INTEGER GROUP, SIZE, IERROR
20
21    MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR)
22        INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR
23
24    MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR)
25        INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
26
27    MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
28        NEWINTERCOMM, IERROR)
29        INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
30        NEWINTERCOMM, IERROR
31
32    MPI_INTERCOMM_CREATE_FROM_GROUPS(LOCAL_GROUP, LOCAL_LEADER, REMOTE_GROUP,
33        REMOTE_LEADER, STRINGTAG, INFO, ERRHANDLER, NEWINTERCOMM, IERROR)
34        INTEGER LOCAL_GROUP, LOCAL_LEADER, REMOTE_GROUP, REMOTE_LEADER, INFO,
35        ERRHANDLER, NEWINTERCOMM, IERROR
36        CHARACTER*(*) STRINGTAG
37
38    MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR)
39        INTEGER INTERCOMM, NEWINTRACOMM, IERROR
40        LOGICAL HIGH
41
42    MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL,
43        EXTRA_STATE, IERROR)
44        EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN
45        INTEGER TYPE_KEYVAL, IERROR
46        INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
47
48    MPI_TYPE_DELETE_ATTR(DATATYPE, TYPE_KEYVAL, IERROR)
49        INTEGER DATATYPE, TYPE_KEYVAL, IERROR
50
51    MPI_TYPE_DUP_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
52        ATTRIBUTE_VAL_OUT, FLAG, IERROR)
53        INTEGER OLDTYPE, TYPE_KEYVAL, IERROR

```

INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	1
ATTRIBUTE_VAL_OUT	2
LOGICAL FLAG	3
MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)	4
INTEGER TYPE_KEYVAL, IERROR	5
	6
MPI_TYPE_GET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)	7
INTEGER DATATYPE, TYPE_KEYVAL, IERROR	8
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL	9
LOGICAL FLAG	10
	11
MPI_TYPE_GET_NAME(DATATYPE, TYPE_NAME, RESULTLEN, IERROR)	12
INTEGER DATATYPE, RESULTLEN, IERROR	13
CHARACTER*(*) TYPE_NAME	14
	15
MPI_TYPE_NULL_COPY_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	16
ATTRIBUTE_VAL_OUT, FLAG, IERROR)	17
INTEGER OLDTYPE, TYPE_KEYVAL, IERROR	18
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	19
ATTRIBUTE_VAL_OUT	20
LOGICAL FLAG	21
	22
MPI_TYPE_NULL_DELETE_FN(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE,	23
IERROR)	24
INTEGER DATATYPE, TYPE_KEYVAL, IERROR	25
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	26
	27
MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)	28
INTEGER DATATYPE, TYPE_KEYVAL, IERROR	29
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL	30
	31
MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR)	32
INTEGER DATATYPE, IERROR	33
CHARACTER*(*) TYPE_NAME	34
	35
MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,	36
EXTRA_STATE, IERROR)	37
EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN	38
INTEGER WIN_KEYVAL, IERROR	39
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	40
	41
MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR)	42
INTEGER WIN, WIN_KEYVAL, IERROR	43
	44
MPI_WIN_DUP_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	45
ATTRIBUTE_VAL_OUT, FLAG, IERROR)	46
INTEGER OLDWIN, WIN_KEYVAL, IERROR	47
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	48
ATTRIBUTE_VAL_OUT	
LOGICAL FLAG	
MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)	

```

1      INTEGER WIN_KEYVAL, IERROR
2
3      MPI_WIN_GET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
4          INTEGER WIN, WIN_KEYVAL, IERROR
5          INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
6          LOGICAL FLAG
7
8      MPI_WIN_GET_NAME(WIN, WIN_NAME, RESULTLEN, IERROR)
9          INTEGER WIN, RESULTLEN, IERROR
10         CHARACTER*(*) WIN_NAME
11
12     MPI_WIN_NULL_COPY_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
13         ATTRIBUTE_VAL_OUT, FLAG, IERROR)
14         INTEGER OLDWIN, WIN_KEYVAL, IERROR
15         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
16         ATTRIBUTE_VAL_OUT
17         LOGICAL FLAG
18
19     MPI_WIN_NULL_DELETE_FN(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)
20         INTEGER WIN, WIN_KEYVAL, IERROR
21         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
22
23     MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR)
24         INTEGER WIN, WIN_KEYVAL, IERROR
25         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
26
27     MPI_WIN_SET_NAME(WIN, WIN_NAME, IERROR)
28         INTEGER WIN, IERROR
29         CHARACTER*(*) WIN_NAME

```

A.5.6 Process Topologies Fortran Bindings

```

30     MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
31         INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR
32
33     MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
34         INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
35         LOGICAL PERIODS(*), REORDER
36
37     MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
38         INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
39         LOGICAL PERIODS(*)
40
41     MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)
42         INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR
43         LOGICAL PERIODS(*)
44
45     MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
46         INTEGER COMM, COORDS(*), RANK, IERROR
47
48     MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)
49         INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR

```

MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)	1
INTEGER COMM, NEWCOMM, IERROR	2
LOGICAL REMAIN_DIMS(*)	3
	4
MPI_CARTDIM_GET(COMM, NDIMS, IERROR)	5
INTEGER COMM, NDIMS, IERROR	6
	7
MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)	8
INTEGER NNODES, NDIMS, DIMS(*), IERROR	9
	10
MPI_DIST_GRAPH_CREATE(COMM_OLD, N, SOURCES, DEGREES, DESTINATIONS, WEIGHTS, INFO, REORDER, COMM_DIST_GRAPH, IERROR)	11
INTEGER COMM_OLD, N, SOURCES(*), DEGREES(*), DESTINATIONS(*), WEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR	12
LOGICAL REORDER	13
	14
MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS, OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER, COMM_DIST_GRAPH, IERROR)	15
INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE, DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR	16
LOGICAL REORDER	17
	18
MPI_DIST_GRAPH_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, SOURCEWEIGHTS, MAXOUTDEGREE, DESTINATIONS, DESTWEIGHTS, IERROR)	19
INTEGER COMM, MAXINDEGREE, SOURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE, DESTINATIONS(*), DESTWEIGHTS(*), IERROR	20
	21
MPI_DIST_GRAPH_NEIGHBORS_COUNT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR)	22
INTEGER COMM, INDEGREE, OUTDEGREE, IERROR	23
LOGICAL WEIGHTED	24
	25
MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH, IERROR)	26
INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR	27
LOGICAL REORDER	28
	29
MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR)	30
INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR	31
	32
MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR)	33
INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR	34
	35
MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)	36
INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR	37
	38
MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)	39
INTEGER COMM, RANK, NNEIGHBORS, IERROR	40
	41
MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR)	42
INTEGER COMM, NNODES, NEDGES, IERROR	43
	44
MPI_INEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVMODE, COMM, REQUEST, IERROR)	45
<type> SENDBUF(*), RECVBUF(*)	46
	47
	48

```

1      INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVMYPE, COMM, REQUEST, IERROR
2
3      MPI_INEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
4          DISPLS, RECVMYPE, COMM, REQUEST, IERROR)
5      <type> SENDBUF(*), RECVBUF(*)
6      INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVMYPE, COMM,
7          REQUEST, IERROR
8
9      MPI_INEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
10         RECVMYPE, COMM, REQUEST, IERROR)
11     <type> SENDBUF(*), RECVBUF(*)
12     INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVMYPE, COMM, REQUEST, IERROR
13
14     MPI_INEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
15         RECVCOUNTS, RDISPLS, RECVMYPE, COMM, REQUEST, IERROR)
16     <type> SENDBUF(*), RECVBUF(*)
17     INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
18         RECVMYPE, COMM, REQUEST, IERROR
19
20     MPI_INEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
21         RECVCOUNTS, RDISPLS, RECVMYPES, COMM, REQUEST, IERROR)
22     <type> SENDBUF(*), RECVBUF(*)
23     INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVMYPES(*), COMM,
24         REQUEST, IERROR
25     INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
26
27     MPI_NEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
28         RECVMYPE, COMM, IERROR)
29     <type> SENDBUF(*), RECVBUF(*)
30     INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVMYPE, COMM, IERROR
31
32     MPI_NEIGHBOR_ALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
33         RECVMYPE, COMM, INFO, REQUEST, IERROR)
34     <type> SENDBUF(*), RECVBUF(*)
35     INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVMYPE, COMM, INFO, REQUEST,
36         IERROR
37
38     MPI_NEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
39         DISPLS, RECVMYPE, COMM, IERROR)
40     <type> SENDBUF(*), RECVBUF(*)
41     INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVMYPE, COMM,
42         IERROR
43
44     MPI_NEIGHBOR_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
45         DISPLS, RECVMYPE, COMM, INFO, REQUEST, IERROR)
46     <type> SENDBUF(*), RECVBUF(*)
47     INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVMYPE, COMM,
48         INFO, REQUEST, IERROR
49
50     MPI_NEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
51         RECVMYPE, COMM, IERROR)
52     <type> SENDBUF(*), RECVBUF(*)

```



```

INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVMODE, COMM, IERROR 1
MPI_NEIGHBOR_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 2
    RECVTYPE, COMM, INFO, REQUEST, IERROR) 3
<type> SENDBUF(*), RECVBUF(*) 4
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVMODE, COMM, INFO, REQUEST, 5
    IERROR 6
MPI_NEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, 8
    RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR) 9
<type> SENDBUF(*), RECVBUF(*) 10
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), 11
    RECVTYPE, COMM, IERROR 12
MPI_NEIGHBOR_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, 14
    RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR) 15
<type> SENDBUF(*), RECVBUF(*) 16
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), 17
    RECVTYPE, COMM, INFO, REQUEST, IERROR 18
MPI_NEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, 19
    RECVCOUNTS, RDISPLS, RECVTYPES, COMM, IERROR) 20
<type> SENDBUF(*), RECVBUF(*) 21
INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM, 22
    IERROR 23
INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*) 24
MPI_NEIGHBOR_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, 26
    RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST, IERROR) 27
<type> SENDBUF(*), RECVBUF(*) 28
INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM, 29
    INFO, REQUEST, IERROR 30
INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*) 31
MPI_TOPO_TEST(COMM, STATUS, IERROR) 32
INTEGER COMM, STATUS, IERROR 33
A.5.7 MPI Environmental Management Fortran Bindings 34
DOUBLE PRECISION MPI_WTICK() 35
DOUBLE PRECISION MPI_WTIME() 36
MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR) 37
    INTEGER ERRORCLASS, IERROR 38
MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR) 39
    INTEGER ERRORCLASS, ERRORCODE, IERROR 40
MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR) 41
    INTEGER ERRORCODE, IERROR 42
    CHARACTER*(*) STRING 43

```

```

1 MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
2   INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
3   INTEGER INFO, IERROR
4
5 If the Fortran compiler provides TYPE(C_PTR), then overloaded by:
6   INTERFACE MPI_ALLOC_MEM
7     SUBROUTINE MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
8       IMPORT :: MPI_ADDRESS_KIND
9       INTEGER :: INFO, IERROR
10      INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR
11    END SUBROUTINE
12    SUBROUTINE MPI_ALLOC_MEM_CPTR(SIZE, INFO, BASEPTR, IERROR)
13      USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
14      IMPORT :: MPI_ADDRESS_KIND
15      INTEGER :: INFO, IERROR
16      INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
17      TYPE(C_PTR) :: BASEPTR
18    END SUBROUTINE
19  END INTERFACE
20
21 MPI_COMM_CALL_ERRHANDLER(COMM, ERRORCODE, IERROR)
22   INTEGER COMM, ERRORCODE, IERROR
23
24 MPI_COMM_CREATE_ERRHANDLER(COMM_ERRHANDLER_FN, ERRHANDLER, IERROR)
25   EXTERNAL COMM_ERRHANDLER_FN
26   INTEGER ERRHANDLER, IERROR
27
28 MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
29   INTEGER COMM, ERRHANDLER, IERROR
30
31 MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
32   INTEGER COMM, ERRHANDLER, IERROR
33
34 MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)
35   INTEGER ERRHANDLER, IERROR
36
37 MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
38   INTEGER ERRORCODE, ERRORCLASS, IERROR
39
40 MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)
41   INTEGER ERRORCODE, RESULTLEN, IERROR
42   CHARACTER*(*) STRING
43
44 MPI_FILE_CALL_ERRHANDLER(FH, ERRORCODE, IERROR)
45   INTEGER FH, ERRORCODE, IERROR
46
47 MPI_FILE_CREATE_ERRHANDLER(FILE_ERRHANDLER_FN, ERRHANDLER, IERROR)
48   EXTERNAL FILE_ERRHANDLER_FN
49   INTEGER ERRHANDLER, IERROR
50
51 MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
52   INTEGER FILE, ERRHANDLER, IERROR
53
54 MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)

```

INTEGER FILE, ERRHANDLER, IERROR	1
MPI_FREE_MEM(BASE, IERROR)	2
<type> BASE(*)	3
INTEGER IERROR	4
MPI_GET_LIBRARY_VERSION(VERSION, RESULTLEN, IERROR)	5
CHARACTER*(*) VERSION	6
INTEGER RESULTLEN, IERROR	7
MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERROR)	8
CHARACTER*(*) NAME	9
INTEGER RESULTLEN, IERROR	10
MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)	11
INTEGER VERSION, SUBVERSION, IERROR	12
MPI_SESSION_CALL_ERRHANDLER(SESSION, ERRORCODE, IERROR)	13
INTEGER SESSION, ERRORCODE, IERROR	14
MPI_SESSION_CREATE_ERRHANDLER(SESSION_ERRHANDLER_FN, ERRHANDLER, IERROR)	15
EXTERNAL SESSION_ERRHANDLER_FN	16
INTEGER ERRHANDLER, IERROR	17
MPI_SESSION_GET_ERRHANDLER(SESSION, ERRHANDLER, IERROR)	18
INTEGER SESSION, ERRHANDLER, IERROR	19
MPI_SESSION_SET_ERRHANDLER(SESSION, ERRHANDLER, IERROR)	20
INTEGER SESSION, ERRHANDLER, IERROR	21
MPI_WIN_CALL_ERRHANDLER(WIN, ERRORCODE, IERROR)	22
INTEGER WIN, ERRORCODE, IERROR	23
MPI_WIN_CREATE_ERRHANDLER(WIN_ERRHANDLER_FN, ERRHANDLER, IERROR)	24
EXTERNAL WIN_ERRHANDLER_FN	25
INTEGER ERRHANDLER, IERROR	26
MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR)	27
INTEGER WIN, ERRHANDLER, IERROR	28
MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR)	29
INTEGER WIN, ERRHANDLER, IERROR	30
A.5.8 The Info Object Fortran Bindings	31
MPI_INFO_CREATE(INFO, IERROR)	32
INTEGER INFO, IERROR	33
MPI_INFO_CREATE_ENV(INFO, IERROR)	34
INTEGER INFO, IERROR	35
MPI_INFO_DELETE(INFO, KEY, IERROR)	36
INTEGER INFO, IERROR	37
CHARACTER*(*) KEY	38
	39
	40
	41
	42
	43
	44
	45
	46
	47
	48

```

1 MPI_INFO_DUP(INFO, NEWINFO, IERROR)
2     INTEGER INFO, NEWINFO, IERROR
3
4 MPI_INFO_FREE(INFO, IERROR)
5     INTEGER INFO, IERROR
6
7 MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)
8     INTEGER INFO, NKEYS, IERROR
9
10 MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERROR)
11     INTEGER INFO, N, IERROR
12     CHARACTER*(*) KEY
13
14 MPI_INFO_GET_STRING(INFO, KEY, BUFLen, VALUE, FLAG, IERROR)
15     INTEGER INFO, BUFLen, IERROR
16     CHARACTER*(*) KEY, VALUE
17     LOGICAL FLAG
18
19 MPI_INFO_SET(INFO, KEY, VALUE, IERROR)
20     INTEGER INFO, IERROR
21     CHARACTER*(*) KEY, VALUE
22
23 A.5.9 Process Creation and Management Fortran Bindings
24
25 MPI_ABORT(COMM, ERRORCODE, IERROR)
26     INTEGER COMM, ERRORCODE, IERROR
27
28 MPI_CLOSE_PORT(PORT_NAME, IERROR)
29     CHARACTER*(*) PORT_NAME
30     INTEGER IERROR
31
32 MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
33     CHARACTER*(*) PORT_NAME
34     INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
35
36 MPI_COMM_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
37     CHARACTER*(*) PORT_NAME
38     INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
39
40 MPI_COMM_DISCONNECT(COMM, IERROR)
41     INTEGER COMM, IERROR
42
43 MPI_COMM_GET_PARENT(PARENT, IERROR)
44     INTEGER PARENT, IERROR
45
46 MPI_COMM_JOIN(FD, INTERCOMM, IERROR)
47     INTEGER FD, INTERCOMM, IERROR
48
49 MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,
50     ARRAY_OF_ERRCODES, IERROR)
51     CHARACTER*(*) COMMAND, ARGV(*)
52     INTEGER MAXPROCS, INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR

```

MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,	1
ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,	2
ARRAY_OF_ERRCODES, IERROR)	3
INTEGER COUNT, ARRAY_OF_MAXPROCS(*), ARRAY_OF_INFO(*), ROOT, COMM,	4
INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR	5
CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)	6
	7
MPI_FINALIZE(IERROR)	8
INTEGER IERROR	9
	10
MPI_FINALIZED(FLAG, IERROR)	11
LOGICAL FLAG	12
INTEGER IERROR	13
	14
MPI_INIT(IERROR)	15
INTEGER IERROR	16
	17
MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)	18
INTEGER REQUIRED, PROVIDED, IERROR	19
	20
MPI_INITIALIZED(FLAG, IERROR)	21
LOGICAL FLAG	22
INTEGER IERROR	23
	24
MPI_IS_THREAD_MAIN(FLAG, IERROR)	25
LOGICAL FLAG	26
INTEGER IERROR	27
	28
MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)	29
CHARACTER*(*) SERVICE_NAME, PORT_NAME	30
INTEGER INFO, IERROR	31
	32
MPI_OPEN_PORT(INFO, PORT_NAME, IERROR)	33
INTEGER INFO, IERROR	34
CHARACTER*(*) PORT_NAME	35
	36
MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)	37
CHARACTER*(*) SERVICE_NAME, PORT_NAME	38
INTEGER INFO, IERROR	39
	40
MPI_QUERY_THREAD(PROVIDED, IERROR)	41
INTEGER PROVIDED, IERROR	42
	43
MPI_SESSION_FINALIZE(SESSION, IERROR)	44
INTEGER SESSION, IERROR	45
	46
MPI_SESSION_GET_INFO(SESSION, INFO_USED, IERROR)	47
INTEGER SESSION, INFO_USED, IERROR	48
MPI_SESSION_GET_NTH_PSET(SESSION, INFO, N, PSET_LEN, PSET_NAME, IERROR)	
INTEGER SESSION, INFO, N, PSET_LEN, IERROR	
CHARACTER*(*) PSET_NAME	
MPI_SESSION_GET_NUM_PSETS(SESSION, INFO, NPSET_NAMES, IERROR)	

```

1      INTEGER SESSION, INFO, NPSET_NAMES, IERROR
2
3      MPI_SESSION_GET_PSET_INFO(SESSION, PSET_NAME, INFO, IERROR)
4          INTEGER SESSION, INFO, IERROR
5          CHARACTER*(*) PSET_NAME
6
7      MPI_SESSION_INIT(INFO, ERRHANDLER, SESSION, IERROR)
8          INTEGER INFO, ERRHANDLER, SESSION, IERROR
9
10     MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
11         CHARACTER*(*) SERVICE_NAME, PORT_NAME
12         INTEGER INFO, IERROR
13
14     A.5.10 One-Sided Communications Fortran Bindings
15
16     MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
17                   TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
18         <type> ORIGIN_ADDR(*)
19         INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
20                   TARGET_DATATYPE, OP, WIN, IERROR
21         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
22
23     MPI_COMPARE_AND_SWAP(ORIGIN_ADDR, COMPARE_ADDR, RESULT_ADDR, DATATYPE,
24                          TARGET_RANK, TARGET_DISP, WIN, IERROR)
25         <type> ORIGIN_ADDR(*), COMPARE_ADDR(*), RESULT_ADDR(*)
26         INTEGER DATATYPE, TARGET_RANK, WIN, IERROR
27         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
28
29     MPI_FETCH_AND_OP(ORIGIN_ADDR, RESULT_ADDR, DATATYPE, TARGET_RANK, TARGET_DISP,
30                    OP, WIN, IERROR)
31         <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
32         INTEGER DATATYPE, TARGET_RANK, OP, WIN, IERROR
33         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
34
35     MPI_GET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP,
36            TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
37         <type> ORIGIN_ADDR(*)
38         INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
39                   TARGET_DATATYPE, WIN, IERROR
40         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
41
42     MPI_GET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_ADDR,
43                      RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, TARGET_DISP,
44                      TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
45         <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
46         INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
47                   TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR
48         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
49
50     MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP,
51            TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)

```

```

<type> ORIGIN_ADDR(*)
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
      TARGET_DATATYPE, WIN, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
MPI_RACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
      TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
      IERROR)
<type> ORIGIN_ADDR(*)
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
      TARGET_DATATYPE, OP, WIN, REQUEST, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
MPI_RGET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP,
      TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST, IERROR)
<type> ORIGIN_ADDR(*)
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
      TARGET_DATATYPE, WIN, REQUEST, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
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 ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
      TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
      IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
MPI_RPUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP,
      TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST, IERROR)
<type> ORIGIN_ADDR(*)
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
      TARGET_DATATYPE, WIN, REQUEST, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
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

```

```

1      IMPORT :: MPI_ADDRESS_KIND
2      INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
3      INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
4      TYPE(C_PTR) :: BASEPTR
5      END SUBROUTINE
6      END INTERFACE
7
8      MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
9      INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
10     INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
11     If the Fortran compiler provides TYPE(C_PTR), then overloaded by:
12     INTERFACE MPI_WIN_ALLOCATE_SHARED
13         SUBROUTINE MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, &
14             BASEPTR, WIN, IERROR)
15             IMPORT :: MPI_ADDRESS_KIND
16             INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
17             INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR
18         END SUBROUTINE
19         SUBROUTINE MPI_WIN_ALLOCATE_SHARED_CPTR(SIZE, DISP_UNIT, INFO, COMM, &
20             BASEPTR, WIN, IERROR)
21             USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
22             IMPORT :: MPI_ADDRESS_KIND
23             INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
24             INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
25             TYPE(C_PTR) :: BASEPTR
26         END SUBROUTINE
27     END INTERFACE
28
29     MPI_WIN_ATTACH(WIN, BASE, SIZE, IERROR)
30     INTEGER WIN, IERROR
31     <type> BASE(*)
32     INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
33
34     MPI_WIN_COMPLETE(WIN, IERROR)
35     INTEGER WIN, IERROR
36
37     MPI_WIN_CREATE(BASE, SIZE, DISP_UNIT, INFO, COMM, WIN, IERROR)
38     <type> BASE(*)
39     INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
40     INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
41
42     MPI_WIN_CREATE_DYNAMIC(INFO, COMM, WIN, IERROR)
43     INTEGER INFO, COMM, WIN, IERROR
44
45     MPI_WIN_DETACH(WIN, BASE, IERROR)
46     INTEGER WIN, IERROR
47     <type> BASE(*)
48
49     MPI_WIN_FENCE(ASSERT, WIN, IERROR)
50     INTEGER ASSERT, WIN, IERROR

```


MPI_WIN_FLUSH(RANK, WIN, IERROR)	1
INTEGER RANK, WIN, IERROR	2
	3
MPI_WIN_FLUSH_ALL(WIN, IERROR)	4
INTEGER WIN, IERROR	5
	6
MPI_WIN_FLUSH_LOCAL(RANK, WIN, IERROR)	7
INTEGER RANK, WIN, IERROR	8
	9
MPI_WIN_FLUSH_LOCAL_ALL(WIN, IERROR)	10
INTEGER WIN, IERROR	11
	12
MPI_WIN_FREE(WIN, IERROR)	13
INTEGER WIN, IERROR	14
	15
MPI_WIN_GET_GROUP(WIN, GROUP, IERROR)	16
INTEGER WIN, GROUP, IERROR	17
	18
MPI_WIN_GET_INFO(WIN, INFO_USED, IERROR)	19
INTEGER WIN, INFO_USED, IERROR	20
	21
MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR)	22
INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR	23
	24
MPI_WIN_LOCK_ALL(ASSERT, WIN, IERROR)	25
INTEGER ASSERT, WIN, IERROR	26
	27
MPI_WIN_POST(GROUP, ASSERT, WIN, IERROR)	28
INTEGER GROUP, ASSERT, WIN, IERROR	29
	30
MPI_WIN_SET_INFO(WIN, INFO, IERROR)	31
INTEGER WIN, INFO, IERROR	32
	33
MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, BASEPTR, IERROR)	34
INTEGER WIN, RANK, DISP_UNIT, IERROR	35
INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR	36
	37
If the Fortran compiler provides TYPE(C_PTR), then overloaded by:	38
INTERFACE MPI_WIN_SHARED_QUERY	39
SUBROUTINE MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, &	40
BASEPTR, IERROR)	41
IMPORT :: MPI_ADDRESS_KIND	42
INTEGER :: WIN, RANK, DISP_UNIT, IERROR	43
INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR	44
END SUBROUTINE	45
SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, &	46
BASEPTR, IERROR)	47
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	48
IMPORT :: MPI_ADDRESS_KIND	
INTEGER :: WIN, RANK, DISP_UNIT, IERROR	
INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE	
TYPE(C_PTR) :: BASEPTR	
END SUBROUTINE	
END INTERFACE	

```

1 MPI_WIN_START(GROUP, ASSERT, WIN, IERROR)
2     INTEGER GROUP, ASSERT, WIN, IERROR
3
4 MPI_WIN_SYNC(WIN, IERROR)
5     INTEGER WIN, IERROR
6
7 MPI_WIN_TEST(WIN, FLAG, IERROR)
8     INTEGER WIN, IERROR
9     LOGICAL FLAG
10
11 MPI_WIN_UNLOCK(RANK, WIN, IERROR)
12     INTEGER RANK, WIN, IERROR
13
14 MPI_WIN_UNLOCK_ALL(WIN, IERROR)
15     INTEGER WIN, IERROR
16
17 MPI_WIN_WAIT(WIN, IERROR)
18     INTEGER WIN, IERROR
19
20 A.5.11 External Interfaces Fortran Bindings
21
22 MPI_GREQUEST_COMPLETE(REQUEST, IERROR)
23     INTEGER REQUEST, IERROR
24
25 MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST, IERROR)
26     EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN
27     INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
28     INTEGER REQUEST, IERROR
29
30 MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR)
31     INTEGER STATUS(MPI_STATUS_SIZE), IERROR
32     LOGICAL FLAG
33
34 MPI_STATUS_SET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
35     INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
36
37 MPI_STATUS_SET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)
38     INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR
39     INTEGER(KIND=MPI_COUNT_KIND) COUNT
40
41 A.5.12 I/O Fortran Bindings
42
43 MPI_CONVERSION_FN_NULL(USERBUF, DATATYPE, COUNT, FILEBUF, POSITION,
44     EXTRA_STATE, IERROR)
45     <TYPE> USERBUF(*), FILEBUF(*)
46     INTEGER DATATYPE, COUNT, IERROR
47     INTEGER(KIND=MPI_OFFSET_KIND) POSITION
48     INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
49
50 MPI_FILE_CLOSE(FH, IERROR)
51     INTEGER FH, IERROR
52
53

```

MPI_FILE_DELETE(FILENAME, INFO, IERROR)	1
CHARACTER*(*) FILENAME	2
INTEGER INFO, IERROR	3
	4
MPI_FILE_GET_AMODE(FH, AMODE, IERROR)	5
INTEGER FH, AMODE, IERROR	6
	7
MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR)	8
INTEGER FH, IERROR	9
LOGICAL FLAG	10
	11
MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR)	12
INTEGER FH, IERROR	13
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP	14
	15
MPI_FILE_GET_GROUP(FH, GROUP, IERROR)	16
INTEGER FH, GROUP, IERROR	17
	18
MPI_FILE_GET_INFO(FH, INFO_USED, IERROR)	19
INTEGER FH, INFO_USED, IERROR	20
	21
MPI_FILE_GET_POSITION(FH, OFFSET, IERROR)	22
INTEGER FH, IERROR	23
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	24
	25
MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERROR)	26
INTEGER FH, IERROR	27
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	28
	29
MPI_FILE_GET_SIZE(FH, SIZE, IERROR)	30
INTEGER FH, IERROR	31
INTEGER(KIND=MPI_OFFSET_KIND) SIZE	32
	33
MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERROR)	34
INTEGER FH, DATATYPE, IERROR	35
INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT	36
	37
MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)	38
INTEGER FH, ETYPE, FILETYPE, IERROR	39
INTEGER(KIND=MPI_OFFSET_KIND) DISP	40
CHARACTER*(*) DATAREP	41
	42
MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)	43
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR	44
<type> BUF(*)	45
	46
MPI_FILE_IREAD_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)	47
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR	48
<type> BUF(*)	
MPI_FILE_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)	
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR	
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	
<type> BUF(*)	

```
1 MPI_FILE_IREAD_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
2   INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
3   INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
4   <type> BUF(*)
5
6 MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
7   INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
8   <type> BUF(*)
9
10 MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
11   INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
12   <type> BUF(*)
13
14 MPI_FILE_IWRITE_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
15   INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
16   <type> BUF(*)
17
18 MPI_FILE_IWRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
19   INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
20   INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
21   <type> BUF(*)
22
23 MPI_FILE_IWRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
24   INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
25   INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
26   <type> BUF(*)
27
28 MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
29   INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
30   <type> BUF(*)
31
32 MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR)
33   INTEGER COMM, AMODE, INFO, FH, IERROR
34   CHARACTER*(*) FILENAME
35
36 MPI_FILE_PREALLOCATE(FH, SIZE, IERROR)
37   INTEGER FH, IERROR
38   INTEGER(KIND=MPI_OFFSET_KIND) SIZE
39
40 MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
41   INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
42   <type> BUF(*)
43
44 MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
45   INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
46   <type> BUF(*)
47
48 MPI_FILE_READ_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
49   INTEGER FH, COUNT, DATATYPE, IERROR
50   <type> BUF(*)
51
52 MPI_FILE_READ_ALL_END(FH, BUF, STATUS, IERROR)
53   INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
```

<type> BUF(*)	1
MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)	2
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	3
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	4
<type> BUF(*)	5
MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)	6
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	7
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	8
<type> BUF(*)	9
MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)	10
INTEGER FH, COUNT, DATATYPE, IERROR	11
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	12
<type> BUF(*)	13
MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)	14
INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR	15
<type> BUF(*)	16
MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)	17
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	18
<type> BUF(*)	19
MPI_FILE_READ_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)	20
INTEGER FH, COUNT, DATATYPE, IERROR	21
<type> BUF(*)	22
MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR)	23
INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR	24
<type> BUF(*)	25
MPI_FILE_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)	26
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	27
<type> BUF(*)	28
MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)	29
INTEGER FH, WHENCE, IERROR	30
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	31
MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR)	32
INTEGER FH, WHENCE, IERROR	33
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	34
MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR)	35
INTEGER FH, IERROR	36
LOGICAL FLAG	37
MPI_FILE_SET_INFO(FH, INFO, IERROR)	38
INTEGER FH, INFO, IERROR	39
MPI_FILE_SET_SIZE(FH, SIZE, IERROR)	40
INTEGER FH, IERROR	41
	42
	43
	44
	45
	46
	47
	48

```
1     INTEGER(KIND=MPI_OFFSET_KIND) SIZE
2
3 MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)
4     INTEGER FH, ETYPE, FILETYPE, INFO, IERROR
5     INTEGER(KIND=MPI_OFFSET_KIND) DISP
6     CHARACTER*(*) DATAREP
7
8 MPI_FILE_SYNC(FH, IERROR)
9     INTEGER FH, IERROR
10
11 MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
12     INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
13     <type> BUF(*)
14
15 MPI_FILE_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
16     INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
17     <type> BUF(*)
18
19 MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
20     INTEGER FH, COUNT, DATATYPE, IERROR
21     <type> BUF(*)
22
23 MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)
24     INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
25     <type> BUF(*)
26
27 MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
28     INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
29     INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
30     <type> BUF(*)
31
32 MPI_FILE_WRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
33     INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
34     INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
35     <type> BUF(*)
36
37 MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
38     INTEGER FH, COUNT, DATATYPE, IERROR
39     INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
40     <type> BUF(*)
41
42 MPI_FILE_WRITE_AT_ALL_END(FH, BUF, STATUS, IERROR)
43     INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
44     <type> BUF(*)
45
46 MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
47     INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
48     <type> BUF(*)
49
50 MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
51     INTEGER FH, COUNT, DATATYPE, IERROR
52     <type> BUF(*)
53
54 MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)
```

INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR	1
<type> BUF(*)	2
MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)	3
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	4
<type> BUF(*)	5
MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN,	6
DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR)	7
CHARACTER*(*) DATAREP	8
EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN	9
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	10
INTEGER IERROR	11
	12
	13
	14
A.5.13 Language Bindings Fortran Bindings	15
MPI_F_SYNC_REG(BUF)	16
<type> BUF(*)	17
	18
The following procedure is not available with mpif.h:	19
MPI_STATUS_F082F(F08_STATUS, F_STATUS, IERROR)	20
TYPE(MPI_Status) :: F08_STATUS	21
INTEGER :: F_STATUS(MPI_STATUS_SIZE), IERROR	22
	23
The following procedure is not available with mpif.h:	24
MPI_STATUS_F2F08(F_STATUS, F08_STATUS, IERROR)	25
INTEGER :: F_STATUS(MPI_STATUS_SIZE), IERROR	26
TYPE(MPI_Status) :: F08_STATUS	27
MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR)	28
INTEGER P, R, NEWTYPE, IERROR	29
	30
MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR)	31
INTEGER R, NEWTYPE, IERROR	32
MPI_TYPE_CREATE_F90_REAL(P, R, NEWTYPE, IERROR)	33
INTEGER P, R, NEWTYPE, IERROR	34
	35
MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, DATATYPE, IERROR)	36
INTEGER TYPECLASS, SIZE, DATATYPE, IERROR	37
	38
	39
A.5.14 Tools / Profiling Interface Fortran Bindings	40
MPI_PCONTROL(LEVEL)	41
INTEGER LEVEL	42
	43
	44
A.5.15 Deprecated Fortran Bindings	45
MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)	46
INTEGER COMM, KEYVAL, IERROR	47
	48

```
1 MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
2     INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
3     LOGICAL FLAG
4
5 MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
6     INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
7
8 MPI_DUP_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT,
9           FLAG, IERR)
10    INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT,
11          IERR
12    LOGICAL FLAG
13
14 MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)
15    INTEGER INFO, VALUELEN, IERROR
16    CHARACTER*(*) KEY, VALUE
17    LOGICAL FLAG
18
19 MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
20    INTEGER INFO, VALUELEN, IERROR
21    CHARACTER*(*) KEY
22    LOGICAL FLAG
23
24 MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
25    EXTERNAL COPY_FN, DELETE_FN
26    INTEGER KEYVAL, EXTRA_STATE, IERROR
27
28 MPI_KEYVAL_FREE(KEYVAL, IERROR)
29    INTEGER KEYVAL, IERROR
30
31 MPI_NULL_COPY_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
32                ATTRIBUTE_VAL_OUT, FLAG, IERR)
33    INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT,
34          IERR
35    LOGICAL FLAG
36
37 MPI_NULL_DELETE_FN(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)
38    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR
39
40 MPI_SIZEOF(X, SIZE, IERROR)
41    <type> X
42    INTEGER SIZE, IERROR
43
44
45
46
47
48
```


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.6 were already introduced in the corresponding sections in previous versions of this standard.

B.1 Changes from Version 4.0 to Version 4.1

B.1.1 Fixes to Errata in Previous Versions of MPI

1. Section 4.3 on page 108, and MPI-4.0 Section 4.3.3. on page 115. Example 4.4 on page 111, and MPI-4.0 Example 4.4 on page 115.
Fixed and simplified erroneous MPI-4.0 Example 4.4. The example could deadlock due to incorrect use of the flag variable in multiple MPI test procedure calls or thread concurrent access. The example was also simplified by removing unnecessary code and updated according to current best practice in OpenMP.
2. Section 7.8 on page 360 was amended to allow MPI_COMM_NULL, MPI_DATATYPE_NULL, and MPI_WIN_NULL to be passed to MPI_COMM_GET_NAME, MPI_TYPE_GET_NAME and MPI_WIN_GET_NAME, respectively.
3. Section 8.5.5 on page 381 and MPI-4.0 Section 8.5.5 on page 403.
The unintended change in the specification of argument coords in MPI_CART_COORDS in MPI-4.0 is reverted to the original meaning in MPI-1.1 to MPI-3.1. It is clarified that the outcome of MPI_CART_GET and MPI_CART_COORDS is unspecified for the case that maxdims is less than ndims.
4. Section 12.5.2 on page 565.
The definition of MPI_WIN_TEST was clarified.
5. Section 12.5.4 on page 574, MPI-3 Section 11.5.4 on page 450, MPI-3.1 Section 11.5.4 on page 450, and MPI-4 Section 12.5.4 on page 607.
The description of MPI_WIN_SYNC was clarified to include its use for ordering load/store accesses to shared memory. A statement was added to highlight that a call to MPI_WIN_SYNC does not complete pending RMA operations and that a call to MPI_WIN_SYNC does not guarantee any progress of MPI operations.

B.1.2 Changes in MPI-4.1

1. Section 3.7.6 on page 79.
The *progress* rule of MPI_REQUEST_GET_STATUS was clarified.
2. Section 9.4 on page 443.
Added new error class MPI_ERR_ERRHANDLER.
3. Section 12.2.5 on page 535.
Implementations may avoid synchronization of processes in MPI_WIN_FREE if the "no_locks" info key is set to "true".

B.2 Changes from Version 3.1 to Version 4.0

B.2.1 Fixes to Errata in Previous Versions of MPI

1. Sections 8.6.1, 8.6.2 and 8.9 on pages 395, 399 and 421, and MPI-3.1 Sections 7.6.1, 7.6.2 and 7.8 on pages 315, 318 and 329.
MPI_NEIGHBOR_ALLTOALL_{|V|W} and MPI_NEIGHBOR_ALLGATHER_{|V} for Cartesian virtual grids were clarified. An advice to implementors was added to illustrate a correct implementation for the case of periods[d]=1 or .TRUE. and dims[d]=1 or 2 in a direction d.
2. Sections 15.3.6 and 15.3.7 on pages 701 and 708.
The intent of handle arguments of the language independent definition of MPI_T_CVAR_WRITE, MPI_T_PVAR_HANDLE_ALLOC, MPI_T_PVAR_HANDLE_FREE, MPI_T_PVAR_START, MPI_T_PVAR_STOP, MPI_T_PVAR_WRITE, and MPI_T_PVAR_RESET were marked as INOUT in accordance with the special rule for handles described in Section 2.3.
3. Section 19.3.5 on page 806, and MPI-3.1 Section 17.2.5 on page 657 line 11.
Clarified that the MPI_STATUS_F2F08 and MPI_STATUS_F082F routines and the declaration for TYPE(MPI_Status) are not supposed to appear with mpif.h.
4. Sections 2.5.4, 19.3.5, and A.1.1 on pages 18, 805, and 820, and MPI-3.1 Sections 2.5.4, 17.2.5, and A.1.1 on pages 15, 656, and 669.
Define the C constants MPI_F_STATUS_SIZE, MPI_F_SOURCE, MPI_F_TAG, and MPI_F_ERROR.
5. Section 19.3.5 on page 806, and MPI-3.1 Section 17.2.5 on page 658.
Added missing const to IN parameters for MPI_STATUS_F2F08 and MPI_STATUS_F082F.

B.2.2 Changes in MPI-4.0

1. Sections 2.2, 18.2.1, and 19.1.5 on pages 9, 753, and 762.
The limit for the maximum length of MPI identifiers was removed.
2. Section 2.4, 3.4, 3.7.2, 3.7.3, 3.8.1, 3.8.2, 6.13, 14.4.5, and Annex A.2 on pages 11, 46, 60, 67, 80, 83, 260, 653, and 841.
The semantic terms were updated.

3. Sections [2.5.8](#) and [19.2](#) on pages [20](#) and [800](#), and throughout the entire document.

New large count functions `MPI_{...}_c` in C and through function overloading in the Fortran `mpi_f08` module, (with the exception of the explicit Fortran procedures `MPI_Op_create_c` and `MPI_Register_datarep_c`) and the new large count callbacks `MPI_User_function_c` and `MPI_Datarep_conversion_function_c` together with the pre-defined function `MPI_CONVERSION_FN_NULL_C` were introduced to accomodate large buffers and/or datatypes.

Clarifications were added to the behavior of INOUT/OUT parameters that cannot represent the value to be returned for the `MPI_BUFFER_DETACH` and `MPI_FILE_GET_TYPE_EXTENT` functions.

A new error class `MPI_ERR_VALUE_TOO_LARGE` was introduced.
4. Sections [2.8](#), [9.3](#), [9.5](#), and [11.2.1](#) on pages [24](#), [432](#), [446](#), and [462](#).

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.
5. Section [3.7](#) on page [58](#).

The introduction of MPI nonblocking communication was changed to describe correctness and performance reasons for the use of nonblocking communication.
6. Section [3.7.2](#) on page [60](#).

Addition of `MPI_ISENDRECV` and `MPI_ISENDRECV_REPLACE`.
7. Sections [3.7.3](#), [3.9](#), [6.13](#), [8.8](#), and [8.9](#) on pages [67](#), [90](#), [260](#), [413](#), and [421](#).

Persistent collective communication `MPI_{ALLGATHER|...}_INIT` including persistent collective neighborhood communication `MPI_NEIGHBOR_{ALLGATHER|...}_INIT` was added to the standard.
8. Sections [3.8.4](#) and [16.3](#) on pages [88](#) and [747](#).

Cancelling a send request by calling `MPI_CANCEL` has been deprecated and may be removed in a future version of the MPI specification.
9. Chapter [4](#) on page [99](#).

A new chapter on partitioned communication with the new MPI procedures `MPI_{PARRIVED|PREADY}{...}` and `MPI_{PRECV|PSEND}_INIT` was added.
10. Section [7.4.2](#) on page [307](#).

`MPI_COMM_TYPE_HW_UNGUIDED` was added as a new possible value for the `split_type` parameter of the `MPI_COMM_SPLIT_TYPE` function.
11. Section [7.4.2](#) on page [307](#).

`MPI_COMM_TYPE_HW_GUIDED` was added as a new possible value for the `split_type` parameter of the `MPI_COMM_SPLIT_TYPE` function, as well as a new info key "mpi_hw_resource_type". A specific value associated with this new info key is also defined: "mpi_shared_memory".
12. Section [7.4.2](#) on page [307](#).

The functions `MPI_COMM_DUP` and `MPI_COMM_IDUP` were updated to no longer

- 1 propagate info hints.
2 This change may affect backward compatibility.
- 3
4 13. Section [7.4.2](#) on page [307](#).
5 The `MPI_COMM_IDUP_WITH_INFO` function was added.
- 6
7 14. Sections [7.4.4](#), [12.2.7](#), and [14.2.8](#) on pages [324](#), [537](#), and [618](#).
8 The definition of info hints was updated to allow applications to provide assertions
9 regarding their usage of MPI objects and operations.
- 10
11 15. Section [7.4.4](#) on page [324](#).
12 The new info hints "`mpi_assert_no_any_tag`", "`mpi_assert_no_any_source`",
13 "`mpi_assert_exact_length`", and "`mpi_assert_allow_overtaking`" were added for use with com-
14 municators.
- 15
16 16. Sections [7.4.4](#), [12.2.7](#), and [14.2.8](#) on pages [324](#), [537](#), and [618](#).
17 The semantics of the `MPI_COMM_SET_INFO`, `MPI_COMM_GET_INFO`,
18 `MPI_WIN_SET_INFO`, `MPI_WIN_GET_INFO`, `MPI_FILE_SET_INFO`, and
19 `MPI_FILE_GET_INFO` were clarified.
- 20
21 17. Section [8.5](#) on page [370](#).
22 `MPI_DIMS_CREATE` is now guaranteed to return `MPI_SUCCESS` if the number of di-
23 mensions passed to the routine is set to 0 and the number of nodes is set to 1.
- 24
25 18. Sections [9.2](#), [12.2.2](#), and [12.2.3](#) on pages [429](#), [525](#), and [527](#).
26 Introduced alignment requirements for memory allocated through `MPI_ALLOC_MEM`,
27 `MPI_WIN_ALLOCATE`, and `MPI_WIN_ALLOCATE_SHARED` and added a new info key
28 "`mpi_minimum_memory_alignment`" to specify a desired alternative minimum alignment.
- 29
30 19. Sections [9.3](#) and [9.4](#) on pages [432](#) and [443](#).
31 Clarified definition of errors to say that MPI should continue whenever possible and
32 allow the user to recover from errors.
- 33
34 20. Section [9.4](#) on page [443](#).
35 Added text to clarify what is implied about the status of MPI and user visible buffers
36 when MPI functions return `MPI_SUCCESS` or other error codes.
- 37
38 21. Section [9.4](#) on page [444](#).
39 The error class `MPI_ERR_PROC_ABORTED` has been added.
- 40
41 22. Section [10](#) on page [453](#).
42 Added a new function `MPI_INFO_GET_STRING` that takes a buffer length argument
43 for returning info value strings. This function returns the required buffer length for
44 the requested string and guarantees null termination for C strings where buffer size
45 is greater than 0.
- 46
47 23. Section [10](#) on page [453](#) and Section [16.3](#) on page [747](#).
48 `MPI_INFO_GET` and `MPI_INFO_GET_VALUELEN` were deprecated.
- 49
50 24. Chapter [11](#), [3.2.3](#), [7.2.4](#), [7.3.2](#), [7.4.2](#), [7.6.2](#), [9.1.1](#), [9.1.2](#), [9.3](#), [9.3.4](#), [9.5](#), [11.6](#), [14.2.1](#),
[14.2.7](#), [14.7](#), [15.3.4](#), [19.3.4](#), [19.3.6](#), and Annex [A](#) on pages [461](#), [33](#), [295](#), [298](#), [307](#), [337](#),
[425](#), [426](#), [432](#), [440](#), [446](#), [489](#), [611](#), [617](#), [683](#), [698](#), [803](#), [807](#), and [817](#).

- The Sessions Model was added to the standard. New MPI procedures are MPI_SESSION_{INIT|FINALIZE}, MPI_SESSION_GET_{...}, MPI_SESSION_{...}_ERRHANDLER, MPI_GROUP_FROM_SESSION_PSET, MPI_COMM_CREATE_FROM_GROUP, MPI_INTERCOMM_CREATE_FROM_GROUPS, and new conversion functions are MPI_SESSION_{C2F|F2C}. New declarations are MPI_Session in C and TYPE(MPI_Session) together with the related overloaded operators .EQ., .NE., == and /= in the Fortran mpi_f08 and mpi modules, and the callback function prototype MPI_Session_errhandler_function. New constants are MPI_SESSION_NULL, MPI_ERR_SESSION, MPI_MAX_PSET_NAME_LEN, MPI_MAX_STRINGTAG_LEN, MPI_T_BIND_MPI_SESSION and the predefined info key "mpi_size".
25. Section 11.2.1 on page 462.
A new function MPI_INFO_CREATE_ENV was added.
 26. Sections 11.2.1 and 11.10.4 on pages 462 and 517.
Clarified the semantic of failure and error reporting before (and during) MPI_INIT and after MPI_FINALIZE.
 27. Section 11.8.4 on page 502.
Added the "mpi_initial_errhandler" reserved info key with the reserved values "mpi_errors_abort", "mpi_errors_are_fatal", and "mpi_errors_return" to the launch keys in MPI_COMM_SPAWN, MPI_COMM_SPAWN_MULTIPLE, and mpiexec.
 28. Section 12.5.3 on page 569.
RMA passive target synchronization using locks can now be used portably in memory allocated via MPI_WIN_ALLOCATE_SHARED.
 29. Section 13.3 on page 605.
The mpi_f08 binding incorrectly had the dummy parameter flag in the MPI F08 binding for MPI_STATUS_SET_CANCELLED marked as INTENT(OUT). It has been fixed to be INTENT(IN).
 30. Sections 15.3 and 15.3.8 on pages 695 and 721.
A callback-driven event interface with the MPI_T_{SOURCE|EVENT}_{...} and MPI_T_CATEGORY_{GET|GET_NUM}_EVENTS routines, the declaration types MPI_T_cb_safety, MPI_T_event_{instance|registration}, MPI_T_source_order, and the callback function prototypes MPI_T_event_{cb|dropped_cb|free_cb}_function, was added to the MPI tool information interface.
 31. Section 15.3.9 on page 739.
The argument stamp (previously described as a virtual time stamp) from MPI_T_CATEGORY_CHANGED was renamed to update_number and its intended implementation and use was clarified.
 32. Section 15.3.10, Table 15.7, and Section 16.3 on pages 740, 741, and 747.
MPI_T_ERR_INVALID_ITEM is deprecated. MPI routines should return MPI_T_ERR_INVALID_INDEX instead of MPI_T_ERR_INVALID_ITEM.
 33. Section 16.3 on page 749.
MPI_SIZEOF was deprecated.

1 34. Section [19.1.5](#) on page [762](#).

2 An exception was added for the specific Fortran names in the case of TS 29113 interface
3 specifications in `mpif.h` for `MPI_NEIGHBOR_ALLTOALLW_INIT`,
4 `MPI_NEIGHBOR_ALLTOALLV_INIT`, and `MPI_NEIGHBOR_ALLGATHERV_INIT`.

6 B.3 Changes from Version 3.0 to Version 3.1

8 B.3.1 Fixes to Errata in Previous Versions of MPI

- 10 1. Chapters [3–19](#), Annex [A.4](#) on page [879](#), and Example [6.21](#) on page [226](#), and MPI-3.0
11 Chapters [3–17](#), Annex [A.3](#) on page [707](#), and Example [5.21](#) on page [187](#).
12 Within the `mpi_f08` Fortran support method, `BIND(C)` was removed from all
13 `SUBROUTINE`, `FUNCTION`, and `ABSTRACT INTERFACE` definitions.
- 14 2. Section [3.2.5](#) on page [36](#), and MPI-3.0 Section [3.2.5](#) on page [30](#).
15 The three public fields `MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR` of the Fortran derived
16 type `TYPE(MPI_Status)` must be of type `INTEGER`.
- 17 3. Section [3.8.2](#) on page [83](#), and MPI-3.0 Section [3.8.2](#) on page [67](#).
18 The flag arguments of the Fortran interfaces of `MPI_IMPROBE` were originally incor-
19 rectly defined as `INTEGER` (instead as `LOGICAL`).
- 20 4. Section [7.4.2](#) on page [307](#), and MPI-3.0 Section [6.4.2](#) on page [237](#).
21 In the `mpi_f08` binding of `MPI_COMM_IDUP`, the output argument
22 `newcomm` is declared as `ASYNCHRONOUS`.
- 23 5. Section [7.4.4](#) on page [324](#), and MPI-3.0 Section [6.4.4](#) on page [248](#).
24 In the `mpi_f08` binding of `MPI_COMM_SET_INFO`, the `INTENT` of `comm` is `IN`, and the
25 optional output argument `ierror` was missing.
- 26 6. Section [8.6](#) on page [394](#), and MPI-3.0 Sections [7.6](#), on pages [314](#).
27 In the case of virtual general graph topologies (created with `MPI_CART_CREATE`), the
28 use of neighborhood collective communication is restricted to adjacency matrices with
29 the number of edges between any two processes is defined to be the same for both
30 processes (i.e., with a symmetric adjacency matrix).
- 31 7. Section [9.1.1](#) on page [425](#), and MPI-3.0 Section [8.1.1](#) on page [335](#).
32 In the `mpi_f08` binding of `MPI_GET_LIBRARY_VERSION`, a typo in the
33 `resultlen` argument was corrected.
- 34 8. Sections [9.2](#) (`MPI_ALLOC_MEM` and `MPI_ALLOC_MEM_CPTR`),
35 [12.2.2](#) (`MPI_WIN_ALLOCATE` and `MPI_WIN_ALLOCATE_CPTR`),
36 [12.2.3](#) (`MPI_WIN_ALLOCATE_SHARED` and `MPI_WIN_ALLOCATE_SHARED_CPTR`),
37 [12.2.3](#) (`MPI_WIN_SHARED_QUERY` and `MPI_WIN_SHARED_QUERY_CPTR`),
38 [15.2.1](#) and [15.2.6](#) (Profiling interface), and corresponding sections in MPI-3.0.
39 The linker name concept was substituted by defining specific procedure names.
- 40 9. Section [12.2.1](#) on page [522](#), and MPI-3.0 Section [11.2.2](#) on page [407](#).
41 The "same_size" info key can be used with all window flavors, and requires that all
42 processes in the process group of the communicator have provided this info key with
43 the same value.

10. Section [12.3.4](#) on page [545](#), 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. 1
2
3
4
11. Section [12.3.4](#) on page [545](#), 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`. 5
6
7
12. Section [15.3](#) on page [695](#), 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. 8
9
10
11
13. Section [15.3.3](#) on page [697](#), and MPI-3.0 Section 14.3.3 on page 563. New paragraph clarifying variable name equivalence in the tools information interface. 12
13
14
14. Sections [15.3.6](#), [15.3.7](#), and [15.3.9](#) on pages [701](#), [708](#), and [736](#), 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
16
17
18
19
20
15. Section [15.3.7](#) on page [708](#), and MPI-3.0 Section 14.3.7 on page 573. Clarify return code of `MPI_T_PVAR_{START,STOP,RESET}` routines. 21
22
23
16. Section [15.3.7](#) on page [708](#), 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. 24
25
26
17. Section [19.1.4](#) on page [761](#), 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. 27
28
29
18. Section [19.1.5](#) on page [762](#), 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. 30
31
32
33
19. Section [19.1.6](#) on page [767](#), and MPI-3.0 Section 17.1.6 on page 611. The requirements on `BIND(C)` procedure interfaces were removed. 34
35
36
20. Annexes [A.3](#), [A.4](#), and [A.5](#) on pages [842](#), [879](#), and [966](#), 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. 37
38
39
40
21. Annex [A.4.5](#) on page [919](#), 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. 41
42
43
44
45
46
47
48

B.3.2 Changes in MPI-3.1

1. Sections [2.6.4](#) and [5.1.5](#) on pages [24](#) and [134](#).
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 [9.1.1](#), [11.2.1](#), and [11.6](#) on pages [425](#), [462](#), and [489](#).
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 [12.2.1](#) on page [522](#).
The “`same_disp_unit`” info key was added for use in RMA window creation routines.
4. Sections [14.4.2](#) and [14.4.3](#) on pages [628](#) and [635](#).
Added `MPI_FILE_IREAD_AT_ALL`, `MPI_FILE_IWRITE_AT_ALL`, `MPI_FILE_IREAD_ALL`, and `MPI_FILE_IWRITE_ALL`.
5. Sections [15.3.6](#), [15.3.7](#), and [15.3.9](#) on pages [701](#), [708](#), and [736](#).
Clarified that NULL parameters can be provided in `MPI_T_{CVAR|PVAR|CATEGORY}_GET_INFO` routines.
6. Sections [15.3.6](#), [15.3.7](#), [15.3.9](#), and [15.3.10](#) on pages [701](#), [708](#), [736](#), and [740](#).
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.4 Changes from Version 2.2 to Version 3.0

B.4.1 Fixes to Errata in Previous Versions of MPI

1. Sections [2.6.2](#) and [2.6.3](#) on pages [22](#) and [23](#), 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](#), [6.9.2](#), [14.5.2](#) Table [14.2](#), and Annex [A.1.1](#) on pages [31](#), [214](#), [667](#), and [817](#), 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 nonstandard C++ types `Complex<...>` were substituted by the standard types `std::complex<...>`.

3. Sections [6.9.2](#) on pages [214](#) and MPI-2.2 Section 5.9.2, page 165, line 47. 1
This is an MPI-2.2 erratum: MPI_C_COMPLEX was added to the “Complex” reduction group. 2
4. Section [8.5.5](#) on page [381](#), and MPI-2.2, Section 7.5.5 on page 257, C++ interface on page 264, line 3. 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. 4
5. Section [14.5.2](#), Table [14.2](#) on page [667](#), and MPI-2.2, Section 13.5.3, Table 13.2 on page 433. 5
This was an MPI-2.2 erratum: The MPI_C_BOOL “external32” representation is corrected to a 1-byte size. 6
6. MPI-2.2 Section 16.1.16 on page 471, line 45. 7
This is an MPI-2.2 erratum: The constant MPI::LONG_LONG should be MPI::LONG_LONG. 8
7. Annex [A.1.1](#) on page [817](#), Table “Optional datatypes (Fortran),” and MPI-2.2, Annex A.1.1, Table on page 517, lines 34, and 37–41. 9
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. 10

B.4.2 Changes in MPI-3.0 11

1. Section [2.6.1](#) on page [21](#), Section [17.2](#) on page [752](#) and all other chapters. 12
The C++ bindings were removed from the standard. See errata in Section [B.4.1](#) on page [1008](#) for the latest changes to the MPI C++ binding defined in MPI-2.2. 13
This change may affect backward compatibility. 14
2. Section [2.6.1](#) on page [21](#), Section [16.1](#) on page [743](#) and Section [17.1](#) on page [751](#). 15
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. 16
This change may affect backward compatibility. 17
3. Section [2.3](#) on page [10](#). 18
Clarified parameter usage for IN parameters. C bindings are now const-correct where backward compatibility is preserved. 19
4. Section [2.5.4](#) on page [18](#) and Section [8.5.4](#) on page [374](#). 20
The recommended C implementation value for MPI_UNWEIGHTED changed from NULL to non-NULL. An additional weight array constant (MPI_WEIGHTS_EMPTY) was introduced. 21

- 1 5. Section 2.5.4 on page 18 and Section 9.1.1 on page 425.
2 Added the new routine MPI_GET_LIBRARY_VERSION to query library specific ver-
3 sions, and the new constant MPI_MAX_LIBRARY_VERSION_STRING.
4
- 5 6. Sections 2.5.8, 3.2.2, 3.3, 6.9.2, on pages 20, 31, 33, 214, Sections 5.1, 5.1.7, 5.1.8,
6 5.1.11, 13.3 on pages 113, 140, 142, 146, 606, and Annex A.1.1 on page 817.
7 New inquiry functions, MPI_TYPE_SIZE_X, MPI_TYPE_GET_EXTENT_X,
8 MPI_TYPE_GET_TRUE_EXTENT_X, and MPI_GET_ELEMENTS_X, return their re-
9 sults as an MPI_Count value, which is a new type large enough to represent ele-
10 ment counts in memory, file views, etc. A new function,
11 MPI_STATUS_SET_ELEMENTS_X, modifies the opaque part of an MPI_Status object
12 so that a call to MPI_GET_ELEMENTS_X returns the provided MPI_Count value (in
13 Fortran, INTEGER(KIND=MPI_COUNT_KIND)). The corresponding predefined datatype is
14 MPI_COUNT.
- 15 7. Chapter 3 on page 29 through Chapter 19 on page 755.
16 In the C language bindings, the array-arguments' interfaces were modified to consis-
17 tently use use [] instead of *.
18 Exceptions are MPI_INIT, which continues to use char ***argv (correct because of
19 subtle rules regarding the use of the & operator with char *argv[]), and
20 MPI_INIT_THREAD, which is changed to be consistent with MPI_INIT.
21
- 22 8. Sections 3.2.5, 5.1.5, 5.1.11, 5.2 on pages 36, 134, 146, 166.
23 The functions MPI_GET_COUNT and MPI_GET_ELEMENTS were defined to set the
24 count argument to MPI_UNDEFINED when that argument would overflow. The func-
25 tions MPI_PACK_SIZE and MPI_TYPE_SIZE were defined to set the size argument
26 to MPI_UNDEFINED when that argument would overflow. In all other MPI-2.2 rou-
27 tines, the type and semantics of the count arguments remain unchanged, i.e., int or
28 INTEGER.
- 29 9. Section 3.2.6 on page 39, and Section 3.8 on page 80.
30 MPI_STATUS_IGNORE can be also used in MPI_IPROBE, MPI_PROBE, MPI_IMPROBE,
31 and MPI_MPROBE.
32
- 33 10. Section 3.8 on page 80 and Section 3.10 on page 96.
34 The use of MPI_PROC_NULL in probe operations was clarified. A special predefined
35 message MPI_MESSAGE_NO_PROC was defined for the use of matching probe (i.e., the
36 new MPI_MPROBE and MPI_IMPROBE) with MPI_PROC_NULL.
37
- 38 11. Sections 3.8.2, 3.8.3, 19.3.4, A.1.1 on pages 83, 85, 803, 817.
39 Like MPI_PROBE and MPI_IPROBE, the new MPI_MPROBE and
40 MPI_IMPROBE operations allow incoming messages to be queried without actually
41 receiving them, except that MPI_MPROBE and MPI_IMPROBE provide a mechanism
42 to receive the specific message with the new routines MPI_MRECV and
43 MPI_IMRECV regardless of other intervening probe or receive operations. The opaque
44 object MPI_Message, the null handle MPI_MESSAGE_NULL, and the conversion functions
45 MPI_Message_c2f and MPI_Message_f2c were defined.
- 46 12. Section 5.1.2 on page 115 and Section 5.1.13 on page 150.
47 The routine MPI_TYPE_CREATE_HINDEXED_BLOCK and constant
48 MPI_COMBINER_HINDEXED_BLOCK were added.

13. Chapter 6 on page 177 and Section 6.12 on page 237. 1
 Added nonblocking interfaces to all collective operations. 2
14. Sections 7.4.2, 7.4.4, 12.2.7, on pages 307, 324, 537. 3
 The new routines MPI_COMM_DUP_WITH_INFO, MPI_COMM_SET_INFO, 4
 MPI_COMM_GET_INFO, MPI_WIN_SET_INFO, and MPI_WIN_GET_INFO were 5
 added. The routine MPI_COMM_DUP must also duplicate info hints. 6
15. Section 7.4.2 on page 307. 7
 Added MPI_COMM_IDUP. 8
16. Section 7.4.2 on page 307. 9
 Added the new communicator construction routine MPI_COMM_CREATE_GROUP, 10
 which is invoked only by the processes in the group of the new communicator being 11
 constructed. 12
17. Section 7.4.2 on page 307. 13
 Added the MPI_COMM_SPLIT_TYPE routine and the communicator split type con- 14
 stant MPI_COMM_TYPE_SHARED. 15
18. Section 7.6.2 on page 337. 16
 In MPI-2.2, communication involved in an MPI_INTERCOMM_CREATE operation 17
 could interfere with point-to-point communication on the parent communicator with 18
 the same tag or MPI_ANY_TAG. This interference has been removed in MPI-3.0. 19
19. Section 7.8 on page 360. 20
 Section 6.8 on page 238. The constant MPI_MAX_OBJECT_NAME also applies for type 21
 and window names. 22
20. Section 8.5.8 on page 392. 23
 MPI_CART_MAP can also be used for a zero-dimensional topologies. 24
21. Section 8.6 on page 394 and Section 8.7 on page 406. 25
 The following neighborhood collective communication routines were added to sup- 26
 port sparse communication on virtual topology grids: MPI_NEIGHBOR_ALLGATHER, 27
 MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL, 28
 MPI_NEIGHBOR_ALLTOALLV, MPI_NEIGHBOR_ALLTOALLW and the nonblocking 29
 variants MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV, 30
 MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and 31
 MPI_INEIGHBOR_ALLTOALLW. The displacement arguments in 32
 MPI_NEIGHBOR_ALLTOALLW and MPI_INEIGHBOR_ALLTOALLW were defined as 33
 address size integers. In MPI_DIST_GRAPH_NEIGHBORS, an ordering rule was added 34
 for communicators created with MPI_DIST_GRAPH_CREATE_ADJACENT. 35
22. Section 11.2.1 on page 462 and Section 11.2.1 on page 465. 36
 The use of MPI_INIT, MPI_INIT_THREAD and MPI_FINALIZE was clarified. After 37
 MPI is initialized, the application can access information about the execution envi- 38
 ronment by querying the new predefined info object MPI_INFO_ENV. 39
23. Section 11.2.1 on page 462. 40
 Allow calls to MPI_T routines before MPI_INIT and after MPI_FINALIZE. 41

- 1 24. Chapter 12 on page 521.
2 Substantial revision of the entire One-sided chapter, with new routines for window
3 creation, additional synchronization methods in passive target communication, new
4 one-sided communication routines, a new memory model, and other changes.
5
- 6 25. Section 15.3 on page 695.
7 A new MPI Tool Information Interface was added.
8 The following changes are related to the Fortran language support.
9
- 10 26. Section 2.3 on page 10, and Sections 19.1.1, 19.1.2, 19.1.7 on pages 755, 756, and 770.
11 The new `mpi_08` Fortran module was introduced.
- 12 27. Section 2.5.1 on page 15, and Sections 19.1.2, 19.1.3, 19.1.7 on pages 756, 759, and 770.
13 Handles to opaque objects were defined as named types within the `mpi_08` Fortran
14 module. The operators `.EQ.`, `.NE.`, `==`, and `/=` were overloaded to allow the comparison
15 of these handles. The handle types and the overloaded operators are also available
16 through the `mpi` Fortran module.
17
- 18 28. Sections 2.5.4, 2.5.5 on pages 18, 19, Sections 19.1.1, 19.1.10, 19.1.11, 19.1.12, 19.1.13
19 on pages 755, 780, 782, 783, 785, and Sections 19.1.2, 19.1.3, 19.1.7 on pages 756, 759,
20 770.
21 Within the `mpi_08` Fortran module, choice buffers were defined as assumed-type and
22 assumed-rank according to Fortran 2008 with TS 29113 [46], and the compile-time
23 constant `MPI_SUBARRAYS_SUPPORTED` was set to `.TRUE.`. With this, Fortran sub-
24 script triplets can be used in nonblocking MPI operations; vector subscripts are not
25 supported in nonblocking operations. If the compiler does not support this Fortran
26 TS 29113 feature, the constant is set to `.FALSE.`.
27
- 28 29. Section 2.6.2 on page 22, Section 19.1.2 on page 756, and Section 19.1.7 on page 770.
29 The `ieror` dummy arguments are `OPTIONAL` within the `mpi_08` Fortran module.
- 30 30. Section 3.2.5 on page 36, Sections 19.1.2, 19.1.3, 19.1.7, on pages 756, 759, 770, and
31 Section 19.3.5 on page 805.
32 Within the `mpi_08` Fortran module, the status was defined as `TYPE(MPI_Status)`. Ad-
33 ditionally, within both the `mpi` and the `mpi_f08` modules, the constants
34 `MPI_STATUS_SIZE`, `MPI_SOURCE`, `MPI_TAG`, `MPI_ERROR`, and `TYPE(MPI_Status)` are de-
35 fined. New conversion routines were added: `MPI_STATUS_F2F08`,
36 `MPI_STATUS_F082F`, `MPI_Status_c2f08`, and `MPI_Status_f082c`. In `mpi.h`, the new
37 type `MPI_F08_status`, and the external variables `MPI_F08_STATUS_IGNORE` and
38 `MPI_F08_STATUSES_IGNORE` were added.
39
- 40 31. Section 3.6 on page 55.
41 In Fortran with the `mpi` module or `mpif.h`, the type of the `buffer_addr` argument of
42 `MPI_BUFFER_DETACH` is incorrectly defined and the argument is therefore unused.
43
- 44 32. Section 5.1 on page 113, Section 5.1.6 on page 138, and Section 19.1.15 on page 786.
45 The Fortran alignments of basic datatypes within Fortran derived types are imple-
46 mentation dependent; therefore it is recommended to use the `BIND(C)` attribute for
47 derived types in MPI communication buffers. If an array of structures (in C/C++)
48

- 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 [5.1.10](#), [6.9.5](#), [6.9.7](#), [7.7.4](#), [7.8](#), [9.3.1](#), [9.3.2](#), [9.3.3](#), [16.1](#), [19.1.9](#) on pages [145](#), [221](#), [228](#), [354](#), [360](#), [434](#), [436](#), [438](#), [743](#), and [773](#). 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 [7.7.2](#) on page [345](#).
It was clarified that in Fortran, the flag values returned by a `comm_copy_attr_fn` callback, including `MPI_COMM_NULL_COPY_FN` and `MPI_COMM_DUP_FN`, are `.FALSE.` and `.TRUE.`; see `MPI_COMM_CREATE_KEYVAL`.
 35. Section [9.2](#) on page [429](#).
With the `mpi` and `mpi_f08` Fortran modules, `MPI_ALLOC_MEM` now also supports `TYPE(C_PTR)` C-pointers instead of only returning an address-sized integer that may be usable together with a nonstandard Cray-pointer.
 36. Section [19.1.15](#) on page [786](#), and Section [19.1.7](#) on page [770](#).
Fortran `SEQUENCE` and `BIND(C)` derived application types can now be used as buffers in MPI operations.
 37. Section [19.1.16](#) on page [788](#) to Section [19.1.19](#) on page [798](#), Section [19.1.7](#) on page [770](#), and Section [19.1.8](#) on page [772](#).
The sections about Fortran optimization problems and their solutions were partially rewritten and new methods are added, e.g., the use of the `ASYNCHRONOUS` attribute. The constant `MPI_ASYNC_PROTECTS_NONBLOCKING` tells whether the semantics of the `ASYNCHRONOUS` attribute is extended to protect nonblocking operations. The Fortran routine `MPI_F_SYNC_REG` is added. MPI-3.0 compliance for an MPI library together with a Fortran compiler is defined in Section [19.1.7](#).
 38. Section [19.1.2](#) on page [756](#).
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 [19.1.3](#) on page [759](#), and Section [19.1.7](#) on page [770](#).
The existing `mpi` Fortran module must implement compile-time argument checking.

- 1 40. Section [19.1.4](#) on page [761](#).
2 The use of the `mpif.h` Fortran include file is now strongly discouraged.
- 3
4 41. Section [A.1.1](#), Table **Predefined functions** on page [825](#), Section [A.1.3](#) on page [832](#),
5 and Section [A.4.5](#) on page [919](#).
6 Within the new `mpi_f08` module, all callback prototype definitions are now defined
7 with explicit interfaces `PROCEDURE(MPI_...)` that have the `BIND(C)` attribute; user-
8 written callbacks must be modified if the `mpi_f08` module is used.
- 9
10 42. Section [A.1.3](#) on page [832](#).
11 In some routines, the Fortran callback prototype names were changed from `..._FN` to
12 `..._FUNCTION` to be consistent with the other language bindings.

13 B.5 Changes from Version 2.1 to Version 2.2

- 14
15
16 1. Section [2.5.4](#) on page [18](#).
17 It is now guaranteed that predefined named constant handles (as other constants)
18 can be used in initialization expressions or assignments, i.e., also before the call to
19 `MPI_INIT`.
- 20
21 2. Section [2.6](#) on page [21](#), and Section [17.2](#) on page [752](#).
22 The C++ language bindings have been deprecated and may be removed in a future
23 version of the MPI specification.
- 24
25 3. Section [3.2.2](#) on page [31](#).
26 `MPI_CHAR` for printable characters is now defined for C type `char` (instead of signed
27 `char`). This change should not have any impact on applications nor on MPI libraries
28 (except some comment lines), because printable characters could and can be stored in
29 any of the C types `char`, `signed char`, and `unsigned char`, and `MPI_CHAR` is not allowed
30 for predefined reduction operations.
- 31
32 4. Section [3.2.2](#) on page [31](#).
33 `MPI_(U)INT{8,16,32,64}_T`, `MPI_AINT`, `MPI_OFFSET`, `MPI_C_BOOL`,
34 `MPI_C_COMPLEX`, `MPI_C_FLOAT_COMPLEX`, `MPI_C_DOUBLE_COMPLEX`, and
35 `MPI_C_LONG_DOUBLE_COMPLEX` are now valid predefined MPI datatypes.
- 36
37 5. Section [3.4](#) on page [46](#), Section [3.7.2](#) on page [60](#), Section [3.9](#) on page [90](#), and Section [6.1](#)
38 on page [177](#).
39 The read access restriction on the send buffer for blocking, non blocking and collective
40 API has been lifted. It is permitted to access for read the send buffer while the
41 operation is in progress.
- 42
43 6. Section [3.7](#) on page [58](#).
44 The Advice to users for `IBSEND` and `IRSEND` was slightly changed.
- 45
46 7. Section [3.7.3](#) on page [67](#).
47 The advice to free an active request was removed in the Advice to users for
48 `MPI_REQUEST_FREE`.
- 49
50 8. Section [3.7.6](#) on page [79](#).
51 `MPI_REQUEST_GET_STATUS` changed to permit inactive or null requests as input.

9. Section 6.8 on page 206. 1
 “In place” option is added to MPI_ALLTOALL, MPI_ALLTOALLV, and
 MPI_ALLTOALLW for intra-communicators. 2
3
4
10. Section 6.9.2 on page 214. 5
 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. 6
7
8
11. Section 6.9.2 on page 214. 9
 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. 10
 MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, and 11
 MPI_C_LONG_DOUBLE_COMPLEX are considered Complex types. 12
13
14
15
12. Section 6.9.7 on page 228. 16
 The local routines MPI_REDUCE_LOCAL and MPI_OP_COMMUTATIVE have been
 added. 17
18
19
13. Section 6.10.1 on page 230. 20
 The collective function MPI_REDUCE_SCATTER_BLOCK is added to the MPI stan-
 dard. 21
22
23
14. Section 6.11.2 on page 234. 24
 Added in place argument to MPI_EXSCAN. 25
15. Section 7.4.2 on page 307, and Section 7.6 on page 333. 26
 Implementations that did not implement MPI_COMM_CREATE on inter-communi-
 cators will need to add that functionality. As the standard described the behav-
 ior of this operation on inter-communicators, it is believed that most implementa-
 tions already provide this functionality. Note also that the C++ binding for both
 MPI_COMM_CREATE and MPI_COMM_SPLIT explicitly allow Intercomms. 27
28
29
30
31
32
16. Section 7.4.2 on page 307. 33
 MPI_COMM_CREATE is extended to allow several disjoint subgroups as input if comm
 is an intra-communicator. If comm is an inter-communicator it was clarified that all
 processes in the same local group of comm must specify the same value for group. 34
35
36
37
17. Section 8.5.4 on page 374. 38
 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. 39
40
41
42
18. Section 8.5.5 on page 381. 43
 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. 44
45
46
47
48

- 1 19. Section [8.5.5](#) on page [381](#).
2 Remove ambiguity regarding duplicated neighbors with `MPI_GRAPH_NEIGHBORS`
3 and `MPI_GRAPH_NEIGHBORS_COUNT`.
4
- 5 20. Section [9.1.1](#) on page [425](#).
6 The subversion number changed from 1 to 2.
7
- 8 21. Section [9.3](#) on page [432](#), Section [16.2](#) on page [746](#), and Annex [A.1.3](#) on page [832](#).
9 Changed function pointer typedef names `MPI_{Comm,File,Win}_errhandler_fn` to
10 `MPI_{Comm,File,Win}_errhandler_function`. Deprecated old “_fn” names.
11
- 12 22. Section [11.2.4](#) on page [472](#).
13 Attribute deletion callbacks on `MPI_COMM_SELF` are now called in LIFO order. Imple-
14 mentors must now also register all implementation-internal attribute deletion callbacks
15 on `MPI_COMM_SELF` before returning from `MPI_INIT/MPI_INIT_THREAD`.
16
- 17 23. Section [12.3.4](#) on page [545](#).
18 The restriction added in MPI 2.1 that the operation `MPI_REPLACE` in
19 `MPI_ACCUMULATE` can be used only with predefined datatypes has been removed.
20 `MPI_REPLACE` can now be used even with derived datatypes, as it was in MPI 2.0.
21 Also, a clarification has been made that `MPI_REPLACE` can be used only in
22 `MPI_ACCUMULATE`, not in collective operations that do reductions, such as
23 `MPI_REDUCE` and others.
24
- 25 24. Section [13.2](#) on page [599](#).
26 Add “*” to the `query_fn`, `free_fn`, and `cancel_fn` arguments to the C++ binding for
27 `MPI::Grequest::Start()` for consistency with the rest of MPI functions that take function
28 pointer arguments.
29
- 30 25. Section [14.5.2](#) on page [665](#), and Table [14.2](#) on page [667](#).
31 `MPI_(U)INT{8,16,32,64}_T`, `MPI_AINT`, `MPI_OFFSET`, `MPI_C_COMPLEX`,
32 `MPI_C_FLOAT_COMPLEX`, `MPI_C_DOUBLE_COMPLEX`,
33 `MPI_C_LONG_DOUBLE_COMPLEX`, and `MPI_C_BOOL` are added as predefined datatypes
34 in the “external32” representation.
35
- 36 26. Section [19.3.7](#) on page [810](#).
37 The description was modified that it only describes how an MPI implementation be-
38 haves, but not how MPI stores attributes internally. The erroneous MPI-2.1 Example
39 16.17 was replaced with three new examples [19.13](#), [19.14](#), and [19.15](#) on pages [811–812](#)
40 explicitly detailing cross-language attribute behavior. Implementations that matched
41 the behavior of the old example will need to be updated.
42
- 43 27. Annex [A.1.1](#) on page [817](#).
44 Removed type `MPI::Fint` (compare `MPI_Fint` in Section [A.1.2](#) on page [831](#)).
45
- 46 28. Annex [A.1.1](#) on page [817](#). Table **Named Predefined Datatypes**.
47 Added `MPI_(U)INT{8,16,32,64}_T`, `MPI_AINT`, `MPI_OFFSET`, `MPI_C_BOOL`,
48 `MPI_C_FLOAT_COMPLEX`, `MPI_C_COMPLEX`, `MPI_C_DOUBLE_COMPLEX`, and
`MPI_C_LONG_DOUBLE_COMPLEX` are added as predefined datatypes.

B.6 Changes from Version 2.0 to Version 2.1

1. Section 3.2.2 on page 31, and Annex A.1 on page 817.
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 31, and Annex A.1 on page 817.
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 36.
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 5.1 on page 113.
General rule about derived datatypes: Most datatype constructors have replication count or block length arguments. Allowed values are nonnegative integers. If the value is zero, no elements are generated in the type map and there is no effect on datatype bounds or extent.
5. Section 5.3 on page 173.
MPI_BYTE should be used to send and receive data that is packed using MPI_PACK_EXTERNAL.
6. Section 6.9.6 on page 226.
If comm is an inter-communicator in MPI_ALLREDUCE, then both groups should provide count and datatype arguments that specify the same type signature (i.e., it is not necessary that both groups provide the same count value).
7. Section 7.3.1 on page 296.
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 7.7 on page 343.
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 7.8 on page 360.
In MPI_COMM_GET_NAME: In C, a null character is additionally stored at

1 name[resultlen]. resultlen cannot be larger than MPI_MAX_OBJECT_NAME-1. In For-
2 tran, name is padded on the right with blank characters. resultlen cannot be larger
3 than MPI_MAX_OBJECT_NAME.
4

- 5 10. Section 8.4 on page 369.

6 About MPI_GRAPH_CREATE and MPI_CART_CREATE: All input arguments must
7 have identical values on all processes of the group of comm_old.

- 8 11. Section 8.5.1 on page 370.

9 In MPI_CART_CREATE: If ndims is zero then a zero-dimensional Cartesian topology
10 is created. The call is erroneous if it specifies a grid that is larger than the group size
11 or if ndims is negative.
12

- 13 12. Section 8.5.3 on page 372.

14 In MPI_GRAPH_CREATE: If the graph is empty, i.e., nnodes = 0, then
15 MPI_COMM_NULL is returned in all processes.

- 16 13. Section 8.5.3 on page 372.

17 In MPI_GRAPH_CREATE: A single process is allowed to be defined multiple times
18 in the list of neighbors of a process (i.e., there may be multiple edges between two
19 processes). A process is also allowed to be a neighbor to itself (i.e., a self loop in the
20 graph). The adjacency matrix is allowed to be nonsymmetric.
21

22 *Advice to users.* Performance implications of using multiple edges or a nonsym-
23 metric adjacency matrix are not defined. The definition of a node-neighbor edge
24 does not imply a direction of the communication. (*End of advice to users.*)

- 25 14. Section 8.5.5 on page 381.

26 In MPI_CARTDIM_GET and MPI_CART_GET: If comm is associated with a zero-
27 dimensional Cartesian topology, MPI_CARTDIM_GET returns ndims=0 and
28 MPI_CART_GET will keep all output arguments unchanged.
29

- 30 15. Section 8.5.5 on page 381.

31 In MPI_CART_RANK: If comm is associated with a zero-dimensional Cartesian topol-
32 ogy, coord is not significant and 0 is returned in rank.

- 33 16. Section 8.5.5 on page 381.

34 In MPI_CART_COORDS: If comm is associated with a zero-dimensional Cartesian
35 topology, coords will be unchanged.
36

- 37 17. Section 8.5.6 on page 389.

38 In MPI_CART_SHIFT: It is erroneous to call MPI_CART_SHIFT with a direction that is
39 either negative or greater than or equal to the number of dimensions in the Cartesian
40 communicator. This implies that it is erroneous to call MPI_CART_SHIFT with a
41 comm that is associated with a zero-dimensional Cartesian topology.
42

- 43 18. Section 8.5.7 on page 391.

44 In MPI_CART_SUB: If all entries in remain_dims are false or comm is already associated
45 with a zero-dimensional Cartesian topology then newcomm is associated with a zero-
46 dimensional Cartesian topology.

- 47 18.1. Section 9.1.1 on page 425.

48 The subversion number changed from 0 to 1.

19. Section 9.1.2 on page 426. 1
 In MPI_GET_PROCESSOR_NAME: In C, a null character is additionally stored at 2
 name[resultlen]. resultlen cannot be larger than MPI_MAX_PROCESSOR_NAME-1. In 3
 Fortran, name is padded on the right with blank characters. resultlen cannot be larger 4
 than MPI_MAX_PROCESSOR_NAME. 5
20. Section 9.3 on page 432. 6
 MPI_{COMM,WIN,FILE}_GET_ERRHANDLER behave as if a new error handler object 7
 is created. That is, once the error handler is no longer needed, 8
 MPI_ERRHANDLER_FREE should be called with the error handler returned from 9
 MPI_ERRHANDLER_GET or MPI_{COMM,WIN,FILE}_GET_ERRHANDLER to mark 10
 the error handler for deallocation. This provides behavior similar to that of 11
 MPI_COMM_GROUP and MPI_GROUP_FREE. 12
21. Section 11.2.1 on page 462, see explanations to MPI_FINALIZE. 13
 MPI_FINALIZE is collective over all connected processes. If no processes were spawned, 14
 accepted or connected then this means over MPI_COMM_WORLD; otherwise it is col- 15
 lective over the union of all processes that have been and continue to be connected, 16
 as explained in Section 11.10.4 on page 517. 17
22. Section 11.2.1 on page 462. 18
 About MPI_ABORT: 19
- Advice to users.* Whether the errorcode is returned from the executable or from 20
 the MPI process startup mechanism (e.g., mpiexec), is an aspect of quality of the 21
 MPI library but not mandatory. (*End of advice to users.*) 22
- Advice to implementors.* Where possible, a high-quality implementation will try 23
 to return the errorcode from the MPI process startup mechanism (e.g. mpiexec 24
 or singleton init). (*End of advice to implementors.*) 25
23. Section 10 on page 453. 26
 An implementation must support info objects as caches for arbitrary (key, value) 27
 pairs, regardless of whether it recognizes the key. Each function that takes hints in 28
 the form of an MPI_Info must be prepared to ignore any key it does not recognize. This 29
 description of info objects does not attempt to define how a particular function should 30
 react if it recognizes a key but not the associated value. MPI_INFO_GET_NKEYS, 31
 MPI_INFO_GET_NTHKEY, MPI_INFO_GET_VALUELEN, and MPI_INFO_GET must 32
 retain all (key,value) pairs so that layered functionality can also use the Info object. 33
24. Section 12.3 on page 539. 34
 MPI_PROC_NULL is a valid target rank in the MPI RMA calls MPI_ACCUMULATE, 35
 MPI_GET, and MPI_PUT. The effect is the same as for MPI_PROC_NULL in MPI point- 36
 to-point communication. See also item 25 in this list. 37
25. Section 12.3 on page 539. 38
 After any RMA operation with rank MPI_PROC_NULL, it is still necessary to finish the 39
 RMA epoch with the synchronization method that started the epoch. See also item 24 40
 in this list. 41

- 1 26. Section 12.3.4 on page 545.
2 MPI_REPLACE in MPI_ACCUMULATE, like the other predefined operations, is defined
3 only for the predefined MPI datatypes.
4
- 5 27. Section 14.2.8 on page 618.
6 About MPI_FILE_SET_VIEW and MPI_FILE_SET_INFO: When an info object that
7 specifies a subset of valid hints is passed to MPI_FILE_SET_VIEW or
8 MPI_FILE_SET_INFO, there will be no effect on previously set or defaulted hints that
9 the info does not specify.
10
- 11 28. Section 14.2.8 on page 618.
12 About MPI_FILE_GET_INFO: If no hint exists for the file associated with fh, a handle
13 to a newly created info object is returned that contains no key/value pair.
14
- 15 29. Section 14.3 on page 622.
16 If a file does not have the mode MPI_MODE_SEQUENTIAL, then
17 MPI_DISPLACEMENT_CURRENT is invalid as disp in MPI_FILE_SET_VIEW.
18
- 19 30. Section 14.5.2 on page 665.
20 The bias of 16 byte doubles was defined with 10383. The correct value is 16383.
21
- 22 31. MPI-2.2, Section 16.1.4 (Section was removed in MPI-3.0).
23 In the example in this section, the buffer should be declared as `const void* buf`.
24
- 25 32. Section 19.1.9 on page 773.
26 About MPI_TYPE_CREATE_F90_XXX:
27 *Advice to implementors.* An application may often repeat a call to
28 MPI_TYPE_CREATE_F90_XXX with the same combination of (XXX,p,r). The
29 application is not allowed to free the returned predefined, unnamed datatype
30 handles. To prevent the creation of a potentially huge amount of handles, the
31 MPI implementation should return the same datatype handle for the same (
32 REAL/COMPLEX/INTEGER,p,r) combination. Checking for the combination (
33 p,r) in the preceding call to MPI_TYPE_CREATE_F90_XXX and using a hash-
34 table to find formerly generated handles should limit the overhead of finding
35 a previously generated datatype with same combination of (XXX,p,r). (*End of
36 advice to implementors.*)
37
- 38 33. Section A.1.1 on page 817.
39 MPI_BOTTOM is defined as `void * const MPI::BOTTOM`.
40
41
42
43
44
45
46
47
48

Bibliography

- [1] Reverse domain name notation convention. <https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html>. Citation on page 324.
- [2] 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. Citation on page 2.
- [3] V. Bala, S. Kipnis, L. Rudolph, and Marc Snir. Designing efficient, scalable, and portable collective communication libraries. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint. Citation on page 2.
- [4] Purushotham V. Bangalore, Nathan E. Doss, and Anthony Skjellum. MPI++: Issues and features. In *OON-SKI '94*, page in press, 1994. Citation on page 291.
- [5] 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. Citation on page 2.
- [6] 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. Citation on page 2.
- [7] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *IJHPCN*, 1(1/2/3):91–99, 2004. Citation on page 579.
- [8] 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. Citation on page 609.
- [9] R. Butler and E. Lusk. User’s guide to the p4 programming system. Technical Report TM-ANL-92/17, Argonne National Laboratory, 1992. Citation on page 2.
- [10] 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. Citation on page 2.
- [11] 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. Citation on page 2.
- [12] 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. Citation on page 367.

- [13] 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. Citation on page 367.
- [14] Parasoftware Corporation. Express version 1.0: A communication environment for parallel computers, 1988. Citation on page 2.
- [15] 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. Citations on pages 1022 and 1024.
- [16] 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. Citation on page 609.
- [17] 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. [15], pages 282-291. Citation on page 314.
- [18] 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. Citation on page 2.
- [19] 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. Citation on page 2.
- [20] Edinburgh Parallel Computing Centre, University of Edinburgh. *CHIMP Concepts*, June 1991. Citation on page 2.
- [21] Edinburgh Parallel Computing Centre, University of Edinburgh. *CHIMP Version 1.0 Interface*, May 1992. Citation on page 2.
- [22] D. Feitelson. Communicators: Object-based multiparty interactions for parallel programming. Technical Report 91-12, Dept. Computer Science, The Hebrew University of Jerusalem, November 1991. Citation on page 292.
- [23] Message Passing Interface Forum. MPI: A message-passing interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994. Citation on page 2.
- [24] Message Passing Interface Forum. MPI: A message-passing interface standard (version 1.1). Technical report, 1995. <http://www.mpi-forum.org>. Citation on page 2.
- [25] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Bob Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994. Citation on page 461.

- [26] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. PICL: A portable instrumented communications library, C reference manual. Technical Report TM-11130, Oak Ridge National Laboratory, Oak Ridge, TN, July 1990. Citation on page [2](#). 1
2
3
4
- [27] Brice Goglin, Emmanuel Jeannot, Farouk Mansouri, and Guillaume Mercier. Hardware topology management in MPI applications through hierarchical communicators. *Parallel Computing*, 76:70–90, 2018. Citation on page [321](#). 5
6
7
- [28] Ryan E. Grant, Matthew G. F. Dosanjh, Michael J. Levenhagen, Ron Brightwell, and Anthony Skjellum. Finepoints: Partitioned multithreaded MPI communication. In *ISC High Performance Conference (ISC)*, 2019. Citation on page [99](#). 8
9
10
11
- [29] Ryan E. Grant, Anthony Skjellum, and Purushotham V. Bangalore. Lightweight threading with MPI using persistent communications semantics. In *Workshop on Exascale MPI (ExaMPI)*. Held in conjunction with the 2015 International Conference for High Performance Computing, Networking, Storage and Analysis (SC15), 2015. Citation on page [99](#). 12
13
14
15
16
- [30] D. Gregor, T. Hoefler, B. Barrett, and A. Lumsdaine. Fixing probe for multi-threaded MPI applications. Technical Report 674, Indiana University, Jan. 2009. Citation on page [83](#). 17
18
19
20
- [31] William D. Gropp and Barry Smith. Chameleon parallel programming tools users manual. Technical Report ANL-93/23, Argonne National Laboratory, March 1993. Citation on page [2](#). 21
22
23
24
- [32] Michael Hennecke. A Fortran 90 interface to MPI version 1.1. Technical Report Internal Report 63/96, Rechenzentrum, Universität Karlsruhe, D-76128 Karlsruhe, Germany, June 1996. Citation on page [760](#). 25
26
27
- [33] T. Hoefler, G. Bronevetsky, B. Barrett, B. R. de Supinski, and A. Lumsdaine. Efficient MPI support for advanced hybrid programming models. In *Recent Advances in the Message Passing Interface (EuroMPI'10)*, volume LNCS 6305, pages 50–61. Springer, Sep. 2010. Citation on page [83](#). 28
29
30
31
32
- [34] T. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm. Optimizing a conjugate gradient solver with non-blocking collective operations. *Elsevier Journal of Parallel Computing (PARCO)*, 33(9):624–633, Sep. 2007. Citation on page [237](#). 33
34
35
36
- [35] T. Hoefler, F. Lorenzen, and A. Lumsdaine. Sparse non-blocking collectives in quantum mechanical calculations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting*, volume LNCS 5205, pages 55–63. Springer, Sep. 2008. Citation on page [394](#). 37
38
39
40
- [36] T. Hoefler and A. Lumsdaine. Message progression in parallel computing — to thread or not to thread? In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society, Oct. 2008. Citation on page [237](#). 41
42
43
44
- [37] 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. Citation on page [239](#). 45
46
47
48

- [38] 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. Citation on page [237](#).
- [39] T. Hoefler and J. L. Träff. Sparse collective operations for MPI. In *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium, HIPS'09 Workshop*, May 2009. Citation on page [394](#).
- [40] Torsten Hoefler and Marc Snir. Writing parallel libraries with MPI — common practice, issues, and extensions. In Cotronis et al. [[15](#)], pages 345–355. Citation on page [310](#).
- [41] Daniel J. Holmes, Bradley Morgan, Anthony Skjellum, Purushotham V. Bangalore, and Srinivas Sridharan. Planning for performance: Enhancing achievable performance for MPI through persistent collective operations. *Parallel Computing*, 81:32 – 57, 2019. Citation on page [260](#).
- [42] Institute of Electrical and Electronics Engineers, New York. *IEEE Standard for Binary Floating-Point Arithmetic, IEEE Standard 754-2008*, 2008. Citation on page [665](#).
- [43] 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*, February 1987. Citation on page [665](#).
- [44] International Organization for Standardization, Geneva. *ISO/IEC 9945-1:1996: Information technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language]*, December 1996. Citations on pages [489](#) and [613](#).
- [45] International Organization for Standardization, Geneva. *ISO/IEC 1539-1:2010: Information technology — Programming languages — Fortran — Part 1: Base language*, November 2010. Citations on pages [755](#) and [758](#).
- [46] International Organization for Standardization, Geneva. *ISO/IEC TS 29113:2012: Information technology — Further interoperability of Fortran with C*, December 2012. Citations on pages [755](#), [756](#), [758](#), [771](#), [772](#), and [1012](#).
- [47] 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. Citation on page [129](#).
- [48] 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. Citation on page [609](#).
- [49] O. Krämer and H. Mühlenbein. Mapping strategies in message-based multiprocessor systems. *Parallel Computing*, 9:213–225, 1989. Citation on page [367](#).
- [50] S. J. Lefflet, R. S. Fabry, W. N. Joy, P. Lapsley, S. Miller, and C. Torek. An advanced 4.4BSD interprocess communication tutorial, Unix programmer's supplementary documents (PSD) 21. Technical report, Computer Systems Research Group, Department of

- Electrical Engineering and Computer Science, University of California, Berkeley, 1993. Available online: <https://docs.freebsd.org/44doc/psd/21.ipc/paper.pdf>. Citation on page 519.
- [51] Message Passing Interface Forum. Summary of the semantics of all operation-related MPI procedures, 2020. Available online: <https://www.mpi-forum.org/docs>. Citation on page 841.
- [52] Bradley Morgan, Daniel J. Holmes, Anthony Skjellum, Purushotham Bangalore, and Srinivas Sridharan. Planning for performance: Persistent collective operations for MPI. In *Proceedings of the 24th European MPI Users' Group Meeting, EuroMPI '17*, pages 4:1–4:11, New York, NY, USA, 2017. ACM. Citation on page 260.
- [53] nCUBE Corporation. *nCUBE 2 Programmers Guide, r2.0*, December 1990. Citation on page 2.
- [54] Bill Nitzberg. Performance of the iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, December 1992. Citation on page 609.
- [55] William J. Nitzberg. *Collective Parallel I/O*. PhD thesis, Department of Computer and Information Science, University of Oregon, December 1995. Citation on page 609.
- [56] *4.4BSD Programmer's Supplementary Documents (PSD)*. O'Reilly and Associates, 1994. Citation on page 519.
- [57] Paul Pierce. The NX/2 operating system. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 384–390. ACM Press, 1988. Citation on page 2.
- [58] Martin Schulz and Bronis R. de Supinski. P^NMPI tools: A whole lot greater than the sum of their parts. In *ACM/IEEE Supercomputing Conference (SC)*, pages 1–10. ACM, 2007. Citation on page 694.
- [59] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, December 1995. Citation on page 609.
- [60] A. Skjellum and A. Leung. Zipcode: a portable multicomputer communication library atop the reactive kernel. In D. W. Walker and Q. F. Stout, editors, *Proceedings of the Fifth Distributed Memory Concurrent Computing Conference*, pages 767–776. IEEE Press, 1990. Citations on pages 2 and 292.
- [61] A. Skjellum, S. Smith, C. Still, A. Leung, and M. Morari. The Zipcode message passing system. Technical report, Lawrence Livermore National Laboratory, September 1992. Citation on page 2.
- [62] Anthony Skjellum, Nathan E. Doss, and Purushotham V. Bangalore. Writing libraries 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. Citation on page 291.

- 1 [63] Anthony Skjellum, Nathan E. Doss, and Kishore Viswanathan. Inter-communicator ex-
2 tensions to MPI in the MPIX (MPI eXtension) Library. Technical Report MSU-940722,
3 Mississippi State University — Dept. of Computer Science, August 1994. Archived at
4 <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.6283>. Citation on
5 page 181.
- 6
- 7 [64] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P. Leung, and Manfred
8 Morari. The design and evolution of Zipcode. *Parallel Computing*, 20(4):565–596, April
9 1994. Citations on pages 292 and 333.
- 10
- 11 [65] The Internet Society. XDR: External data representation standard, May 2006. [http:
12 //www.rfc-editor.org/pdf/rfc/rfc4506.txt.pdf](http://www.rfc-editor.org/pdf/rfc/rfc4506.txt.pdf). Citation on page 665.
- 13
- 14 [66] Rajeev Thakur and Alok Choudhary. An extended two-phase method for accessing
15 sections of out-of-core arrays. *Scientific Programming*, 5(4):301–317, Winter 1996.
16 Citation on page 609.
- 17
- 18 [67] Jesper Larsson Träff. SMP-aware message passing programming. In *Eighth Inter-
19 national Workshop on High-level Parallel Programming Models and Supportive Envi-
20 ronments (HIPS), 17th International Parallel and Distributed Processing Symposium
21 (IPDPS)*, pages 56–65, 2003. Citation on page 321.
- 22
- 23 [68] *The Unicode Standard, Version 2.0*. Addison-Wesley, 1996. ISBN 0-201-48345-9. Ci-
24 tation on page 665.
- 25
- 26 [69] D. Walker. Standards for message passing in a distributed memory environment. Tech-
27 nical Report TM-12147, Oak Ridge National Laboratory, August 1992. Citation on
28 page 2.
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48

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 page numbers are used for entries appearing in a section or subsection title.

- !(c), [800](#)
- c, [20](#), [21](#), [222](#), [800](#)
- MPI process initialization
 - Sessions Model, [295](#)

- absolute addresses, [19](#), [134](#), [790](#)
- access epoch, [560](#)
- action
 - in function names, [10](#)
- active, [68](#), [69](#), [70](#), [88](#), [95](#), [96](#), [365](#)
- active target communication, [560](#)
- addresses, [149](#)
 - absolute, [19](#), [20](#), [134](#), [790](#)
 - correct use, [149](#)
 - relative displacement, [19](#), [134](#)
- alignment, [430](#), [527](#), [529](#), [1004](#)
- all-reduce, [226](#)
 - nonblocking, [254](#)
 - persistent, [277](#)
- all-to-all, [206](#)
 - nonblocking, [249](#)
 - persistent, [272](#)
- array arguments, [17](#)
- assertions, [575](#)
- ASYNCHRONOUS
 - Fortran attribute, [792](#)
- attribute, [292](#), [343](#), [810](#)
 - caching, [292](#)

- barrier synchronization, [183](#)
 - nonblocking, [239](#)
 - persistent, [262](#)
- basic datatypes, [31](#)
 - additional host language, [31](#)
 - byte, [31](#)
 - C, [32](#)
 - C and Fortran, [33](#)
 - C++, [33](#)
 - Fortran, [31](#)
 - packed, [31](#)
- blocking, [14](#), [46](#), [50](#), [85](#), [625](#)
 - I/O, [626](#)
 - point-to-point, [30](#)
- blocking I/O, [625](#)

- blocking operation, [12](#)
- blocking procedure, [14](#)
- bounds of datatypes, [140](#)
- broadcast, [184](#)
 - nonblocking, [240](#)
 - persistent, [262](#)
- buffered mode send, [47](#), [51](#), [89](#), [92](#)
 - buffer allocation, [55](#)
 - nonblocking, [59](#), [60](#), [61](#)

- C
 - language binding, [23](#)
- caching, [291](#), [292](#), [343](#)
- callback functions
 - language interoperability, [809](#)
 - prototype definitions, [832](#)
 - deprecated, [838](#)
- cancel, [22](#), [69](#), [80](#), [80](#), [81](#), [88](#), [89](#), [470](#), [747](#), [1003](#)
- cancelled, [80](#), [81](#), [88](#), [89](#)
- canonical pack and unpack, [173](#)
- Cartesian
 - topology, [369](#), [370](#)
- Chameleon, [2](#)
- change-log, [1001](#)
- Chimp, [2](#)
- choice, [19](#)
- class
 - in function names, [10](#)
- clock synchronization, [428](#)
- collective, [14](#), [625](#)
 - split, [674](#)
- collective communication, [177](#)
 - correctness, [283](#)
 - file data access operations, [649](#)
 - neighborhood, [394](#)
 - nonblocking, [237](#)
 - progress, [237](#), [239](#), [288](#), [290](#)
- collective operation, [12](#)
- collective procedure, [14](#)
- commit, [144](#)
- COMMON blocks, [795](#)
- communication, [521](#)
 - collective, [177](#)
 - modes, [46](#)

- 1 one-sided, [521](#)
- 2 overlap with communication, [58](#)
- 3 overlap with computation, [58](#)
- 4 partitioned point-to-point, [99](#)
- 5 point-to-point, [29](#)
- 6 RMA, [521](#)
- 7 typed, untyped, and packed, [43](#)
- 8 communication modes, [46](#)
- 9 communicator, [34](#), [291](#), [292](#)
- 10 hidden, [180](#), [286](#), [569](#)
- 11 complete operation, [11](#)
- 12 completing, [13](#)
- 13 completing procedure, [13](#)
- 14 completion, [47](#), [58](#), [67](#), [68](#), [69](#), [72](#), [90](#)
- 15 multiple, [72](#), [73–76](#)
- 16 completion stage, [11](#)
- 17 concurrent, [674](#)
- 18 conflict, [674](#)
- 19 connected, [517](#)
- 20 constants, [18](#), [814](#), [817](#)
- 21 context, [291](#), [292](#), [294](#)
- 22 control variables
- 23 tools interface, [701](#)
- 24 conversion, [45](#)
- 25 representation, [45](#)
- 26 type, [45](#)
- 27 coordination, [625](#)
- 28 counts, [20](#)
- 29 Cray-pointers, [432](#)
- 30 create
- 31 in function names, [9](#)
- 32 data, [31](#)
- 33 data conversion, [45](#)
- 34 datatype, [15](#)
- 35 derived, [15](#)
- 36 equivalent, [15](#)
- 37 named, [15](#)
- 38 portable, [15](#)
- 39 predefined, [15](#)
- 40 unnamed, [15](#)
- 41 datatypes, [113](#), [808](#)
- 42 deadlock avoidance
- 43 cyclic shift, [40](#)
- 44 nonblocking communication, [58](#), [72](#)
- 45 send modes, [54](#)
- 46 default file error, [684](#)
- 47 delete
- 48 in function names, [9](#)
- 49 deprecated interfaces, [21](#), [743](#)
- 50 derived datatype, [15](#), [113](#), [786](#)
- 51 disconnected, [517](#)
- 52 displacement, [609](#), [623](#)
- 53 distributed graph
- 54 topology, [369](#), [374](#)
- 55 Dynamic Process Model, [461](#)
- 56 dynamically attached memory, [531](#)
- 57 elementary datatype, [609](#), [623](#)
- 58 empty, [68](#), [69](#), [73](#)
- 59 enabled operation state
- 60 sufficiently, [68](#)
- 61 end of file, [610](#)
- 62 envelope, [29](#), [30](#), [33](#), [33](#), [34](#), [36](#), [38](#), [46](#), [57](#)
- 63 data representation conversion, [46](#)
- 64 environmental inquiries, [426](#)
- 65 equivalent datatypes, [15](#)
- 66 erroneous program, [35](#)
- 67 freeing active request, [70](#)
- 68 invalid matched receive, [87](#), [88](#)
- 69 lack of buffer space, [53](#)
- 70 ready mode before receive, [47](#)
- 71 type matching, [43](#)
- 72 error classes, [443](#)
- 73 error handling, [24](#), [432](#)
- 74 default file error handler, [611](#), [615](#), [684](#)
- 75 error codes and classes, [443](#), [446](#)
- 76 error handlers, [434](#), [446](#), [809](#)
- 77 fatal after request free, [71](#)
- 78 finalize, [25](#), [470](#), [518](#)
- 79 I/O, [683](#), [684](#)
- 80 initial error handler, [25](#), [433](#), [463](#), [470](#), [502](#)
- 81 multiple completions, [75](#), [77](#)
- 82 one-sided communication, [577](#)
- 83 process failure, [24](#), [518](#)
- 84 program error, [25](#)
- 85 resource error, [25](#)
- 86 startup, [25](#), [463](#), [502](#)
- 87 transmission failure, [24](#)
- 88 establishing communication, [505](#)
- 89 etype, [609](#), [623](#)
- 90 event types, [721](#)
- 91 events, [708](#), [721](#)
- 92 tools interface, [721](#)
- 93 examples, [27](#)
- 94 exclusive lock, [561](#), [570](#), [572](#), [582](#), [583](#)
- 95 exclusive scan
- 96 nonblocking, [259](#)
- 97 persistent, [282](#)
- 98 explicit offsets, [626](#), [628](#)
- 99 exposure epoch, [560](#)
- 100 extent of datatypes, [114](#), [139](#), [140](#)
- 101 true extent, [142](#)
- 102 external32
- 103 file data representation, [663](#)
- 104 extra-state, [813](#)
- 105 fairness

- not guaranteed, [52](#)
- requirement, [78](#)
- file, [609](#)
 - data access, [625](#)
 - collective operations, [649](#)
 - explicit offsets, [628](#)
 - individual file pointers, [635](#)
 - seek, [651](#)
 - shared file pointers, [645](#)
 - split collective, [653](#)
 - end of file, [610](#)
 - filetype, [609](#)
 - handle, [611](#)
 - interoperability, [661](#)
 - manipulation, [611](#)
 - nonblocking collective
 - progress, [678](#)
 - offset, [20](#), [610](#)
 - pointer, [611](#)
 - progress, [677](#)
 - size, [610](#)
 - view, [609](#), [610](#), [622](#)
- file access
 - concurrent, [674](#)
 - conflict, [674](#)
 - overlap, [674](#)
- file size, [679](#)
- finalize, [468](#)
- finished, [471](#)
- Fortran
 - language binding, [22](#), [755](#)
- Fortran support, [755](#)
- freeing procedure, [14](#)
- freeing stage, [11](#), [70](#), [79](#)
- gather, [186](#)
 - nonblocking, [241](#)
 - persistent, [263](#)
- gather-to-all, [202](#)
 - nonblocking, [246](#)
 - persistent, [269](#)
- general datatype, [113](#)
- generalized requests, [599](#), [599](#)
 - progress, [599](#)
- get
 - in function names, [9](#)
- graph
 - topology, [369](#), [372](#)
- group, [34](#), [291](#), [292](#), [294](#), [335](#)
- group objects, [294](#)
- handles, [15](#), [803](#)
- hardware resource type, [319](#), [320](#)
- host rank, [427](#)
- I/O 1
 - blocking, [625](#) 2
 - nonblocking, [625](#) 3
 - immediate, [14](#), [58](#), [60](#), [74](#), [77](#), [78](#), [81](#), [88](#) 4
 - inactive, [67](#), [68–70](#), [73](#), [74](#), [88](#), [95](#), [96](#) 5
 - inclusive scan, [233](#) 6
 - nonblocking, [258](#) 7
 - persistent, [281](#) 8
 - incomplete, [14](#), [60](#), [85](#) 9
 - incomplete procedure, [14](#) 10
 - independent, [517](#) 11
 - individual file pointers, [626](#), [635](#) 12
 - info object, [453](#) 13
 - file info, [618](#) 14
 - keys, [840](#) 15
 - values, [841](#) 16
 - initial error handler, [25](#) 17
 - initialization, [84](#), [90](#), [95](#) 18
 - initialization procedure, [13](#) 19
 - initialization stage, [11](#) 20
 - initiation, [11](#), [58](#), [60](#), [67](#), [71](#), [82](#) 21
 - initiation procedure, [13](#) 22
 - inter-communication, [293](#), [334](#) 23
 - inter-communicator, [180](#), [293](#), [334](#) 24
 - collective operations, [181](#), [183](#) 25
 - point-to-point, [34](#), [36](#) 26
 - interlanguage communication, [814](#) 27
 - internal 28
 - file data representation, [662](#) 29
 - interoperability, [661](#) 30
 - intra-communication, [293](#), [334](#) 31
 - intra-communicator, [180](#), [292](#), [334](#) 32
 - collective operations, [180](#) 33
 - intra-communicator objects, [295](#) 34
 - I/O, [609](#) 35
 - IO rank, [427](#) 36
 - is 37
 - in function names, [9](#) 38
 - language binding, [21](#), [755](#) 39
 - interoperability, [801](#) 40
 - summary, [817](#) 41
 - large 42
 - count, [21](#), [800](#) 43
 - displacement, [800](#) 44
 - large count, [21](#) 45
 - lb_marker, [128](#), [129](#), [132](#), [138](#), [143](#) 46
 - erased, [142](#) 47
 - local, [12](#), [13](#), [47](#), [58](#), [59](#), [68–70](#), [76](#), [78](#), [81](#), [84](#), [88](#), [90](#), [95](#), [476](#) 48
 - local group, [308](#) 49
 - local procedure, [13](#) 50
 - logically concurrent, [51](#), [52](#) 51
 - loosely synchronous model, [364](#) 52

- 1 lower bound, [138](#)
2 lower-bound markers, [138](#)
3
4 macros, [24](#)
5 main thread, [491](#)
6 matched probe
7 progress, [85](#)
8 matched receive, [84](#), [85](#), [85](#)
9 matching
10 type, [146](#), [678](#)
11 matching probe, [82](#), [83](#), [83](#), [84–86](#)
12 matching receive, [84](#)
13 matching rules
14 blocking with nonblocking, [59](#)
15 cancel, [89](#)
16 envelope, [36](#)
17 null process, [96](#)
18 ordering, [51](#)
19 persistent with nonpersistent, [96](#)
20 probe, [81](#), [83](#)
21 send modes, [50](#)
22 type, [42](#), [42](#)
23 wildcard, [36](#)
24 memory
25 alignment, [430](#), [527](#), [529](#), [1004](#)
26 allocation, [429](#), [525](#), [527](#)
27 system, [15](#)
28 memory model, [521](#), [559](#)
29 separate, [521](#), [530](#)
30 unified, [521](#), [530](#)
31 message, [29](#), [83](#)
32 buffer, [48](#)
33 cancel, [69](#), [80](#), [81](#), [88](#), [89](#)
34 data, [29](#), [30](#), [31](#), [46](#), [48](#)
35 envelope, [29](#), [30](#), [33](#), [33](#), [34](#), [36](#), [38](#), [46](#), [57](#)
36 handle, [83](#), [84](#), [86](#), [87](#)
37 intermediate buffering, [42](#), [46](#)
38 invalid handle, [84](#)
39 predefined handle, [84](#)
40 wildcard, [36](#)
41 message handle, [83](#), [84](#), [86](#), [87](#)
42 invalid, [84](#)
43 predefined, [84](#)
44 modes, [46](#)
45 buffered, [47](#), [51](#)
46 ready, [47](#), [51](#)
47 standard, [46](#), [51](#)
48 synchronous, [47](#), [51](#)
49 module variables, [795](#)
50 MPI datatype, [15](#)
51 mpi module
52 Fortran support, [759](#)
53 MPI operation, [11](#)
54 MPI procedure, [12](#)
55 MPI process initialization, [461](#)
56 Dynamic Process Model, [461](#)
57 Sessions Model, [298](#), [427](#), [461](#), [491](#), [612](#)
58 World Model, [295](#), [298](#), [426](#), [461](#), [491](#), [612](#),
59 [684](#)
60 MPI Session handle, [473](#)
61 mpi_f08 module
62 Fortran support, [756](#)
63 MPI_SIZEOF and storage_size(), [22](#), [749](#), [779](#),
64 [780](#)
65 mpiexec, [464](#), [487](#), [488](#)
66 mpif.h include file
67 Fortran support, [761](#)
68 mpirun, [487](#)
69 multiple completions, [72](#), [73–76](#)
70 error handling, [75](#), [77](#)
71
72 named datatype, [15](#)
73 names, [505](#)
74 name publishing, [510](#)
75 naming objects, [360](#)
76 native
77 file data representation, [662](#)
78 neighborhood collective communication, [394](#)
79 nonblocking, [406](#)
80 periodic and dims=1 or 2, [1002](#)
81 nonblocking, [14](#), [58](#), [70](#), [84](#), [88](#), [539](#), [625](#)
82 communication, [58](#)
83 completion, [67](#), [69](#)
84 Fortran problems, [789](#)
85 I/O, [626](#)
86 initiation, [60](#)
87 persistent
88 partitioned completion, [106](#)
89 request objects, [59](#)
90 nonblocking I/O, [625](#)
91 nonblocking operation, [12](#)
92 nonblocking procedure, [14](#)
93 noncollective operation, [12](#)
94 nonlocal, [12](#), [13](#), [47](#), [59](#), [68](#), [82](#), [85](#)
95 nonlocal procedure, [12](#)
96 nonovertaking, [51](#), [71](#)
97 null handle, [67](#), [73–75](#)
98 null handles, [76](#), [80](#)
99 null processes, [96](#)
100
101 offset, [20](#), [610](#)
102 one-sided communication, [521](#)
103 Fortran problems, [790](#)
104 progress, [586](#), [588](#)
105 opaque objects, [15](#), [807](#)
106 operation, [11](#)
107 blocking, [12](#)
108 collective, [12](#)

- complete, [11](#)
- nonblocking, [12](#)
- noncollective, [12](#)
- operation-related, [13](#)
- partitioned receive, [12](#)
- partitioned send, [12](#)
- persistent, [12](#)
- semantics, [11](#)
- stage, [11](#)
 - completion, [11](#)
 - freeing, [11](#)
 - initialization, [11](#)
 - starting, [11](#)
- state
 - sufficiently enabled, [68](#)
- ordered, [51](#), [71](#)
- origin, [522](#)
- overlap, [674](#)
- pack, [166](#)
 - canonical, [173](#)
- packing unit, [169](#)
- parallel procedure, [364](#)
- partitioned completion, [106](#)
- partitioned point-to-point communication, [99](#)
- passive target communication, [560](#)
- performance variables
 - tools interface, [708](#)
- persistent communication request, [67–70](#), [73–77](#), [88](#), [90](#), [90](#), [91–96](#)
 - active, [68](#)
 - completion, [69](#), [90](#)
 - inactive, [67](#)
 - starting, [90](#), [95](#)
- persistent communication requests
 - collective persistent, [260](#), [413](#)
 - Fortran problems, [790](#)
- persistent operation, [12](#)
- PICL, [2](#)
- PMPI_, [689](#)
- point-to-point communication, [29](#)
 - blocking, [30](#)
 - buffer allocation, [55](#)
 - cancel, [88](#)
 - matched receive, [85](#)
 - matching probe, [83](#)
 - nonblocking, [58](#)
 - persistent, [90](#)
 - probe, [80](#)
 - receive operation, [34](#)
 - send modes, [46](#), [51](#)
 - send operation, [30](#)
 - send-receive operation, [39](#)
 - status, [37](#)
- portable datatype, [15](#)
- ports, [505](#)
- positioning, [625](#)
- POSIX
 - environment, [451](#)
 - FORTRAN, [17](#)
 - I/O, [612](#), [613](#), [625](#), [662](#), [674](#)
 - model, [609](#)
- predefined datatype, [15](#)
- predefined reduction operations, [214](#)
- private window copy, [559](#)
- probe, [80](#), [80](#), [81](#), [84](#)
 - matching, [83](#)
 - progress, [82](#)
- procedure, [12](#)
 - blocking, [14](#)
 - collective, [14](#)
 - completing, [13](#)
 - freeing, [14](#)
 - immediate, [14](#)
 - incomplete, [14](#)
 - initialization, [13](#)
 - initiation, [13](#)
 - local, [13](#)
 - nonblocking, [14](#)
 - nonlocal, [12](#)
 - operation-related, [13](#)
 - semantics, [12](#)
 - specification, [10](#)
 - starting, [13](#)
 - synchronizing, [14](#)
- procedure specification, [10](#)
- process creation, [461](#)
- process failures, [24](#)
- process set names, [839](#)
- processes, [24](#)
- processor name, [428](#)
- profiling interface, [689](#)
- program error, [25](#)
- progress, [677](#), [1002](#)
 - file, [677](#)
 - nonblocking collective, [678](#)
 - finalize
 - buffered data, [470](#)
 - generalized requests, [599](#)
 - load/store access synchronization, [575](#), [1001](#)
 - nonblocking collective communication, [237](#), [239](#), [288](#), [290](#)
 - one-sided communication, [575](#), [586](#), [588](#)
 - partitioned point-to-point communication, [107](#)
 - point-to-point communication, [52](#), [72](#), [80](#), [82](#), [85](#), [1002](#)

- 1 threads, [490](#)
- 2 prototype definitions, [832](#)
- 3 deprecated, [838](#)
- 4 public window copy, [559](#)
- 5 PVM, [2](#)
- 6 raise an event, [721](#)
- 7 rank, [294](#)
- 8 ready mode send, [47](#), [51](#), [94](#), [95](#)
- 9 as standard mode send, [51](#)
- 10 nonblocking, [58](#), [60](#), [63](#)
- 11 persistent, [94](#)
- 12 receive, [29](#), [30](#)
- 13 blocking, [34](#), [34](#)
- 14 buffer, [30](#), [86](#), [95](#)
- 15 complete, [58](#)
- 16 context, [335](#)
- 17 matched, [84](#), [85](#), [85](#)
- 18 nonblocking, [58](#), [64](#)
- 19 persistent, [95](#)
- 20 start, [58](#), [64](#)
- 21 reduce, [212](#)
- 22 nonblocking, [253](#)
- 23 persistent, [276](#)
- 24 reduce-scatter, [230](#)
- 25 nonblocking, [256](#), [257](#)
- 26 persistent, [279](#), [280](#)
- 27 reduction operations, [212](#), [810](#)
- 28 predefined, [214](#)
- 29 process-local, [228](#)
- 30 scan, [233](#)
- 31 user-defined, [221](#)
- 32 registration handle, [728](#)
- 33 related, [169](#)
- 34 relative displacement, [19](#), [134](#)
- 35 remote group, [308](#)
- 36 Remote Memory Access
- 37 see RMA, [521](#)
- 38 removed interfaces, [21](#), [751](#)
- 39 representation
- 40 conversion, [45](#)
- 41 request complete
- 42 I/O, [626](#)
- 43 request objects, [59](#)
- 44 completion, [72](#)
- 45 freeing, [70](#), [96](#)
- 46 initiation, [71](#)
- 47 multiple completions, [72](#)
- 48 null handle, [67](#), [73–75](#)
- 49 null handles, [76](#), [80](#)
- 50 started, [47](#), [72](#), [84](#), [90](#), [95](#), [96](#)
- 51 resource error, [25](#)
- 52 RMA, [521](#)
- 53 communication calls, [539](#)
- 54 request-based, [552](#)
- 55 dynamic window, [532](#)
- 56 memory model, [559](#)
- 57 synchronization calls, [560](#)
- 58 root, [178](#)
- 59 scan, [233](#)
- 60 inclusive, [233](#)
- 61 segmented, [236](#)
- 62 scatter, [196](#)
- 63 nonblocking, [244](#)
- 64 persistent, [266](#)
- 65 seek, [651](#)
- 66 segmented scan, [236](#)
- 67 semantic changes, [753](#)
- 68 semantics, [11](#), [1002](#)
- 69 collective communications, [178](#)
- 70 nonblocking progress, [239](#), [288](#), [290](#)
- 71 data conversion, [45](#)
- 72 deadlock
- 73 buffer space, [53](#)
- 74 cyclic shift, [40](#)
- 75 send to self, [36](#)
- 76 erroneous program
- 77 freeing active request, [70](#)
- 78 invalid matched receive, [87](#), [88](#)
- 79 lack of buffer space, [53](#)
- 80 ready mode before receive, [47](#)
- 81 type matching, [43](#)
- 82 exceptions
- 83 completing and local, [49](#)
- 84 incomplete and nonlocal, [85](#)
- 85 file
- 86 collective, [677](#)
- 87 collective access, [649](#)
- 88 conflicting access, [674](#)
- 89 consistency, [673](#)
- 90 explicit offsets, [628](#)
- 91 nonblocking collective, [677](#)
- 92 progress, [677](#)
- 93 shared file pointer, [645](#)
- 94 split collective, [674](#)
- 95 finalize
- 96 buffered data, [470](#)
- 97 generalized requests
- 98 progress, [599](#)
- 99 inter-communicator, [308](#)
- 100 MPI_COMM_IDUP, [310](#)
- 101 nonblocking communications, [71](#)
- 102 nonblocking completion, [67](#)
- 103 nonblocking partitioned communications, [107](#)
- 104 operation, [11](#)
- 105 overlap, [58](#)

- partitioned point-to-point communication, [100](#)
- progress, [107](#)
- point-to-point communication, [51](#)
 - cancel, [69](#), [80](#), [80](#), [81](#), [88](#), [89](#)
 - deterministic, [51](#)
 - fairness not guaranteed, [52](#)
 - fairness requirement, [78](#)
 - logically concurrent, [51](#)
 - matched receive, [85](#)
 - matching probe, [83](#)
 - nonovertaking, [51](#), [71](#)
 - ordered, [51](#), [71](#)
 - overflow error on truncation, [35](#)
 - persistent, [90](#)
 - probe, [80](#)
 - progress, [52](#), [72](#), [80](#), [82](#), [85](#), [1002](#)
 - resource limitations, [53](#)
 - send modes, [46](#), [51](#)
 - send-receive concurrency, [41](#)
 - wildcard receive, [36](#)
- procedure, [12](#)
- process failure, [518](#)
- terms, [11](#)
- threads
 - progress, [490](#)
- semantics and correctness
 - one-sided communication, [577](#)
 - progress, [586](#), [588](#)
- send, [29](#)
 - blocking, [30](#), [30](#)
 - buffer, [29](#)
 - complete, [58](#)
 - context, [335](#)
 - nonblocking, [58](#)
 - persistent, [91](#)
 - start, [58](#), [61](#)–[63](#)
- send-receive
 - blocking, [39](#), [39](#)
 - nonblocking, [66](#), [67](#)
 - start, [66](#), [67](#)
- separate memory model, [521](#), [530](#), [559](#)
- sequential storage, [150](#)
- serialization, [58](#)
- Sessions Model, [295](#), [298](#), [427](#), [461](#), [491](#), [612](#)
- set
 - in function names, [9](#)
- shared file pointers, [626](#), [645](#)
- shared lock, [561](#), [570](#), [582](#)
- shared memory allocation, [527](#)
- signals, [26](#)
- singleton init, [516](#)
- size changing
 - I/O, [679](#)
- source, [335](#)
- split collective, [625](#), [653](#)
- stage, [11](#)
 - completion, [11](#)
 - freeing, [11](#)
 - initialization, [11](#)
 - starting, [11](#)
- standard mode send, [46](#), [47](#), [51](#), [91](#)
 - as synchronous mode send, [51](#)
 - nonblocking, [59](#), [61](#)
- started, [471](#)
 - request objects, [47](#), [72](#), [84](#), [90](#), [95](#), [96](#)
- starting procedure, [13](#), [90](#)
- starting processes, [493](#), [494](#)
- starting stage, [11](#)
- startup, [462](#)
 - portable, [487](#)
- state, [18](#)
 - operation
 - sufficiently enabled, [68](#)
- status, [36](#), [805](#)
 - array in Fortran, [37](#)
 - associating information, [605](#)
 - derived type in Fortran 2008, [37](#)
 - empty, [68](#), [69](#), [73](#)–[76](#), [80](#)
 - error in status, [37](#)
 - for send operation, [68](#)
 - ignore, [39](#)
 - message length, [37](#)
 - structure in C, [37](#)
 - test, [79](#)
- strong synchronization, [562](#)
- sufficiently enabled operation state, [68](#)
- synchronism, [625](#)
- synchronization, [521](#), [539](#)
- synchronization calls
 - RMA, [560](#)
- synchronizing procedure, [14](#)
- synchronous mode send, [47](#), [51](#), [67](#), [84](#), [93](#)
 - nonblocking, [59](#), [60](#), [62](#)
 - persistent, [93](#)
- system memory, [15](#)
- tag values, [427](#)
- target, [522](#)
- target nodes, [621](#)
- thread compliant, [466](#), [489](#)
- thread-safe, [490](#)
- threads, [489](#)
 - progress, [490](#)
 - thread-safe, [2](#), [426](#), [442](#), [446](#), [490](#), [724](#)
- timers and synchronization, [450](#)
- tool information interface, [695](#)
- tool support, [689](#)

1 topologies, [367](#)
2 topology
3 Cartesian, [369](#), [370](#)
4 distributed graph, [369](#), [374](#)
5 graph, [369](#), [372](#)
6 virtual, [367](#), [368](#)
7 transmission failures, [24](#)
8 true extent of datatypes, [142](#)
9 TS 29113, [19](#), [23](#), [755–760](#), [762](#), [764](#), [765](#),
10 [768–772](#), [781](#), [782](#), [787](#), [792](#), [795](#), [798](#),
11 [1012](#)
12 type
13 conversion, [45](#)
14 matching rules, [42](#)
15 type map, [114](#)
16 type matching, [146](#)
17 type signature, [114](#)
18 types, [831](#)
19 ub_marker, [128](#), [129](#), [132](#), [133](#), [138](#), [143](#)
20 erased, [142](#)
21 unified memory model, [521](#), [529](#), [559](#)
22 universe size, [515](#)
23 unnamed datatype, [15](#)
24 unpack, [166](#)
25 canonical, [173](#)
26 upper bound, [138](#)
27 upper-bound markers, [138](#)
28 user functions at process termination, [472](#)
29 user-defined data representations, [666](#)
30 user-defined reduction operations, [221](#)
31 verbosity levels
32 tools interface, [696](#)
33 version inquiries, [425](#)
34 view, [609](#), [610](#), [622](#)
35 virtual topology, [292](#), [367](#), [368](#)
36 weak synchronization, [562](#)
37 wildcard, [36](#)
38 window
39 allocation, [525](#)
40 creation, [522](#)
41 dynamically attached memory, [531](#)
42 shared memory allocation, [527](#)
43 World Model, [295](#), [298](#), [426](#), [461](#), [491](#), [612](#), [684](#)
44 XDR, [46](#)
45 Zipcode, [2](#)
46
47
48

Examples Index

This index lists code examples throughout the text. Some examples are referred to by content; others are listed by the major MPI function that they are demonstrating. MPI functions listed in all capital letter are Fortran examples; MPI functions listed in mixed case are C examples.

- Absolute addresses for struct
 - MPI_GET_ADDRESS, 808
- Accumulate in RMA
 - MPI_ACCUMULATE, 547
 - MPI_TYPE_GET_EXTENT, 547
 - MPI_WIN_CREATE, 547
 - MPI_WIN_FENCE, 547
 - MPI_WIN_FREE, 547
- Actions after Finalize
 - MPI_Finalize, 471
- Active target and local reads in RMA
 - MPI_Barrier, 585
 - MPI_Put, 585
 - MPI_Win_complete, 585
 - MPI_Win_lock, 585
 - MPI_Win_post, 585
 - MPI_Win_start, 585
 - MPI_Win_unlock, 585
 - MPI_Win_wait, 585
- Allgather
 - MPI_Allgather, 205
- Allreduce of a vector
 - MPI_ALLREDUCE, 228
- argv in C and Fortran
 - MPI_COMM_SPAWN, 497
 - MPI_Comm_spawn, 497
- Array of argv in C and Fortran
 - MPI_COMM_SPAWN_MULTIPLE, 502
 - MPI_Comm_spawn_multiple, 502
- ASYNCHRONOUS, 593, 791, 798
- Attach and detach buffer
 - MPI_Buffer_attach, 56
 - MPI_Buffer_detach, 56
- Attributes between languages, 811
 - MPI_Comm_get_attr, 811
 - MPI_Comm_set_attr, 811
- Basic usage of performance variables
 - MPI_T_finalize, 718
 - MPI_T_init_thread, 718
 - MPI_T_pvar_get_info, 718
 - MPI_T_pvar_handle_alloc, 718
 - MPI_T_pvar_handle_free, 718
 - MPI_T_pvar_read, 718
 - MPI_T_pvar_session_create, 718
 - MPI_T_pvar_start, 718
- Blocking/Nonblocking collectives do not match, 288
- Broadcast
 - MPI_Bcast, 185
- C/Fortran handle conversion, 804
- Cartesian virtual topologies, 421
- Client server code with waitsome
 - MPI_IRECV, 78
 - MPI_ISEND, 78
 - MPI_WAIT, 78
 - MPI_WAIT SOME, 78
- Client-server
 - MPI_Comm_accept, 513
 - MPI_Comm_connect, 513
 - MPI_IRECV, 78
 - MPI_ISEND, 78
 - MPI_Open_port, 513
 - MPI_WAIT, 78
 - MPI_WAITANY, 78
- Client-server (with error)
 - MPI_PROBE, 82
 - MPI_RECV, 82
 - MPI_SEND, 82
- Client-server code, 78
 - with probe, 82
 - with probe (wrong), 82
- Client-server model
 - MPI_Comm_remote_size, 316
 - MPI_Comm_split, 316
- Client-server with probe
 - MPI_PROBE, 82
 - MPI_RECV, 82
 - MPI_SEND, 82
- Collective communication
 - MPI_Bcast, 328
- Communication safety
 - MPI_Comm_create, 329
 - MPI_Group_free, 329
 - MPI_Group_incl, 329
- Counting semaphore (non-scalable)
 - MPI_Accumulate, 592

- 1 MPI_Barrier, 592
- 2 MPI_Get_accumulate, 592
- 3 MPI_Win_flush, 592
- 4 MPI_Win_sync, 592
- 5 Creating a communicator using the Sessions
- 6 Model
- 7 MPI_Comm_create_from_group, 480
- 8 MPI_Group_from_session_pset, 480
- 9 MPI_Info_create, 480
- 10 MPI_Info_set, 480
- 11 MPI_Session_finalize, 480
- 12 MPI_Session_init, 480
- 13 Critical region with Compare-and-Swap
- 14 MPI_Barrier, 593
- 15 MPI_Compare_and_swap, 593
- 16 MPI_Win_flush, 593
- 17 MPI_Win_sync, 593
- 18 Critical region with RMA
- 19 MPI_Accumulate, 592
- 20 MPI_Barrier, 592
- 21 MPI_Get_accumulate, 592
- 22 MPI_Win_flush, 592
- 23 MPI_Win_flush_all, 592
- 24 MPI_Win_sync, 592
- 25 Datatype
- 26 3D array, 158
- 27 absolute addresses, 163
- 28 array of structures, 161
- 29 elaborate example, 171, 172
- 30 matching type, 147
- 31 matrix transpose, 160
- 32 union, 164
- 33 Datatype matching
- 34 MPI_RECV, 43
- 35 MPI_SEND, 43
- 36 Datatypes
- 37 matching, 43
- 38 MPI_BYTE, 44
- 39 MPI_RECV, 44
- 40 MPI_SEND, 44
- 41 not matching, 43
- 42 untyped, 44
- 43 Datatypes for distributed arrays
- 44 MPI_TYPE_CREATE_DARRAY, 134
- 45 Deadlock
- 46 if not buffered, 54
- 47 with MPI_Bcast, 284
- 48 wrong message exchange, 53
- Decoding a datatype
- MPI_Type_get_contents, 165
- MPI_Type_get_envelope, 165
- Decoding amode in Fortran
- MPI_FILE_GET_AMODE, 618
- Defining a user function
- MPI_OP_CREATE, 226
- MPI_REDUCE, 226
- MPI_USER_FUNCTION, 226
- Dims create
- MPI_DIMS_CREATE, 372
- dist graph creation
- MPI_DIST_GRAPH_CREATE, 379
- MPI_DIST_GRAPH_CREATE_ADJACENT, 379
- Dist_graph_create
- MPI_Dist_graph_create, 380
- Double buffer in RMA
- MPI_Barrier, 591
- MPI_Get, 591
- MPI_Win_complete, 591
- MPI_Win_post, 591
- MPI_Win_start, 591
- MPI_Win_wait, 591
- Errant message exchange
- MPI_RECV, 53
- MPI_SEND, 53
- Erroneous matching of blocking and nonblocking collectives
- MPI_Alltoall, 288
- MPI_lalltoall, 288
- MPI_Wait, 288
- Erroneous matching of collectives
- MPI_Bcast, 287
- MPI_lbarrier, 287
- MPI_Wait, 287
- Erroneous use of Bcast
- MPI_Bcast, 284
- Example 3.1: Hello world, 29
- Example 3.2: Datatype matching, 43
- Example 3.3: Datatype matching, 43
- Example 3.4: Datatypes, 44
- Example 3.5: Fortran CHARACTER, 44
- Example 3.6: Nonovertaking messages, 52
- Example 3.7: Message matching, 52
- Example 3.8: Message exchange, 53
- Example 3.9: Errant message exchange, 53
- Example 3.10: Exchange relies on buffering, 54
- Example 3.11: Attach and detach buffer, 56
- Example 3.12: Nonblocking point-to-point, 70
- Example 3.13: Nonblocking send and receive with request free, 71
- Example 3.14: Message ordering for nonblocking operations, 71
- Example 3.15: Progress semantics, 72
- Example 3.16: Client-server, 78
- Example 3.17: Client server code with waitsome, 78

- Example 3.18: Client-server with probe, [82](#)
- Example 3.19: Client-server (with error), [82](#)
- Example 4.1: Partitioned communication, [100](#)
- Example 4.2: Partitioned communication using threads, [108](#)
- Example 4.3: Partitioned communication with tasks, [109](#)
- Example 4.4: Partitioned communication with partial completion, [111](#)
- Example 5.1: Typemap, [115](#)
- Example 5.2: Typemap for contiguous, [116](#)
- Example 5.3: Typemap for vector, [117](#)
- Example 5.4: Typemap for vector, [117](#)
- Example 5.5: Typemap for indexed, [120](#)
- Example 5.6: Typemap for create struct, [126](#)
- Example 5.7: Datatypes for distributed arrays, [134](#)
- Example 5.8: Get_address, [135](#)
- Example 5.9: Typemap of nested datatypes, [138](#)
- Example 5.10: Type_commit, [144](#)
- Example 5.11: Matching type with datatypes, [147](#)
- Example 5.12: Using Get_count and Get_elements, [149](#)
- Example 5.13: Send/receive of a 3D array, [158](#)
- Example 5.14: Using indexed datatype, [159](#)
- Example 5.15: Nested vector datatypes, [160](#)
- Example 5.16: Transpose with datatypes, [160](#)
- Example 5.17: Using datatypes with array of structures, [161](#)
- Example 5.18: Using datatypes with array of structures with absolute addresses, [163](#)
- Example 5.19: Using datatypes with unions, [164](#)
- Example 5.20: Decoding a datatype, [165](#)
- Example 5.21: Using Pack, [171](#)
- Example 5.22: Pack/Unpack with struct datatype, [171](#)
- Example 5.23: Pack and Pack_size, [172](#)
- Example 6.1: Broadcast, [185](#)
- Example 6.2: Gather, [190](#)
- Example 6.3: Gather with allocation at root, [190](#)
- Example 6.4: Gather with datatype, [190](#)
- Example 6.5: Gather, [191](#)
- Example 6.6: Gather with datatype, [191](#)
- Example 6.7: Gather with datatype, [192](#)
- Example 6.8: Gather with struct datatype, [193](#)
- Example 6.9: Gather with vector datatype, [194](#)
- Example 6.10: Gather and Gather, [194](#)
- Example 6.11: Scatter, [200](#)
- Example 6.12: Scatter, [200](#)
- Example 6.13: Scatter with vector datatype, [201](#)
- Example 6.14: Allgather, [205](#)
- Example 6.15: Reduction, [216](#)
- Example 6.16: Reduction of a vector, [217](#)
- Example 6.17: Reduction with maxloc, [219](#)
- Example 6.18: Reduction with maxloc, [220](#)
- Example 6.19: Reduction with minloc, [220](#)
- Example 6.20: Reduction with user-defined op, [225](#)
- Example 6.21: Defining a user function, [226](#)
- Example 6.22: Allreduce of a vector, [228](#)
- Example 6.23: User-defined operation with Scan, [236](#)
- Example 6.24: Ibcast, [241](#)
- Example 6.25: Erroneous use of Bcast, [284](#)
- Example 6.26: Erroneous use of Bcast, [284](#)
- Example 6.27: Erroneous use of Bcast, [284](#)
- Example 6.28: Nondeterministic use of Bcast, [285](#)
- Example 6.29: Mixing blocking and nonblocking collective operations, [286](#)
- Example 6.30: Erroneous matching of collectives, [287](#)
- Example 6.31: Progress of nonblocking collectives, [288](#)
- Example 6.32: Erroneous matching of blocking and nonblocking collectives, [288](#)
- Example 6.33: Mixing collective and point-to-point, [288](#)
- Example 6.34: Pipelining nonblocking collectives, [289](#)
- Example 6.35: Overlapping communicators and collectives, [289](#)
- Example 6.36: Independence of nonblocking operations, [290](#)
- Example 7.1: Inter-communicator creation, [312](#)
- Example 7.2: Client-server model, [316](#)
- Example 7.3: Splitting into NUMANode subcommunicators, [320](#)
- Example 7.4: Recursive splitting of COMM_WORLD, [321](#)
- Example 7.5: Parallel output of a message, [327](#)
- Example 7.6: Message exchange, [327](#)
- Example 7.7: Collective communication, [328](#)
- Example 7.8: Using Group_excl, [328](#)
- Example 7.9: Communication safety, [329](#)
- Example 7.10: Library Example #1, [330](#)
- Example 7.11: Library Example #2, [331](#)
- Example 7.12: Three-Group “Pipeline”, [341](#)
- Example 7.13: Three-Group “Ring”, [342](#)
- Example 8.1: Dims create, [372](#)

- 1 Example 8.2: Graph create, [373](#)
2 Example 8.3: dist_graph creation, [379](#)
3 Example 8.4: Dist_graph_create, [380](#)
4 Example 8.5: Graph creation and neighbors
5 count, [386](#)
6 Example 8.6: Graph creation, [387](#)
7 Example 8.7: Using Cart_shift, [390](#)
8 Example 8.8: Subgroup cart process topology,
9 [391](#)
10 Example 8.9: Neighbor allgather, [397](#)
11 Example 8.10: Neighbor alltoall, [401](#)
12 Example 8.11: Neighborhood collective
13 communication, [421](#)
14 Example 9.1: Use of Alloc_mem in Fortran,
15 [431](#)
16 Example 9.2: Use of Alloc_mem in Fortran
17 with Cray pointers, [432](#)
18 Example 9.3: Using Alloc_mem, [432](#)
19 Example 9.4: Using MPI_Wtime, [451](#)
20 Example 11.1: Initializing MPI, [463](#)
21 Example 11.2: mpiexec and environment
22 variables, [464](#)
23 Example 11.3: Rules for finalize, [469](#)
24 Example 11.4: Rules for finalize, [469](#)
25 Example 11.5: Finalize and request free, [469](#)
26 Example 11.6: Finalize and buffer attach, [469](#)
27 Example 11.7: Finalize and cancel, [470](#)
28 Example 11.8: Actions after Finalize, [471](#)
29 Example 11.9: Finalize in the Sessions Model,
30 [476](#)
31 Example 11.10: Creating a communicator
32 using the Sessions Model, [480](#)
33 Example 11.11: Using process set query in
34 group creation, [482](#)
35 Example 11.12: Using process set query in
36 group creation, [484](#)
37 Example 11.13: Using mpiexec
38 using -n, [489](#)
39 Example 11.14: Using mpiexec
40 using -host, [489](#)
41 Example 11.15: Using mpiexec
42 starting programs with separate
43 argument lists, [489](#)
44 Example 11.16: Using mpiexec
45 using -arch, [489](#)
46 Example 11.17: Using mpiexec
47 using -configfile, [489](#)
48 Example 11.18: Threads and MPI, [490](#)
Example 11.19: argv in C and Fortran, [497](#)
Example 11.20: Array of argv in C and
Fortran, [502](#)
Example 11.21: Manager-worker with
Comm_spawn, [503](#)
Example 11.22: Client-server, [513](#)
Example 11.23: Name publishing, [513](#)
Example 11.24: Client-server, [513](#)
Example 12.1: Using Get with indexed
datatype, [543](#)
Example 12.2: Using Get, [545](#)
Example 12.3: Accumulate in RMA, [547](#)
Example 12.4: RMA Start and complete, [566](#)
Example 12.5: RMA Lock and unlock, [572](#)
Example 12.6: Update location in separate
memory model, [582](#)
Example 12.7: Update location in unified
memory model, [582](#)
Example 12.8: Read data in RMA, [583](#)
Example 12.9: Read data in RMA (unsafe), [583](#)
Example 12.10: Public and private memory in
RMA, [584](#)
Example 12.11: Public and private memory in
RMA, [584](#)
Example 12.12: Active target and local reads
in RMA, [585](#)
Example 12.13: Register and Compiler
Optimization, [589](#)
Example 12.14: Put with Fence, [589](#)
Example 12.15: Get with fence, [589](#)
Example 12.16: Put with PSCW, [590](#)
Example 12.17: Get with split phases, [590](#)
Example 12.18: Double buffer in RMA, [591](#)
Example 12.19: Counting semaphore
(nonscalable), [592](#)
Example 12.20: Critical region with RMA, [592](#)
Example 12.21: Critical region with
Compare-and-Swap, [593](#)
Example 12.22: Shared memory windows, [593](#)
Example 12.23: Requests in RMA, [594](#)
Example 12.24: Linked list in RMA, [595](#)
Example 13.1: User-defined reduce, [603](#)
Example 14.1: Decoding amode in Fortran, [618](#)
Example 14.2: Read to end of file, [637](#)
Example 14.3: File pointer update semantics,
[640](#)
Example 15.1: Measurement wrapper, [692](#)
Example 15.2: Profiling interface
implementation using weak symbols, [692](#)
Example 15.3: Profiling interface
implementation using the C macro
preprocessor, [692](#)
Example 15.5: Listing names of control
variables, [705](#)
Example 15.6: Reading a control variable, [708](#)
Example 15.7: Basic usage of performance
variables, [718](#)
Example 19.1: Fortran 90 register
optimization, [789](#)

Example 19.2: Fortran 90 register optimization, 789	MPI_SESSION_FINALIZE, 476	1
Example 19.3: Fortran 90 register optimization, 790	Fortran 90	2
Example 19.4: Fortran 90 register optimization, 790	copying and sequence problem, 782 , 784 , 785	3
Example 19.5: Protecting nonblocking communication with ASYNCHRONOUS, 791	derived types	4
Example 19.6: Fortran 90 overlapping communication and computation, 796	MPI_GET_ADDRESS, 786	5
Example 19.7: Fortran 90 overlapping communication and computation, 796	MPI_TYPE_COMMIT, 786	6
Example 19.8: Fortran 90 overlapping communication and computation, 796	MPI_TYPE_CREATE_RESIZED, 786	7
Example 19.9: Using separated variables, 798	MPI_TYPE_CREATE_STRUCT, 786	8
Example 19.10: Fortran 90 overlapping communication and computation, 799	heterogeneous communication (unsafe), 779 , 780	9
Example 19.11: C/Fortran handle conversion, 804	invalid KIND, 776	10
Example 19.12: Absolute addresses for struct, 808	MPI_TYPE_MATCH_SIZE	11
Example 19.13: Attributes between languages, 811	implementation, 779	12
Example 19.14: Setting an attribute in Fortran and reading in C or Fortran, 812	overlapping communication and computation, 796 , 799	13
Example 19.15: Setting an attribute in Fortran and reading in C or Fortran, 812	register optimization, 789 , 790	14
Example 19.16: Interlanguage communication, 814	Fortran CHARACTER	15
Exchange relies on buffering	MPI_CHARACTER, 44	16
MPI_RECV, 54	MPI_RECV, 44	17
MPI_SEND, 54	MPI_SEND, 44	18
False matching of collective operations, 287	Gather	19
File pointer update semantics	MPI_Gather, 190	20
MPI_FILE_CLOSE, 640	Gather and Gatherv	21
MPI_FILE_IREAD, 640	MPI_Gather, 194	22
MPI_FILE_OPEN, 640	MPI_Gatherv, 194	23
MPI_FILE_SET_VIEW, 640	MPI_Type_commit, 194	24
MPI_WAIT, 640	MPI_Type_create_struct, 194	25
Finalize and buffer attach	Gather with allocation at root	26
MPI_Buffer_attach, 469	MPI_Gather, 190	27
MPI_Finalize, 469	Gather with datatype	28
Finalize and cancel	MPI_Gather, 190	29
MPI_Barrier, 470	MPI_Type_commit, 190	30
MPI_Cancel, 470	MPI_Type_contiguous, 190	31
MPI_Finalize, 470	Gatherv	32
MPI_Iprobe, 470	MPI_Gatherv, 191	33
MPI_Test_cancelled, 470	Gatherv with datatype	34
Finalize and request free	MPI_Gatherv, 191 , 192	35
MPI_Finalize, 469	MPI_Type_commit, 191 , 192	36
MPI_Request_free, 469	MPI_Type_vector, 191 , 192	37
Finalize in the Sessions Model	Gatherv with struct datatype	38
	MPI_Gatherv, 193	39
	MPI_Type_commit, 193	40
	MPI_Type_create_struct, 193	41
	Gatherv with vector datatype	42
	MPI_Gatherv, 194	43
	MPI_Type_commit, 194	44
	MPI_Type_vector, 194	45
	Get with fence	46
	MPI_Get, 589	47
	MPI_Win_fence, 589	48
	Get with split phases	
	MPI_Get, 590	

- 1 MPI_Win_complete, 590
- 2 MPI_Win_post, 590
- 3 MPI_Win_start, 590
- 4 MPI_Win_wait, 590
- 5 Get_address
 - 6 MPI_GET_ADDRESS, 135
- 6 Graph create
 - 7 MPI_GRAPH_CREATE, 373
- 8 Graph creation
 - 9 MPI_GRAPH_CREATE, 387
- 10 Graph creation and neighbors count
 - 11 MPI_GRAPH_CREATE, 386
 - 12 MPI_GRAPH_NEIGHBORS, 386
 - 13 MPI_GRAPH_NEIGHBORS_COUNT, 386
- 14 Hello world
 - 15 MPI_Comm_rank, 29
 - 16 MPI_Init, 29
 - 17 MPI_Recv, 29
 - 18 MPI_Send, 29
- 19 Ibcast
 - 20 MPI_Ibcast, 241
- 21 Independence of nonblocking operations, 290
 - 22 MPI_Ibcast, 290
- 23 Initializing MPI
 - 24 MPI_Init, 463
- 25 Inter-communicator, 312, 316
- 26 Inter-communicator creation
 - 27 MPI_Comm_create, 312
 - 28 MPI_Comm_group, 312
 - 29 MPI_Group_free, 312
 - 30 MPI_Group_incl, 312
- 31 Interlanguage communication, 814
 - 32 MPI_GET_ADDRESS, 814
 - 33 MPI_TYPE_CREATE_STRUCT, 814
- 34 Intertwined matching pairs, 52
- 35 Library Example #1
 - 36 MPI_Comm_dup, 330
 - 37 MPI_Reduce, 330
- 38 Library Example #2
 - 39 MPI_Comm_create, 331
 - 40 MPI_Comm_group, 331
 - 41 MPI_Group_free, 331
 - 42 MPI_Group_incl, 331
- 43 Linked list in RMA
 - 44 MPI_Accumulate, 595
 - 45 MPI_Aint_add, 595
 - 46 MPI_Alloc_mem, 595
 - 47 MPI_Compare_and_swap, 595
 - 48 MPI_Free_mem, 595
 - MPI_Get_accumulate, 595
 - MPI_Win_attach, 595
 - MPI_Win_create_dynamic, 595
 - MPI_Win_detach, 595
 - MPI_Win_flush, 595
 - MPI_Win_lock_all, 595
 - MPI_Win_unlock_all, 595
- Linking libraries when using the profiling interface, 693
- Listing names of control variables
 - MPI_T_cvar_get_info, 705
- Manager-worker with Comm_spawn
 - MPI_Comm_get_parent, 503
 - MPI_Comm_spawn, 503
- Matching type with datatypes
 - MPI_RECV, 147
 - MPI_SEND, 147
 - MPI_TYPE_CONTIGUOUS, 147
- Measurement wrapper
 - MPI_Send, 692
 - MPI_Type_size, 692
 - MPI_Wtime, 692
- Message exchange
 - MPI_RECV, 53
 - MPI_SEND, 53
- Message exchange (ping-pong), 53
- Message matching
 - MPI_BSEND, 52
 - MPI_RECV, 52
 - MPI_SSEND, 52
- Message ordering for nonblocking operations
 - MPI_IRECV, 71
 - MPI_ISEND, 71
 - MPI_WAIT, 71
- Mixing blocking and nonblocking collective operations, 286
 - MPI_Bcast, 286
 - MPI_lbarrier, 286
 - MPI_Wait, 286
- Mixing collective and point-to-point
 - MPI_lbarrier, 288
 - MPI_lrecv, 288
 - MPI_Send, 288
 - MPI_Wait, 288
 - MPI_Waitall, 288
- Mixing collective and point-to-point requests, 288
- MPI_ACCUMULATE
 - Accumulate in RMA, 547
- MPI_Accumulate
 - Counting semaphore (non-scalable), 592
 - Critical region with RMA, 592
 - Linked list in RMA, 595
- MPI_Aint
 - Using datatypes with array of structures, 161

MPI_Aint_add			
Linked list in RMA, 595			
MPI_Allgather			
Allgather, 205			
MPI_ALLOC_MEM			
Use of Alloc_mem in Fortran, 431			
Use of Alloc_mem in Fortran with Cray pointers, 432			
MPI_Alloc_mem			
Linked list in RMA, 595			
Using Alloc_mem, 432			
MPI_ALLREDUCE			
Allreduce of a vector, 228			
MPI_Alltoall			
Erroneous matching of blocking and nonblocking collectives, 288			
MPI_ASYNC_PROTECTS_NONBLOCKING,			
593			
MPI_ATTR_GET			
Setting an attribute in Fortran and reading in C or Fortran, 812			
MPI_BARRIER			
Using process set query in group creation, 484			
MPI_Barrier			
Active target and local reads in RMA, 585			
Counting semaphore (non-scalable), 592			
Critical region with Compare-and-Swap, 593			
Critical region with RMA, 592			
Double buffer in RMA, 591			
Finalize and cancel, 470			
Public and private memory in RMA, 584			
Read data in RMA, 583			
Read data in RMA (unsafe), 583			
Update location in separate memory model, 582			
Update location in unified memory model, 582			
MPI_Bcast			
Broadcast, 185			
Collective communication, 328			
Erroneous matching of collectives, 287			
Erroneous use of Bcast, 284			
Mixing blocking and nonblocking collective operations, 286			
Nondeterministic use of Bcast, 285			
MPI_BSEND			
Message matching, 52			
Nonovertaking messages, 52			
MPI_Buffer_attach			
Attach and detach buffer, 56			
Finalize and buffer attach, 469			
MPI_Buffer_detach			
Attach and detach buffer, 56			1
MPI_BYTE			2
Datatypes, 44			3
MPI_Cancel			4
Finalize and cancel, 470			5
MPI_CART_COORDS			6
Using Cart_shift, 390			7
MPI_CART_GET			8
Neighborhood collective communication, 421			9
MPI_CART_RANK			10
Using Cart_shift, 390			11
MPI_CART_SHIFT			12
Neighbor alltoall, 401			13
Neighborhood collective communication, 421			14
Using Cart_shift, 390			15
MPI_CART_SUB			16
Subgroup cart process topology, 391			17
MPI_CARTDIM_GET			18
Neighbor alltoall, 401			19
MPI_CHARACTER			20
Fortran CHARACTER, 44			21
MPI_Comm_accept			22
Client-server, 513			23
Name publishing, 513			24
MPI_Comm_connect			25
Client-server, 513			26
Name publishing, 513			27
MPI_Comm_create			28
Communication safety, 329			29
Inter-communicator creation, 312			30
Library Example #2, 331			31
Using Group_excl, 328			32
MPI_COMM_CREATE_FROM_GROUP			33
Using process set query in group creation, 484			34
MPI_Comm_create_from_group			35
Creating a communicator using the Sessions Model, 480			36
MPI_Comm_dup			37
Library Example #1, 330			38
MPI_COMM_GET_ATTR			39
Setting an attribute in Fortran and reading in C or Fortran, 812			40
MPI_Comm_get_attr			41
Attributes between languages, 811			42
MPI_Comm_get_parent			43
Manager-worker with Comm_spawn, 503			44
MPI_Comm_group			45
Inter-communicator creation, 312			46
Library Example #2, 331			47
MPI_Comm_rank			48
Hello world, 29			

- 1 MPI_Comm_remote_size
- 2 Client-server model, 316
- 3 MPI_COMM_SET_ATTR
- 4 Setting an attribute in Fortran and
- 5 reading in C or Fortran, 812
- 6 MPI_Comm_set_attr
- 7 Attributes between languages, 811
- 8 MPI_COMM_SPAWN
- 9 argv in C and Fortran, 497
- 10 MPI_Comm_spawn
- 11 argv in C and Fortran, 497
- 12 Manager-worker with Comm_spawn, 503
- 13 MPI_COMM_SPAWN_MULTIPLE
- 14 Array of argv in C and Fortran, 502
- 15 MPI_Comm_spawn_multiple
- 16 Array of argv in C and Fortran, 502
- 17 MPI_Comm_split
- 18 Client-server model, 316
- 19 Three-Group “Pipeline”, 341
- 20 Three-Group “Ring”, 342
- 21 MPI_Comm_split_type
- 22 Recursive splitting of COMM_WORLD,
- 23 321
- 24 Splitting into NUMANode
- 25 subcommunicators, 320
- 26 MPI_Compare_and_swap
- 27 Critical region with Compare-and-Swap,
- 28 593
- 29 Linked list in RMA, 595
- 30 MPI_DIMS_CREATE
- 31 Dims create, 372
- 32 Neighborhood collective communication,
- 33 421
- 34 MPI_DIST_GRAPH_CREATE
- 35 dist graph creation, 379
- 36 MPI_Dist_graph_create
- 37 Dist_graph_create, 380
- 38 MPI_DIST_GRAPH_CREATE_ADJACENT
- 39 dist graph creation, 379
- 40 MPI_F_SYNC_REG
- 41 Shared memory windows, 593
- 42 MPI_FILE_CLOSE
- 43 File pointer update semantics, 640
- 44 Read to end of file, 637
- 45 MPI_FILE_GET_AMODE
- 46 Decoding amode in Fortran, 618
- 47 MPI_FILE_IREAD
- 48 File pointer update semantics, 640
- 49 MPI_FILE_OPEN
- 50 File pointer update semantics, 640
- 51 Read to end of file, 637
- 52 MPI_FILE_READ
- 53 Read to end of file, 637
- 54 MPI_FILE_SET_VIEW
- 55 File pointer update semantics, 640
- 56 Read to end of file, 637
- 57 MPI_Finalize
- 58 Actions after Finalize, 471
- 59 Finalize and buffer attach, 469
- 60 Finalize and cancel, 470
- 61 Finalize and request free, 469
- 62 Rules for finalize, 469
- 63 MPI_FREE_MEM
- 64 Use of Alloc_mem in Fortran, 431
- 65 Use of Alloc_mem in Fortran with Cray
- 66 pointers, 432
- 67 MPI_Free_mem
- 68 Linked list in RMA, 595
- 69 MPI_Gather
- 70 Gather, 190
- 71 Gather and Gatherv, 194
- 72 Gather with allocation at root, 190
- 73 Gather with datatype, 190
- 74 Pack and Pack_size, 172
- 75 MPI_Gatherv
- 76 Gather and Gatherv, 194
- 77 Gatherv, 191
- 78 Gatherv with datatype, 191, 192
- 79 Gatherv with struct datatype, 193
- 80 Gatherv with vector datatype, 194
- 81 Pack and Pack_size, 172
- 82 MPI_GET
- 83 Using Get, 545
- 84 Using Get with indexed datatype, 543
- 85 MPI_Get
- 86 Double buffer in RMA, 591
- 87 Get with fence, 589
- 88 Get with split phases, 590
- 89 Public and private memory in RMA, 584
- 90 Update location in separate memory
- 91 model, 582
- 92 Update location in unified memory model,
- 93 582
- 94 MPI_Get_accumulate
- 95 Counting semaphore (nonscalable), 592
- 96 Critical region with RMA, 592
- 97 Linked list in RMA, 595
- 98 MPI_GET_ADDRESS
- 99 Absolute addresses for struct, 808
- 100 Fortran 90
- 101 derived types, 786
- 102 Get_address, 135
- 103 Interlanguage communication, 814
- 104 MPI_Get_address
- 105 Pack/Unpack with struct datatype, 171
- 106 Using datatypes with array of structures,
- 107 161

Using datatypes with array of structures with absolute addresses, 163	
Using datatypes with unions, 164	
MPI_GET_COUNT	
Using Get_count and Get_elements, 149	
MPI_GET_ELEMENTS	
Using Get_count and Get_elements, 149	
MPI_GRAPH_CREATE	
Graph create, 373	
Graph creation, 387	
Graph creation and neighbors count, 386	
MPI_GRAPH_NEIGHBORS	
Graph creation and neighbors count, 386	
MPI_GRAPH_NEIGHBORS_COUNT	
Graph creation and neighbors count, 386	
MPI_Grequest_complete	
User-defined reduce, 603	
MPI_Grequest_start	
User-defined reduce, 603	
MPI_Group_excl	
Using Group_excl, 328	
MPI_Group_free	
Communication safety, 329	
Inter-communicator creation, 312	
Library Example #2, 331	
Using Group_excl, 328	
MPI_GROUP_FROM_SESSION_PSET	
Using process set query in group creation, 484	
MPI_Group_from_session_pset	
Creating a communicator using the Sessions Model, 480	
Using process set query in group creation, 482	
MPI_Group_incl	
Communication safety, 329	
Inter-communicator creation, 312	
Library Example #2, 331	
MPI_iallreduce	
Overlapping communicators and collectives, 289	
MPI_ialltoall	
Erroneous matching of blocking and nonblocking collectives, 288	
MPI_Ibarrier	
Erroneous matching of collectives, 287	
Mixing blocking and nonblocking collective operations, 286	
Mixing collective and point-to-point, 288	
Progress of nonblocking collectives, 288	
MPI_Ibcast	
Ibcast, 241	
Independence of nonblocking operations, 290	
Pipelining nonblocking collectives, 289	1
MPI_Info_create	2
Creating a communicator using the Sessions Model, 480	3 4
MPI_INFO_ENV	5
mpiexec and environment variables, 464	6
MPI_Info_set	7
Creating a communicator using the Sessions Model, 480	8
MPI_Init	9
Hello world, 29	10
Initializing MPI, 463	11
MPI_Intercomm_create	12
Three-Group “Pipeline”, 341	13
Three-Group “Ring”, 342	14
MPI_Iprobe	15
Finalize and cancel, 470	16
MPI_Irecv	17
Client server code with waitsome, 78	18
Client-server, 78	19
Message ordering for nonblocking operations, 71	20
Nonblocking point-to-point, 70	21
Nonblocking send and receive with request free, 71	22 23
Progress semantics, 72	24
MPI_Irecv	25
Mixing collective and point-to-point, 288	26
MPI_ISEND	27
Client server code with waitsome, 78	28
Client-server, 78	29
Message ordering for nonblocking operations, 71	30
Nonblocking point-to-point, 70	31
Nonblocking send and receive with request free, 71	32
MPI_NEIGHBOR_ALLGATHER	33
Neighbor allgather, 397	34
MPI_NEIGHBOR_ALLTOALL	35
Neighbor alltoall, 401	36
MPI_Neighbor_alltoall, 401	37
MPI_OP_CREATE	38
Defining a user function, 226	39
MPI_Op_create	40
Reduction with user-defined op, 225	41
User-defined operation with Scan, 236	42
MPI_Open_port	43
Client-server, 513	44
Name publishing, 513	45
MPI_Pack	46
Pack and Pack_size, 172	47
Pack/Unpack with struct datatype, 171	48
Using Pack, 171	
MPI_Pack_size	

- 1 Pack and Pack_size, [172](#)
- 2 MPI_Parrived
- 3 Partitioned communication with partial
- 4 completion, [111](#)
- 5 MPI_Pready
- 6 Partitioned communication, [100](#)
- 7 Partitioned communication using threads,
- 8 [108](#)
- 9 Partitioned communication with partial
- 10 completion, [111](#)
- 11 Partitioned communication with tasks,
- 12 [109](#)
- 13 MPI_Precv_init
- 14 Partitioned communication, [100](#)
- 15 Partitioned communication using threads,
- 16 [108](#)
- 17 Partitioned communication with partial
- 18 completion, [111](#)
- 19 Partitioned communication with tasks,
- 20 [109](#)
- 21 MPI_PROBE
- 22 Client-server (with error), [82](#)
- 23 Client-server with probe, [82](#)
- 24 MPI_Psend_init
- 25 Partitioned communication, [100](#)
- 26 Partitioned communication using threads,
- 27 [108](#)
- 28 Partitioned communication with partial
- 29 completion, [111](#)
- 30 Partitioned communication with tasks,
- 31 [109](#)
- 32 MPI_Publish_name
- 33 Name publishing, [513](#)
- 34 MPI_Put
- 35 Active target and local reads in RMA, [585](#)
- 36 Put with Fence, [589](#)
- 37 Put with PSCW, [590](#)
- 38 Read data in RMA, [583](#)
- 39 Read data in RMA (unsafe), [583](#)
- 40 Register and Compiler Optimization, [589](#)
- 41 RMA Lock and unlock, [572](#)
- 42 RMA Start and complete, [566](#)
- 43 MPI_RECV
- 44 Client-server (with error), [82](#)
- 45 Client-server with probe, [82](#)
- 46 Datatype matching, [43](#)
- 47 Datatypes, [44](#)
- 48 Errant message exchange, [53](#)
- Exchange relies on buffering, [54](#)
- Fortran CHARACTER, [44](#)
- Matching type with datatypes, [147](#)
- Message exchange, [53](#)
- Message matching, [52](#)
- Nonovertaking messages, [52](#)
- Progress semantics, [72](#)
- MPI_Recv
- Hello world, [29](#)
- Progress of nonblocking collectives, [288](#)
- MPI_REDUCE
- Defining a user function, [226](#)
- Reduction, [216](#)
- Reduction of a vector, [217](#)
- Reduction with maxloc, [220](#)
- MPI_Reduce
- Library Example #1, [330](#)
- Reduction with maxloc, [219](#)
- Reduction with minloc, [220](#)
- Reduction with user-defined op, [225](#)
- MPI_REQUEST_FREE
- Nonblocking send and receive with request
- free, [71](#)
- MPI_Request_free
- Finalize and request free, [469](#)
- MPI_Rget
- Requests in RMA, [594](#)
- MPI_Rput
- Requests in RMA, [594](#)
- MPI_Scan
- User-defined operation with Scan, [236](#)
- MPI_Scatter
- Scatter, [200](#)
- MPI_Scatterv
- Scatterv, [200](#)
- Scatterv with vector datatype, [201](#)
- MPI_SEND
- Client-server (with error), [82](#)
- Client-server with probe, [82](#)
- Datatype matching, [43](#)
- Datatypes, [44](#)
- Errant message exchange, [53](#)
- Exchange relies on buffering, [54](#)
- Fortran CHARACTER, [44](#)
- Matching type with datatypes, [147](#)
- Message exchange, [53](#)
- Progress semantics, [72](#)
- MPI_Send
- Hello world, [29](#)
- Measurement wrapper, [692](#)
- Mixing collective and point-to-point, [288](#)
- Pack/Unpack with struct datatype, [171](#)
- Progress of nonblocking collectives, [288](#)
- Using datatypes with array of structures,
- [161](#)
- Using datatypes with array of structures
- with absolute addresses, [163](#)
- Using datatypes with unions, [164](#)
- MPI_SENDRECV
- Neighbor alltoall, [401](#)

Nested vector datatypes, 160	
Send/receive of a 3D array, 158	
Transpose with datatypes, 160	
Using indexed datatype, 159	
MPI_SENDRECV_REPLACE	
Using Cart_shift, 390	
MPI_SESSION_FINALIZE	
Finalize in the Sessions Model, 476	
Using process set query in group creation, 484	
MPI_Session_finalize	
Creating a communicator using the Sessions Model, 480	
Using process set query in group creation, 482	
MPI_SESSION_GET_NTH_PSET	
Using process set query in group creation, 484	
MPI_Session_get_nth_pset	
Using process set query in group creation, 482	
MPI_SESSION_GET_NUM_PSETS	
Using process set query in group creation, 484	
MPI_Session_get_num_psets	
Using process set query in group creation, 482	
MPI_SESSION_INIT	
Using process set query in group creation, 484	
MPI_Session_init	
Creating a communicator using the Sessions Model, 480	
Using process set query in group creation, 482	
MPI_SSEND	
Message matching, 52	
Progress semantics, 72	
MPI_T_cvar_get_info	
Listing names of control variables, 705	
MPI_T_cvar_handle_alloc	
Reading a control variable, 708	
MPI_T_cvar_handle_free	
Reading a control variable, 708	
MPI_T_cvar_read	
Reading a control variable, 708	
MPI_T_finalize	
Basic usage of performance variables, 718	
MPI_T_init_thread	
Basic usage of performance variables, 718	
MPI_T_pvar_get_info	
Basic usage of performance variables, 718	
MPI_T_pvar_handle_alloc	
Basic usage of performance variables, 718	
MPI_T_pvar_handle_free	1
Basic usage of performance variables, 718	2
MPI_T_pvar_read	3
Basic usage of performance variables, 718	4
MPI_T_pvar_session_create	5
Basic usage of performance variables, 718	6
MPI_T_pvar_start	7
Basic usage of performance variables, 718	8
MPI_Test_cancelled	9
Finalize and cancel, 470	
MPI_TYPE_COMMIT	10
Fortran 90	11
derived types, 786	12
Nested vector datatypes, 160	13
Send/receive of a 3D array, 158	14
Transpose with datatypes, 160	15
Type_commit, 144	16
Using Get with indexed datatype, 543	17
Using indexed datatype, 159	18
MPI_Type_commit	19
Gather and Gatherv, 194	20
Gather with datatype, 190	21
Gatherv with datatype, 191 , 192	22
Gatherv with struct datatype, 193	23
Gatherv with vector datatype, 194	24
Pack/Unpack with struct datatype, 171	25
Scatterv with vector datatype, 201	26
User-defined operation with Scan, 236	27
Using datatypes with array of structures, 161	28
Using datatypes with array of structures with absolute addresses, 163	29
Using datatypes with unions, 164	30
MPI_TYPE_CONTIGUOUS	31
Matching type with datatypes, 147	32
Typemap for contiguous, 116	33
Typemap of nested datatypes, 138	34
Using Get_count and Get_elements, 149	35
MPI_Type_contiguous	36
Gather with datatype, 190	37
MPI_TYPE_CREATE_DARRAY	38
Datatypes for distributed arrays, 134	39
MPI_TYPE_CREATE_HVECTOR	40
Nested vector datatypes, 160	41
Send/receive of a 3D array, 158	42
MPI_Type_create_hvector	43
Using datatypes with array of structures, 161	44
Using datatypes with array of structures with absolute addresses, 163	45
MPI_TYPE_CREATE_INDEXED_BLOCK	46
Using Get with indexed datatype, 543	47
MPI_TYPE_CREATE_RESIZED	48
Fortran 90	

- 1 derived types, [786](#)
- 2 MPI_Type_create_resized
- 3 Using datatypes with unions, [164](#)
- 4 MPI_TYPE_CREATE_STRUCT
- 5 Fortran 90
- 6 derived types, [786](#)
- 7 Interlanguage communication, [814](#)
- 8 Transpose with datatypes, [160](#)
- 9 Typemap for create struct, [126](#)
- 10 Typemap of nested datatypes, [138](#)
- 11 MPI_Type_create_struct
- 12 Gather and Gatherv, [194](#)
- 13 Gatherv with struct datatype, [193](#)
- 14 Pack/Unpack with struct datatype, [171](#)
- 15 User-defined operation with Scan, [236](#)
- 16 Using datatypes with array of structures,
- 17 [161](#)
- 18 Using datatypes with array of structures
- 19 with absolute addresses, [163](#)
- 20 MPI_TYPE_EXTENT
- 21 Using Get with indexed datatype, [543](#)
- 22 MPI_TYPE_FREE
- 23 Using Get with indexed datatype, [543](#)
- 24 MPI_Type_get_contents
- 25 Decoding a datatype, [165](#)
- 26 MPI_Type_get_envelope
- 27 Decoding a datatype, [165](#)
- 28 MPI_TYPE_GET_EXTENT
- 29 Accumulate in RMA, [547](#)
- 30 Nested vector datatypes, [160](#)
- 31 Send/receive of a 3D array, [158](#)
- 32 Transpose with datatypes, [160](#)
- 33 Using Get, [545](#)
- 34 MPI_Type_get_extent
- 35 Using datatypes with array of structures,
- 36 [161](#)
- 37 MPI_TYPE_INDEXED
- 38 Typemap for indexed, [120](#)
- 39 Using indexed datatype, [159](#)
- 40 MPI_Type_indexed
- 41 Using datatypes with array of structures,
- 42 [161](#)
- 43 Using datatypes with array of structures
- 44 with absolute addresses, [163](#)
- 45 MPI_Type_size
- 46 Measurement wrapper, [692](#)
- 47 MPI_TYPE_VECTOR
- 48 Nested vector datatypes, [160](#)
- Send/receive of a 3D array, [158](#)
- Transpose with datatypes, [160](#)
- Typemap for vector, [117](#)
- MPI_Type_vector
- Gatherv with datatype, [191](#), [192](#)
- Gatherv with vector datatype, [194](#)
- Scatterv with vector datatype, [201](#)
- MPI_Unpack
- Pack and Pack_size, [172](#)
- Pack/Unpack with struct datatype, [171](#)
- MPI_Unpublish_name
- Name publishing, [513](#)
- MPI_USER_FUNCTION
- Defining a user function, [226](#)
- MPI_WAIT
- Client server code with waitsome, [78](#)
- Client-server, [78](#)
- File pointer update semantics, [640](#)
- Message ordering for nonblocking
- operations, [71](#)
- Nonblocking point-to-point, [70](#)
- Nonblocking send and receive with request
- free, [71](#)
- Progress semantics, [72](#)
- MPI_Wait
- Erroneous matching of blocking and
- nonblocking collectives, [288](#)
- Erroneous matching of collectives, [287](#)
- Mixing blocking and nonblocking
- collective operations, [286](#)
- Mixing collective and point-to-point, [288](#)
- Progress of nonblocking collectives, [288](#)
- MPI_Waitall
- Mixing collective and point-to-point, [288](#)
- Overlapping communicators and
- collectives, [289](#)
- Pipelining nonblocking collectives, [289](#)
- Requests in RMA, [594](#)
- MPI_WAITANY
- Client-server, [78](#)
- MPI_Waitany
- Requests in RMA, [594](#)
- MPI_WAITSOME
- Client server code with waitsome, [78](#)
- MPI_Win_attach
- Linked list in RMA, [595](#)
- MPI_Win_complete
- Active target and local reads in RMA, [585](#)
- Double buffer in RMA, [591](#)
- Get with split phases, [590](#)
- Public and private memory in RMA, [584](#)
- Put with PSCW, [590](#)
- RMA Start and complete, [566](#)
- MPI_WIN_CREATE
- Accumulate in RMA, [547](#)
- Using Get, [545](#)
- Using Get with indexed datatype, [543](#)
- MPI_Win_create_dynamic
- Linked list in RMA, [595](#)
- MPI_Win_detach

- Linked list in RMA, [595](#)
- MPI_WIN_FENCE
 - Accumulate in RMA, [547](#)
 - Using Get, [545](#)
 - Using Get with indexed datatype, [543](#)
- MPI_Win_fence
 - Get with fence, [589](#)
 - Put with Fence, [589](#)
 - Register and Compiler Optimization, [589](#)
- MPI_Win_flush
 - Counting semaphore (non-scalable), [592](#)
 - Critical region with Compare-and-Swap, [593](#)
 - Critical region with RMA, [592](#)
 - Linked list in RMA, [595](#)
 - Read data in RMA (unsafe), [583](#)
- MPI_Win_flush_all
 - Critical region with RMA, [592](#)
- MPI_Win_flush_local
 - Update location in unified memory model, [582](#)
- MPI_WIN_FREE
 - Accumulate in RMA, [547](#)
 - Using Get, [545](#)
- MPI_Win_lock
 - Active target and local reads in RMA, [585](#)
 - Public and private memory in RMA, [584](#)
 - Read data in RMA, [583](#)
 - RMA Lock and unlock, [572](#)
 - Update location in separate memory model, [582](#)
- MPI_Win_lock_all
 - Linked list in RMA, [595](#)
 - Read data in RMA (unsafe), [583](#)
 - Requests in RMA, [594](#)
 - Shared memory windows, [593](#)
- MPI_Win_post
 - Active target and local reads in RMA, [585](#)
 - Double buffer in RMA, [591](#)
 - Get with split phases, [590](#)
 - Public and private memory in RMA, [584](#)
 - Put with PSCW, [590](#)
- MPI_Win_start
 - Active target and local reads in RMA, [585](#)
 - Double buffer in RMA, [591](#)
 - Get with split phases, [590](#)
 - Public and private memory in RMA, [584](#)
 - Put with PSCW, [590](#)
 - RMA Start and complete, [566](#)
- MPI_Win_sync
 - shared memory windows, [593](#)
- MPI_Win_sync
 - Counting semaphore (non-scalable), [592](#)
- Critical region with Compare-and-Swap, [593](#)
- Critical region with RMA, [592](#)
- Update location in unified memory model, [582](#)
- MPI_Win_unlock
 - Active target and local reads in RMA, [585](#)
 - Public and private memory in RMA, [584](#)
 - Read data in RMA, [583](#)
 - RMA Lock and unlock, [572](#)
 - Update location in separate memory model, [582](#)
- MPI_Win_unlock_all
 - Linked list in RMA, [595](#)
 - Read data in RMA (unsafe), [583](#)
 - Requests in RMA, [594](#)
- MPI_Win_wait
 - Active target and local reads in RMA, [585](#)
 - Double buffer in RMA, [591](#)
 - Get with split phases, [590](#)
 - Public and private memory in RMA, [584](#)
 - Put with PSCW, [590](#)
- MPI_WTIME
 - Using MPI_Wtime, [451](#)
- MPI_Wtime
 - Measurement wrapper, [692](#)
- mpexec, [489](#)
- mpexec
 - mpexec and environment variables, [464](#)
 - Using mpexec
 - starting programs with separate argument lists, [489](#)
 - using -arch, [489](#)
 - using -configfile, [489](#)
 - using -host, [489](#)
 - using -n, [489](#)
- mpexec and environment variables
 - MPI_INFO_ENV, [464](#)
 - mpexec, [464](#)
- Name publishing
 - MPI_Comm_accept, [513](#)
 - MPI_Comm_connect, [513](#)
 - MPI_Open_port, [513](#)
 - MPI_Publish_name, [513](#)
 - MPI_Unpublish_name, [513](#)
- Neighbor allgather
 - MPI_NEIGHBOR_ALLGATHER, [397](#)
- Neighbor alltoall
 - MPI_CART_SHIFT, [401](#)
 - MPI_CARTDIM_GET, [401](#)
 - MPI_NEIGHBOR_ALLTOALL, [401](#)
 - MPI_SENDRECV, [401](#)
- Neighborhood collective communication, [421](#)

- 1 MPI_CART_GET, [421](#)
- 2 MPI_CART_SHIFT, [421](#)
- 3 MPI_DIMS_CREATE, [421](#)
- 4 Nested vector datatypes
 - 5 MPI_SENDRECV, [160](#)
 - 6 MPI_TYPE_COMMIT, [160](#)
 - 7 MPI_TYPE_CREATE_HVECTOR, [160](#)
 - 8 MPI_TYPE_GET_EXTENT, [160](#)
 - 9 MPI_TYPE_VECTOR, [160](#)
- 10 Nonblocking operations, [70, 71](#)
 - 11 message ordering, [71](#)
 - 12 progress, [72](#)
- 13 Nonblocking point-to-point
 - 14 MPI_IRecv, [70](#)
 - 15 MPI_ISEND, [70](#)
 - 16 MPI_WAIT, [70](#)
- 17 Nonblocking send and receive with request free
 - 18 MPI_IRecv, [71](#)
 - 19 MPI_ISEND, [71](#)
 - 20 MPI_REQUEST_FREE, [71](#)
 - 21 MPI_WAIT, [71](#)
- 22 Nondeterministic program with MPI_Bcast,
 - 23 [285](#)
- 24 Nondeterministic use of Bcast
 - 25 MPI_Bcast, [285](#)
- 26 Nonovertaking messages, [52](#)
 - 27 MPI_BSEND, [52](#)
 - 28 MPI_RECV, [52](#)
- 29 Overlapping communicators, [289](#)
- 30 Overlapping communicators and collectives
 - 31 MPI_Iallreduce, [289](#)
 - 32 MPI_Waitall, [289](#)
- 33 Pack and Pack_size
 - 34 MPI_Gather, [172](#)
 - 35 MPI_Gatherv, [172](#)
 - 36 MPI_Pack, [172](#)
 - 37 MPI_Pack_size, [172](#)
 - 38 MPI_Unpack, [172](#)
- 39 Pack/Unpack with struct datatype
 - 40 MPI_Get_address, [171](#)
 - 41 MPI_Pack, [171](#)
 - 42 MPI_Send, [171](#)
 - 43 MPI_Type_commit, [171](#)
 - 44 MPI_Type_create_struct, [171](#)
 - 45 MPI_Unpack, [171](#)
- 46 Partitioned communication
 - 47 Equal send/rcv partitioning, [108](#)
 - 48 MPI_Pready, [100](#)
 - MPI_Precv_init, [100](#)
 - MPI_Psend_init, [100](#)
 - Partial completion notification, [111](#)
 - Send with tasks, [109](#)
 - Simple example, [100](#)
- Partitioned communication using threads
 - MPI_Pready, [108](#)
 - MPI_Precv_init, [108](#)
 - MPI_Psend_init, [108](#)
- Partitioned communication with partial completion
 - MPI_Parrived, [111](#)
 - MPI_Pready, [111](#)
 - MPI_Precv_init, [111](#)
 - MPI_Psend_init, [111](#)
- Partitioned communication with tasks
 - MPI_Pready, [109](#)
 - MPI_Precv_init, [109](#)
 - MPI_Psend_init, [109](#)
- Pipelining nonblocking collective operations,
 - [289](#)
- Pipelining nonblocking collectives
 - MPI_Ibcast, [289](#)
 - MPI_Waitall, [289](#)
- Point-to-point
 - Hello world, [29](#)
- Profiling interface
 - implementation using the C macro preprocessor, [692](#)
 - implementation using weak symbols, [692](#)
 - measurement wrapper, [692](#)
- Progress of matching pairs, [52](#)
- Progress of nonblocking collective operations,
 - [288](#)
- Progress of nonblocking collectives
 - MPI_Ibarrier, [288](#)
 - MPI_Recv, [288](#)
 - MPI_Send, [288](#)
 - MPI_Wait, [288](#)
- Progress semantics
 - MPI_IRecv, [72](#)
 - MPI_RECV, [72](#)
 - MPI_SEND, [72](#)
 - MPI_SSEND, [72](#)
 - MPI_WAIT, [72](#)
- Public and private memory in RMA
 - MPI_Barrier, [584](#)
 - MPI_Get, [584](#)
 - MPI_Win_complete, [584](#)
 - MPI_Win_lock, [584](#)
 - MPI_Win_post, [584](#)
 - MPI_Win_start, [584](#)
 - MPI_Win_unlock, [584](#)
 - MPI_Win_wait, [584](#)
- Put with Fence
 - MPI_Put, [589](#)
 - MPI_Win_fence, [589](#)
- Put with PSCW

MPI_Put, 590		
MPI_Win_complete, 590		
MPI_Win_post, 590		
MPI_Win_start, 590		
MPI_Win_wait, 590		
Read data in RMA		
MPI_Barrier, 583		
MPI_Put, 583		
MPI_Win_lock, 583		
MPI_Win_unlock, 583		
Read data in RMA (unsafe)		
MPI_Barrier, 583		
MPI_Put, 583		
MPI_Win_flush, 583		
MPI_Win_lock_all, 583		
MPI_Win_unlock_all, 583		
Read to end of file		
MPI_FILE_CLOSE, 637		
MPI_FILE_OPEN, 637		
MPI_FILE_READ, 637		
MPI_FILE_SET_VIEW, 637		
Reading a control variable		
MPI_T_cvar_handle_alloc, 708		
MPI_T_cvar_handle_free, 708		
MPI_T_cvar_read, 708		
Recursive splitting of COMM_WORLD		
MPI_Comm_split_type, 321		
Reduction		
MPI_REDUCE, 216		
Reduction of a vector		
MPI_REDUCE, 217		
Reduction with maxloc		
MPI_REDUCE, 220		
MPI_Reduce, 219		
Reduction with minloc		
MPI_Reduce, 220		
Reduction with user-defined op		
MPI_Op_create, 225		
MPI_Reduce, 225		
Register and Compiler Optimization		
MPI_Put, 589		
MPI_Win_fence, 589		
Requests in RMA		
MPI_Rget, 594		
MPI_Rput, 594		
MPI_Waitall, 594		
MPI_Waitany, 594		
MPI_Win_lock_all, 594		
MPI_Win_unlock_all, 594		
RMA Lock and unlock		
MPI_Put, 572		
MPI_Win_lock, 572		
MPI_Win_unlock, 572		
RMA Start and complete		1
MPI_Put, 566		2
MPI_Win_complete, 566		3
MPI_Win_start, 566		4
Rules for finalize		5
MPI_Finalize, 469		6
Scatter		7
MPI_Scatter, 200		8
Scatterv		9
MPI_Scatterv, 200		10
Scatterv with vector datatype		11
MPI_Scatterv, 201		12
MPI_Type_commit, 201		13
MPI_Type_vector, 201		14
Send/receive of a 3D array		15
MPI_SENDRECV, 158		16
MPI_TYPE_COMMIT, 158		17
MPI_TYPE_CREATE_HVECTOR, 158		18
MPI_TYPE_GET_EXTENT, 158		19
MPI_TYPE_VECTOR, 158		20
Setting an attribute in Fortran and reading in C or Fortran		21
MPI_ATTR_GET, 812		22
MPI_COMM_GET_ATTR, 812		23
MPI_COMM_SET_ATTR, 812		24
Shared memory windows		25
MPI_F_SYNC_REG, 593		26
MPI_Win_lock_all, 593		27
MPI_Win_sync, 593		28
Splitting into NUMANode subcommunicators		29
MPI_Comm_split_type, 320		30
Subgroup cart process topology		31
MPI_CART_SUB, 391		32
Threads and MPI, 490		33
Three-Group “Pipeline”		34
MPI_Comm_split, 341		35
MPI_Intercomm_create, 341		36
Three-Group “Ring”		37
MPI_Comm_split, 342		38
MPI_Intercomm_create, 342		39
Tool information interface		40
basic usage of performance variables, 718		41
listing names of all control variables, 705		42
reading the value of a control variable, 708		43
Topologies, 421		44
Transpose with datatypes		45
MPI_SENDRECV, 160		46
MPI_TYPE_COMMIT, 160		47
MPI_TYPE_CREATE_STRUCT, 160		48
MPI_TYPE_GET_EXTENT, 160		
MPI_TYPE_VECTOR, 160		
Type_commit		

- 1 MPI_TYPE_COMMIT, 144
- 2 Typemap, 115–117, 120, 126, 134
- 3 Typemap for contiguous
 - 4 MPI_TYPE_CONTIGUOUS, 116
- 5 Typemap for create struct
 - 6 MPI_TYPE_CREATE_STRUCT, 126
- 7 Typemap for indexed
 - 8 MPI_TYPE_INDEXED, 120
- 9 Typemap for vector
 - 10 MPI_TYPE_VECTOR, 117
- 11 Typemap of nested datatypes
 - 12 MPI_TYPE_CONTIGUOUS, 138
 - 13 MPI_TYPE_CREATE_STRUCT, 138
- 14 Update location in separate memory model
 - 15 MPI_Barrier, 582
 - 16 MPI_Get, 582
 - 17 MPI_Win_lock, 582
 - 18 MPI_Win_unlock, 582
- 19 Update location in unified memory model
 - 20 MPI_Barrier, 582
 - 21 MPI_Get, 582
 - 22 MPI_Win_flush_local, 582
 - 23 MPI_Win_sync, 582
- 24 Use of Alloc_mem in Fortran
 - 25 MPI_ALLOC_MEM, 431
 - 26 MPI_FREE_MEM, 431
- 27 Use of Alloc_mem in Fortran with Cray pointers
 - 28 MPI_ALLOC_MEM, 432
 - 29 MPI_FREE_MEM, 432
- 30 User-defined operation with Scan
 - 31 MPI_Op_create, 236
 - 32 MPI_Scan, 236
 - 33 MPI_Type_commit, 236
 - 34 MPI_Type_create_struct, 236
- 35 User-defined reduce
 - 36 MPI_Grequest_complete, 603
 - 37 MPI_Grequest_start, 603
- 38 Using Alloc_mem
 - 39 MPI_Alloc_mem, 432
- 40 Using Cart_shift
 - 41 MPI_CART_COORDS, 390
 - 42 MPI_CART_RANK, 390
 - 43 MPI_CART_SHIFT, 390
 - 44 MPI_SENDRECV_REPLACE, 390
- 45 Using datatypes with array of structures
 - 46 MPI_Aint, 161
 - 47 MPI_Get_address, 161
 - 48 MPI_Send, 161
 - MPI_Type_commit, 161
 - MPI_Type_create_hvector, 161
 - MPI_Type_create_struct, 161
 - MPI_Type_get_extent, 161
 - MPI_Type_indexed, 161
- Using datatypes with array of structures with absolute addresses
 - MPI_Get_address, 163
 - MPI_Send, 163
 - MPI_Type_commit, 163
 - MPI_Type_create_hvector, 163
 - MPI_Type_create_struct, 163
 - MPI_Type_indexed, 163
- Using datatypes with unions
 - MPI_Get_address, 164
 - MPI_Send, 164
 - MPI_Type_commit, 164
 - MPI_Type_create_resized, 164
- Using Get
 - MPI_GET, 545
 - MPI_TYPE_GET_EXTENT, 545
 - MPI_WIN_CREATE, 545
 - MPI_WIN_FENCE, 545
 - MPI_WIN_FREE, 545
- Using Get with indexed datatype
 - MPI_GET, 543
 - MPI_TYPE_COMMIT, 543
 - MPI_TYPE_CREATE_INDEXED_BLOCK, 543
 - MPI_TYPE_EXTENT, 543
 - MPI_TYPE_FREE, 543
 - MPI_WIN_CREATE, 543
 - MPI_WIN_FENCE, 543
- Using Get_count and Get_elements
 - MPI_GET_COUNT, 149
 - MPI_GET_ELEMENTS, 149
 - MPI_TYPE_CONTIGUOUS, 149
- Using Group_excl
 - MPI_Comm_create, 328
 - MPI_Group_excl, 328
 - MPI_Group_free, 328
- Using indexed datatype
 - MPI_SENDRECV, 159
 - MPI_TYPE_COMMIT, 159
 - MPI_TYPE_INDEXED, 159
- Using MPI_Wtime
 - MPI_WTIME, 451
- Using mpiexec
 - starting programs with separate argument lists
 - mpiexec, 489
 - using -arch
 - mpiexec, 489
 - using -configfile
 - mpiexec, 489
 - using -host
 - mpiexec, 489
 - using -n

mpiexec, 489	1
Using Pack	2
MPI_Pack, 171	3
Using process set query in group creation	4
MPI_BARRIER, 484	5
MPI_COMM_CREATE_FROM_GROUP,	6
484	
MPI_GROUP_FROM_SESSION_PSET, 484	7
MPI_Group_from_session_pset, 482	8
MPI_SESSION_FINALIZE, 484	9
MPI_Session_finalize, 482	10
MPI_SESSION_GET_NTH_PSET, 484	11
MPI_Session_get_nth_pset, 482	12
MPI_SESSION_GET_NUM_PSETS, 484	13
MPI_Session_get_num_psets, 482	14
MPI_SESSION_INIT, 484	15
MPI_Session_init, 482	16
Virtual topologies, 421	17
	18
	19
	20
	21
	22
	23
	24
	25
	26
	27
	28
	29
	30
	31
	32
	33
	34
	35
	36
	37
	38
	39
	40
	41
	42
	43
	44
	45
	46
	47
	48

MPI Constant and Predefined Handle Index

This index lists predefined MPI constants and handles, including info keys and values. Underlined page numbers give the location of the primary definition or use of the indexed term.

"access_style", [620](#), [840](#)
"accumulate_ops", [524](#), [840](#)
"accumulate_ordering", [524](#), [586](#), [840](#)
"alloc_shared_noncontig", [528](#), [529](#), [840](#)
"appnum", [517](#), [840](#)
"arch", [464](#), [502](#), [840](#)
"argv", [464](#), [840](#)

"cb_block_size", [621](#), [840](#)
"cb_buffer_size", [621](#), [840](#)
"cb_nodes", [621](#), [840](#)
"chunked", [621](#), [840](#)
"chunked_item", [621](#), [840](#)
"chunked_size", [621](#), [840](#)
"collective_buffering", [620](#), [840](#)
"command", [464](#), [840](#)

"external32", [173](#), [662–668](#), [673](#), [775–777](#), [780](#),
[839](#), [1009](#), [1016](#)

"false", [325](#), [524](#), [620](#), [841](#)
"file", [464](#), [502](#), [840](#)
"file_perm", [620](#), [621](#), [840](#)
"filename", [621](#), [840](#)

"host", [464](#), [502](#), [840](#)
"hwloc://L3Cache", [476](#)

"internal", [662–664](#), [673](#), [839](#)
"io_node_list", [621](#), [840](#)
"ip_address", [513](#), [840](#)
"ip_port", [512](#), [840](#)

"maxprocs", [464](#), [840](#)
"mpi://", [476](#), [839](#)
"mpi://SELF", [476](#), [477](#), [612](#), [839](#)
"mpi://WORLD", [476](#), [477](#), [482](#), [488](#), [516](#), [839](#)
MPI::_LONG_LONG, [1009](#)
MPI::BOOL, [1008](#)
MPI::COMPLEX, [1008](#)
MPI::DOUBLE_COMPLEX, [1008](#)
MPI::F_COMPLEX16, [1009](#)
MPI::F_COMPLEX32, [1009](#)
MPI::F_COMPLEX4, [1009](#)
MPI::F_COMPLEX8, [1009](#)
MPI::INTEGER16, [1009](#)
MPI::LONG_DOUBLE_COMPLEX, [1008](#)
MPI::LONG_LONG, [1009](#)
MPI::REAL16, [1009](#)
MPI_2DOUBLE_PRECISION, [218](#), [219](#), [823](#)
MPI_2INT, [219](#), [823](#)
MPI_2INTEGER, [218](#), [219](#), [823](#)
MPI_2REAL, [218](#), [219](#), [823](#)
MPI_ADDRESS_KIND, [18](#), [19](#), [20](#), [33](#), [115](#), [134](#),
[155](#), [344](#), [430](#), [526](#), [528](#), [531](#), [769](#), [782](#),
[800](#), [801](#), [810](#), [820](#), [822](#)
MPI_AINT, [33](#), [115](#), [216](#), [532](#), [667](#), [800](#), [821](#),
[822](#), [1014–1016](#)
MPI_ANY_SOURCE, [35](#), [36](#), [40](#), [41](#), [51](#), [64–66](#),
[68](#), [80](#), [81](#), [83–85](#), [94](#), [325](#), [365](#), [428](#),
[819](#)
MPI_ANY_TAG, [18](#), [35](#), [36](#), [38](#), [40](#), [41](#), [64–66](#),
[68](#), [80](#), [81](#), [83–85](#), [87](#), [88](#), [94](#), [96](#), [97](#),
[325](#), [819](#), [1011](#)
MPI_APPNUM, [516](#), [517](#), [826](#)
MPI_ARGV_NULL, [19](#), [496](#), [497](#), [501](#), [781](#), [828](#)
MPI_ARGVS_NULL, [19](#), [501](#), [781](#), [828](#)
"mpi_assert_allow_overtaking", [325](#), [840](#), [1004](#)
"mpi_assert_exact_length", [325](#), [840](#), [1004](#)
"mpi_assert_no_any_source", [325](#), [840](#), [1004](#)
"mpi_assert_no_any_tag", [325](#), [840](#), [1004](#)
MPI_ASYNC_PROTECTS_NONBLOCKING, [19](#),
[593](#), [756](#), [757](#), [759](#), [761](#), [764](#), [771](#), [773](#),
[792](#), [820](#), [1013](#)
MPI_BAND, [215](#), [216](#), [824](#)
MPI_BOR, [215](#), [216](#), [824](#)
MPI_BOTTOM, [10](#), [19](#), [20](#), [39](#), [107](#), [134](#), [149](#),
[150](#), [181](#), [376](#), [378](#), [498](#), [532](#), [536](#), [757](#),
[760](#), [767](#), [781](#), [786](#), [788–791](#), [793–795](#),
[798](#), [808](#), [809](#), [814](#), [819](#), [1020](#)
MPI_BSEND_OVERHEAD, [57](#), [819](#)
MPI_BXOR, [215](#), [216](#), [824](#)
MPI_BYTE, [31](#), [32](#), [43–45](#), [173](#), [216](#), [610](#), [662](#),
[663](#), [667](#), [678](#), [814](#), [821](#), [822](#), [1017](#)
Fortran example usage, [44](#)
MPI_C_BOOL, [32](#), [215](#), [667](#), [821](#), [1009](#),

1014–1016	"MPI_COMM_SELF", 361, 839	1
MPI_C_COMPLEX, 32 , 215 , 667 , 821 , 1009 , 1014–1016	MPI_COMM_TYPE_HW_GUIDED, 319 , 320 , 823 , 1003	2
MPI_C_DOUBLE_COMPLEX, 32 , 215 , 667 , 821 , 1014–1016	MPI_COMM_TYPE_HW_UNGUIDED, 320 , 321 , 322 , 823 , 1003	3
MPI_C_FLOAT_COMPLEX, 32 , 215 , 667 , 821 , 1014–1016	MPI_COMM_TYPE_SHARED, 319 , 320 , 823 , 1011	4
MPI_C_LONG_DOUBLE_COMPLEX, 32 , 216 , 667 , 821 , 1014–1016	MPI_COMM_WORLD, 18 , 26 , 34 , 293 , 295 , 297 , 298 , 305–307 , 320–322 , 326 , 329 ,	5
MPI_CART, 381 , 825	337 , 361 , 426–428 , 435 , 447 , 461–463 ,	6
MPI_CHAR, 32 , 45 , 126 , 217 , 667 , 700 , 821 , 1014	465 , 467–470 , 473 , 487 , 488 , 493–496 ,	7
MPI_CHARACTER, 31 , 44 , 45 , 217 , 667 , 822 Fortran example usage, 44	499 , 501 , 515–518 , 661 , 683 , 684 , 706 , 715 , 753 , 802 , 814 , 823 , 1003 , 1019	8
MPI_COMBINER_CONTIGUOUS, 152 , 156 , 827	"MPI_COMM_WORLD", 361, 839	9
MPI_COMBINER_DARRAY, 152 , 158 , 827	MPI_COMPLEX, 31 , 215 , 665 , 667 , 773 , 822	10
MPI_COMBINER_DUP, 152 , 156 , 827	MPI_COMPLEX16, 32 , 216 , 668 , 822	11
MPI_COMBINER_F90_COMPLEX, 152 , 158 , 827	MPI_COMPLEX32, 33 , 216 , 668 , 822	12
MPI_COMBINER_F90_INTEGER, 152 , 158 , 827	MPI_COMPLEX4, 32 , 216 , 668 , 822	13
MPI_COMBINER_F90_REAL, 152 , 158 , 827	MPI_COMPLEX8, 32 , 216 , 668 , 822	14
MPI_COMBINER_HINDEXED, 22 , 152 , 157 , 827	MPI_CONGRUENT, 307 , 336 , 823	15
MPI_COMBINER_HINDEXED_BLOCK, 152 , 157 , 827 , 1010	MPI_CONVERSION_FN_NULL, 672 , 825 , 1007	16
MPI_COMBINER_HINDEXED_INTEGER, 22 , 752 , 1009	MPI_CONVERSION_FN_NULL_C, 672 , 800 , 825 , 1003	17
MPI_COMBINER_HVECTOR, 22 , 152 , 156 , 827	MPI_COUNT, 33 , 115 , 216 , 223 , 667 , 700 , 800 , 821 , 822 , 1010	18
MPI_COMBINER_HVECTOR_INTEGER, 22 , 752 , 1009	MPI_COUNT_KIND, 18 , 20 , 21 , 33 , 801 , 820 , 822 , 1010	19
MPI_COMBINER_INDEXED, 152 , 157 , 827	MPI_CXX_BOOL, 33 , 215 , 667 , 668 , 822 , 1008	20
MPI_COMBINER_INDEXED_BLOCK, 152 , 157 , 827	MPI_CXX_DOUBLE_COMPLEX, 33 , 216 , 667 , 668 , 822 , 1008	21
MPI_COMBINER_NAMED, 152 , 156 , 827	MPI_CXX_FLOAT_COMPLEX, 33 , 216 , 667 , 668 , 822 , 1008	22
MPI_COMBINER_RESIZED, 152 , 158 , 827	MPI_CXX_LONG_DOUBLE_COMPLEX, 33 , 216 , 667 , 668 , 822 , 1008	23
MPI_COMBINER_STRUCT, 22 , 152 , 157 , 827	MPI_DATATYPE_NULL, 145 , 363 , 824 , 1001	24
MPI_COMBINER_STRUCT_INTEGER, 22 , 752 , 1009	"MPI_DATATYPE_NULL", 363, 839	25
MPI_COMBINER_SUBARRAY, 152 , 157 , 827	MPI_DISPLACEMENT_CURRENT, 623 , 828 , 1020	26
MPI_COMBINER_VECTOR, 152 , 156 , 827	MPI_DIST_GRAPH, 381 , 825 , 1015	27
MPI_COMM_DUP_FN, 22 , 347 , 348 , 762 , 825 , 1007 , 1013	MPI_DISTRIBUTE_BLOCK, 131 , 132 , 828	28
MPI_COMM_NULL, 295 , 311 , 312 , 314–316 , 319 , 320 , 323 , 324 , 339 , 361 , 371 , 373 , 499 , 518–520 , 803 , 824 , 1001 , 1018	MPI_DISTRIBUTE_CYCLIC, 131 , 132 , 828	29
"MPI_COMM_NULL", 361, 839	MPI_DISTRIBUTE_DFLT_DARG, 131 , 132 , 828	30
MPI_COMM_NULL_COPY_FN, 22 , 23 , 347 , 348 , 758 , 809 , 825 , 1007 , 1013	MPI_DISTRIBUTE_NONE, 131 , 132 , 828	31
MPI_COMM_NULL_DELETE_FN, 22 , 347 , 348 , 825 , 1007	MPI_DOUBLE, 32 , 215 , 667 , 700 , 709 , 710 , 773 , 821	32
"MPI_COMM_PARENT", 361, 839	MPI_DOUBLE_COMPLEX, 31 , 216 , 665 , 667 , 773 , 822	33
MPI_COMM_SELF, 25 , 295 , 314 , 326 , 343 , 361 , 432 , 462 , 463 , 468 , 470 , 472 , 518 , 612 , 753 , 823 , 1003 , 1016	MPI_DOUBLE_INT, 219 , 823	34
	MPI_DOUBLE_PRECISION, 31 , 215 , 667 , 773 , 822	35
	MPI_DUP_FN, 22 , 347 , 744 , 826	36
	MPI_ERR_ACCESS, 444 , 615 , 685 , 818	37
	MPI_ERR_AMODE, 444 , 613 , 685 , 818	38
	MPI_ERR_ARG, 444 , 817	39

1	MPI_ERR_ASSERT, 444 , 577 , 818	MPI_ERR_TOPOLOGY, 445 , 817
2	MPI_ERR_BAD_FILE, 444 , 685 , 818	MPI_ERR_TRUNCATE, 445 , 817
3	MPI_ERR_BASE, 431 , 444 , 577 , 818	MPI_ERR_TYPE, 152 , 154 , 445 , 684 , 817
4	MPI_ERR_BUFFER, 444 , 817	MPI_ERR_UNKNOWN, 443 , 445 , 817
5	MPI_ERR_COMM, 444 , 817	MPI_ERR_UNSUPPORTED_DATAREP, 445 , 685 , 818
6	MPI_ERR_CONVERSION, 444 , 672 , 685 , 818	MPI_ERR_UNSUPPORTED_OPERATION, 445 , 685 , 818
7	MPI_ERR_COUNT, 444 , 817	MPI_ERR_VALUE_TOO_LARGE, 445 , 670 , 818 , 1003
8	MPI_ERR_DIMS, 444 , 817	MPI_ERR_WIN, 445 , 577 , 818
9	MPI_ERR_DISP, 444 , 577 , 818	MPI_ERRCODES_IGNORE, 19 , 498 , 781 , 828
10	MPI_ERR_DUP_DATAREP, 444 , 669 , 685 , 818	MPI_ERRHANDLER_NULL, 442 , 824
11	MPI_ERR_ERRHANDLER, 444 , 818 , 1002	MPI_ERROR, 37 , 68 , 238 , 552 , 805 , 820 , 1006 , 1012
12	MPI_ERR_FILE, 444 , 685 , 818	MPI_ERRORS_ABORT, 433 , 463 , 503 , 820 , 1003
13	MPI_ERR_FILE_EXISTS, 444 , 685 , 818	"mpi_errors_abort", 502 , 841 , 1005
14	MPI_ERR_FILE_IN_USE, 444 , 615 , 685 , 818	MPI_ERRORS_ARE_FATAL, 433 , 434 , 449 , 502 , 577 , 683 , 820 , 1003
15	MPI_ERR_GROUP, 444 , 817	"mpi_errors_are_fatal", 502 , 841 , 1005
16	MPI_ERR_IN_STATUS, 37 , 39 , 68 , 75 , 77 , 435 , 443 , 444 , 602 , 627 , 818	MPI_ERRORS_RETURN, 433 , 434 , 450 , 487 , 503 , 684 , 814 , 820
17	MPI_ERR_INFO, 444 , 818	"mpi_errors_return", 502 , 841 , 1005
18	MPI_ERR_INFO_KEY, 444 , 455 , 818	MPI_F08_STATUS_IGNORE, 806 , 829 , 1012
19	MPI_ERR_INFO_NOKEY, 444 , 455 , 818	MPI_F08_STATUSES_IGNORE, 806 , 829 , 1012
20	MPI_ERR_INFO_VALUE, 444 , 455 , 818	MPI_F_ERROR, 805 , 820 , 1002
21	MPI_ERR_INTERN, 433 , 444 , 817	MPI_F_SOURCE, 805 , 820 , 1002
22	MPI_ERR_IO, 444 , 685 , 818	MPI_F_STATUS_IGNORE, 805 , 829
23	MPI_ERR_KEYVAL, 358 , 444 , 818	MPI_F_STATUS_SIZE, 18 , 805 , 820 , 1002
24	MPI_ERR_LASTCODE, 443 , 445 , 447 , 448 , 740 , 819	MPI_F_STATUSES_IGNORE, 805 , 829
25	MPI_ERR_LOCKTYPE, 444 , 577 , 818	MPI_F_TAG, 805 , 820 , 1002
26	MPI_ERR_NAME, 444 , 512 , 818	MPI_FILE_NULL, 614 , 684 , 824
27	MPI_ERR_NO_MEM, 430 , 444 , 818	MPI_FLOAT, 32 , 126 , 213 , 215 , 664 , 667 , 821
28	MPI_ERR_NO_SPACE, 444 , 685 , 818	MPI_FLOAT_INT, 15 , 219 , 823
29	MPI_ERR_NO_SUCH_FILE, 444 , 614 , 685 , 818	MPI_GRAPH, 381 , 825
30	MPI_ERR_NOT_SAME, 444 , 685 , 818	MPI_GROUP_EMPTY, 294 , 300 , 301 , 311 , 312 , 314 , 323 , 339 , 824
31	MPI_ERR_OP, 445 , 577 , 817	MPI_GROUP_NULL, 294 , 304 , 824
32	MPI_ERR_OTHER, 443 , 445 , 817	MPI_HOST, 427 , 823
33	MPI_ERR_PENDING, 75 , 445 , 817	"mpi_hw_resource_type", 319 – 321 , 840 , 1003
34	MPI_ERR_PORT, 445 , 509 , 818	MPI_IDENT, 298 , 307 , 823
35	MPI_ERR_PROC_ABORTED, 445 , 487 , 818 , 1004	MPI_IN_PLACE, 19 , 180 , 209 , 760 , 781 , 819
36	MPI_ERR_QUOTA, 445 , 685 , 818	MPI_INFO_ENV, 458 , 464 , 473 , 823 , 1011
37	MPI_ERR_RANK, 445 , 577 , 817	MPI_INFO_NULL, 319 , 379 , 430 , 458 , 498 , 507 , 613 , 614 , 624 , 824
38	MPI_ERR_READ_ONLY, 445 , 685 , 818	"mpi_initial_errhandler", 463 , 464 , 502 , 840 , 1005
39	MPI_ERR_REQUEST, 445 , 817	MPI_INT, 15 , 32 , 114 , 215 , 664 , 665 , 667 , 699 , 700 , 703 , 709 , 712 , 773 , 814 , 815 , 821
40	MPI_ERR_RMA_ATTACH, 445 , 577 , 818	MPI_INT16_T, 32 , 215 , 667 , 821 , 1014 – 1016
41	MPI_ERR_RMA_CONFLICT, 445 , 577 , 818	MPI_INT32_T, 32 , 215 , 667 , 700 , 821 , 1014 – 1016
42	MPI_ERR_RMA_FLAVOR, 445 , 531 , 577 , 818	
43	MPI_ERR_RMA_RANGE, 445 , 577 , 818	
44	MPI_ERR_RMA_SHARED, 445 , 577 , 818	
45	MPI_ERR_RMA_SYNC, 445 , 577 , 818	
46	MPI_ERR_ROOT, 445 , 817	
47	MPI_ERR_SERVICE, 445 , 512 , 818	
48	MPI_ERR_SESSION, 445 , 818 , 1005	
	MPI_ERR_SIZE, 445 , 577 , 818	
	MPI_ERR_SPAWN, 445 , 497 , 498 , 818	
	MPI_ERR_TAG, 445 , 817	

MPI_INT64_T, 32 , 215 , 667 , 700 , 821 , 1014–1016	MPI_MODE_DELETE_ON_CLOSE, 612 , 613 , 614 , 827	1
MPI_INT8_T, 32 , 215 , 667 , 821 , 1014–1016	MPI_MODE_EXCL, 612 , 613 , 827	2
MPI_INTEGER, 31 , 43 , 215 , 667 , 773 , 815 , 822	MPI_MODE_NOCHECK, 571 , 575 , 576 , 827	3
MPI_INTEGER1, 32 , 215 , 668 , 822	MPI_MODE_NOPRECEDE, 565 , 575 , 576 , 827	4
MPI_INTEGER16, 215 , 668 , 822	MPI_MODE_NOPUT, 575 , 576 , 827	5
MPI_INTEGER2, 32 , 215 , 665 , 668 , 822	MPI_MODE_NOSTORE, 575 , 576 , 827	6
MPI_INTEGER4, 32 , 215 , 668 , 822	MPI_MODE_NOSUCCEED, 575 , 576 , 827	7
MPI_INTEGER8, 32 , 215 , 668 , 777 , 822	MPI_MODE_RDONLY, 612 , 613 , 618 , 827	8
MPI_INTEGER_KIND, 18 , 820	MPI_MODE_RDWR, 612 , 613 , 827	9
MPI_IO, 427 , 823	MPI_MODE_SEQUENTIAL, 612 , 613 , 615 , 616 , 623 , 628 , 636 , 651 , 676 , 827 , 1020	10
MPI_KEYVAL_INVALID, 347 , 348–350 , 819	MPI_MODE_UNIQUE_OPEN, 612 , 613 , 827	11
MPI_LAND, 215 , 216 , 824	MPI_MODE_WRONLY, 612 , 613 , 827	12
MPI_LASTUSED_CODE, 447 , 826	MPI_NO_OP, 524 , 550 , 551 , 824 , 1007	13
MPI_LB, 22 , 752 , 1009	MPI_NULL_COPY_FN, 22 , 347 , 744 , 826	14
MPI_LOCK_EXCLUSIVE, 570 , 819	MPI_NULL_DELETE_FN, 22 , 347 , 744 , 826	15
MPI_LOCK_SHARED, 570 , 571 , 819	MPI_OFFSET, 33 , 216 , 667 , 821 , 822 , 1014–1016	16
MPI_LOGICAL, 31 , 215 , 667 , 822	MPI_OFFSET_KIND, 19 , 20 , 33 , 678 , 782 , 801 , 802 , 820 , 822 , 828	17
MPI_LONG, 32 , 215 , 667 , 821	MPI_OP_NULL, 225 , 824	18
MPI_LONG_DOUBLE, 32 , 215 , 667 , 821	MPI_ORDER_C, 18 , 128 , 131 , 132 , 828	19
MPI_LONG_DOUBLE_INT, 219 , 823	MPI_ORDER_FORTRAN, 18 , 128 , 131 , 132 , 828	20
MPI_LONG_INT, 219 , 823	MPI_PACKED, 15 , 31 , 32 , 43 , 167 , 169 , 173 , 666 , 667 , 814 , 821 , 822	21
MPI_LONG_LONG, 32 , 215 , 821 , 1017	MPI_PROC_NULL, 30 , 34 , 36 , 81 , 84 , 85 , 87 , 88 , 96 , 97 , 183 , 185 , 187 , 189 , 197 , 199 , 214 , 297 , 390 , 394 , 395 , 427 , 428 , 530 , 531 , 539 , 819 , 1010 , 1017 , 1019	22
MPI_LONG_LONG_INT, 32 , 215 , 667 , 821 , 1017	MPI_PROD, 215 , 216 , 824	23
MPI_LOR, 215 , 216 , 824	MPI_REAL, 31 , 43 , 215 , 665 , 667 , 773 , 774 , 780 , 822	24
MPI_LXOR, 215 , 216 , 824	MPI_REAL16, 32 , 215 , 668 , 822	25
MPI_MAX, 213 , 215 , 216 , 235 , 824	MPI_REAL2, 32 , 215 , 668 , 822	26
MPI_MAX_DATAREP_STRING, 18 , 625 , 669 , 820	MPI_REAL4, 32 , 215 , 668 , 773 , 777 , 822	27
MPI_MAX_ERROR_STRING, 18 , 442 , 448 , 820	MPI_REAL8, 32 , 215 , 668 , 773 , 822 , 1015	28
MPI_MAX_INFO_KEY, 18 , 444 , 453 , 456 , 747 , 748 , 820	MPI_REPLACE, 547 , 548 , 550 , 551 , 593 , 824 , 1016 , 1020	29
MPI_MAX_INFO_VAL, 18 , 444 , 453 , 820	MPI_REQUEST_NULL, 67 , 68–70 , 73–77 , 602 , 824	30
MPI_MAX_LIBRARY_VERSION_STRING, 18 , 426 , 820 , 1010	MPI_ROOT, 183 , 819	31
MPI_MAX_OBJECT_NAME, 18 , 360–363 , 820 , 1011 , 1018	MPI_SEEK_CUR, 644 , 652 , 828	32
MPI_MAX_PORT_NAME, 18 , 507 , 820	MPI_SEEK_END, 644 , 652 , 828	33
MPI_MAX_PROCESSOR_NAME, 18 , 429 , 820 , 1019	MPI_SEEK_SET, 644 , 652 , 828	34
MPI_MAX_PSET_NAME_LEN, 479 , 820 , 1005	MPI_SESSION_NULL, 475 , 824 , 1005	35
MPI_MAX_STRINGTAG_LEN, 323 , 339 , 820 , 1005	"mpi_shared_memory", 320 , 841 , 1003	36
MPI_MAXLOC, 215 , 217 , 218 , 221 , 824	MPI_SHORT, 32 , 215 , 667 , 821	37
MPI_MESSAGE_NO_PROC, 84 , 85 , 87 , 88 , 97 , 819 , 1010	MPI_SHORT_INT, 219 , 823	38
MPI_MESSAGE_NULL, 84 , 86–88 , 824 , 1010	MPI_SIGNED_CHAR, 32 , 215 , 217 , 667 , 821 , 1017	39
MPI_MIN, 215 , 216 , 824	MPI_SIMILAR, 298 , 307 , 336 , 823	40
"mpi_minimum_memory_alignment", 430 , 527 , 529 , 840 , 1004	"mpi_size", 477 , 480 , 484 , 840 , 1005	41
MPI_MINLOC, 215 , 217 , 218 , 221 , 824	MPI_SOURCE, 37 , 238 , 805 , 820 , 1006 , 1012	42
MPI_MODE_APPEND, 612 , 613 , 827		43
MPI_MODE_CREATE, 612 , 613 , 621 , 827		44
		45
		46
		47
		48

1	MPI_STATUS_IGNORE, 10 , 19 , 39 , 601 , 627 ,	MPI_T_ERR_OUT_OF_HANDLES, 741 , 819
2	760 , 781 , 805 , 806 , 814 , 828 , 829 , 1010	MPI_T_ERR_OUT_OF_SESSIONS, 741 , 819
3	MPI_STATUS_SIZE, 18 , 37 , 761 , 820 , 1012	MPI_T_ERR_PVAR_NO_ATOMIC, 718 , 741 ,
4	MPI_STATUSES_IGNORE, 18 , 19 , 39 , 601 , 602 ,	819
5	781 , 805 , 806 , 828 , 829	MPI_T_ERR_PVAR_NO_STARTSTOP, 716 ,
6	MPI_SUBARRAYS_SUPPORTED, 19 , 756 , 757 ,	741 , 819
7	760–764 , 768–771 , 783 , 784 , 820 , 1012	MPI_T_ERR_PVAR_NO_WRITE, 717 , 741 , 819
8	MPI_SUBVERSION, 18 , 426 , 829	MPI_T_PVAR_ALL_HANDLES, 715 , 716–718 ,
9	MPI_SUCCESS, 23 , 68 , 75 , 77 , 346 , 347 ,	830
10	349–352 , 355 , 356 , 372 , 443 , 444 ,	MPI_T_PVAR_CLASS_AGGREGATE, 710 , 830
11	448–450 , 463 , 498 , 672 , 695 , 704 , 713 ,	MPI_T_PVAR_CLASS_COUNTER, 710 , 830
12	716 , 717 , 723 , 727 , 731 , 738 , 740 , 741 ,	MPI_T_PVAR_CLASS_GENERIC, 710 , 830
13	744 , 817 , 1004	MPI_T_PVAR_CLASS_HIGHWATERMARK,
14	MPI_SUM, 215 , 216 , 547 , 814 , 824	709 , 830
15	MPI_T_BIND_MPI_COMM, 697 , 830	MPI_T_PVAR_CLASS_LEVEL, 709 , 830
16	MPI_T_BIND_MPI_DATATYPE, 697 , 830	MPI_T_PVAR_CLASS_LOWWATERMARK,
17	MPI_T_BIND_MPI_ERRHANDLER, 697 , 830	710 , 830
18	MPI_T_BIND_MPI_FILE, 697 , 830	MPI_T_PVAR_CLASS_PERCENTAGE, 709 , 830
19	MPI_T_BIND_MPI_GROUP, 697 , 830	MPI_T_PVAR_CLASS_SIZE, 709 , 830
20	MPI_T_BIND_MPI_INFO, 697 , 830	MPI_T_PVAR_CLASS_STATE, 709 , 830
21	MPI_T_BIND_MPI_MESSAGE, 697 , 830	MPI_T_PVAR_CLASS_TIMER, 710 , 830
22	MPI_T_BIND_MPI_OP, 697 , 830	MPI_T_PVAR_HANDLE_NULL, 715 , 829
23	MPI_T_BIND_MPI_REQUEST, 697 , 830	MPI_T_PVAR_SESSION_NULL, 714 , 829
24	MPI_T_BIND_MPI_SESSION, 697 , 830 , 1005	MPI_T_SCOPE_ALL, 704 , 830
25	MPI_T_BIND_MPI_WIN, 697 , 830	MPI_T_SCOPE_ALL_EQ, 704 , 707 , 830
26	MPI_T_BIND_NO_OBJECT, 697 , 703 , 706 ,	MPI_T_SCOPE_CONSTANT, 704 , 830
27	712 , 714 , 727 , 728 , 830	MPI_T_SCOPE_GROUP, 704 , 830
28	MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE,	MPI_T_SCOPE_GROUP_EQ, 704 , 707 , 830
29	723 , 724 , 831	MPI_T_SCOPE_LOCAL, 704 , 830
30	MPI_T_CB_REQUIRE_MPI_RESTRICTED, 723 ,	MPI_T_SCOPE_READONLY, 704 , 830
31	831	MPI_T_SOURCE_ORDERED, 722 , 831
32	MPI_T_CB_REQUIRE_NONE, 723 , 831	MPI_T_SOURCE_UNORDERED, 722 , 831
33	MPI_T_CB_REQUIRE_THREAD_SAFE, 723 ,	MPI_T_VERBOSENESS_MPIDEV_ALL, 696 , 829
34	723 , 724 , 831	MPI_T_VERBOSENESS_MPIDEV_BASIC, 696 ,
35	MPI_T_CVAR_HANDLE_NULL, 706 , 829	829
36	MPI_T_ENUM_NULL, 703 , 712 , 726 , 829	MPI_T_VERBOSENESS_MPIDEV_DETAIL, 696 ,
37	MPI_T_ERR_CANNOT_INIT, 741 , 819	829
38	MPI_T_ERR_CVAR_SET_NEVER, 707 , 741 , 819	MPI_T_VERBOSENESS_TUNER_ALL, 696 , 829
39	MPI_T_ERR_CVAR_SET_NOT_NOW, 707 , 741 ,	MPI_T_VERBOSENESS_TUNER_BASIC, 696 , 829
40	819	MPI_T_VERBOSENESS_TUNER_DETAIL, 696 ,
41	MPI_T_ERR_INVALID, 741 , 819 , 1008	829
42	MPI_T_ERR_INVALID_HANDLE, 715 , 731 , 741 ,	MPI_T_VERBOSENESS_USER_ALL, 696 , 829
43	819	MPI_T_VERBOSENESS_USER_BASIC, 696 , 829
44	MPI_T_ERR_INVALID_INDEX, 22 , 723 , 727 ,	MPI_T_VERBOSENESS_USER_DETAIL, 696 , 829
45	741 , 749 , 819 , 1005	MPI_TAG, 37 , 238 , 805 , 820 , 1006 , 1012
46	MPI_T_ERR_INVALID_ITEM, 22 , 749 , 819 ,	MPI_TAG_UB, 34 , 427 , 810 , 813 , 823
47	1005	MPI_THREAD_FUNNELED, 465 , 827
48	MPI_T_ERR_INVALID_NAME, 704 , 713 , 727 ,	"MPI_THREAD_FUNNELED", 474 , 841
	738 , 741 , 819 , 1008	MPI_THREAD_MULTIPLE, 465 , 466 , 468 , 827 ,
	MPI_T_ERR_INVALID_SESSION, 741 , 819	1008
	MPI_T_ERR_MEMORY, 741 , 819	"MPI_THREAD_MULTIPLE", 474 , 841
	MPI_T_ERR_NOT_ACCESSIBLE, 724 , 741 , 819	MPI_THREAD_SERIALIZED, 465 , 827
	MPI_T_ERR_NOT_INITIALIZED, 741 , 819	"MPI_THREAD_SERIALIZED", 474 , 841
	MPI_T_ERR_NOT_SUPPORTED, 723 , 741 , 819	MPI_THREAD_SINGLE, 465 , 466 , 475 , 827

"MPI_THREAD_SINGLE", 474 , 841	"mpix://UNIVERSE", 476	1
MPI_TYPE_DUP_FN, 355 , 825 , 1007	"native", 662 – 664 , 673 , 839	2
MPI_TYPE_NULL_COPY_FN, 355 , 825 , 1007	"nb_proc", 621 , 840	3
MPI_TYPE_NULL_DELETE_FN, 355 , 825 , 1007 , 1013	"no_locks", 524 , 535 , 840 , 1002	4
MPI_TYPECLASS_COMPLEX, 779 , 828	"none", 586 , 841	5
MPI_TYPECLASS_INTEGER, 779 , 828	"num_io_nodes", 621 , 840	6
MPI_TYPECLASS_REAL, 779 , 828	"path", 502 , 840	7
MPI_UB, 4 , 22 , 752 , 1009	"random", 620 , 841	8
MPI_UINT16_T, 32 , 215 , 667 , 821 , 1014 – 1016	"rar", 586 , 841	9
MPI_UINT32_T, 32 , 215 , 667 , 700 , 821 , 1014 – 1016	"rar,raw,war,waw", 524 , 586	10
MPI_UINT64_T, 32 , 215 , 667 , 700 , 821 , 1014 – 1016	"rar,waw", 586	11
MPI_UINT8_T, 32 , 215 , 667 , 821 , 1014 – 1016	"raw", 586 , 841	12
MPI_UNDEFINED, 21 , 38 , 56 , 73 , 74 , 77 , 78 , 138 , 141 , 143 , 148 , 170 , 296 , 297 , 315 , 316 , 319 , 381 , 392 , 393 , 665 , 670 , 774 , 775 , 819 , 1010 , 1017	"read_mostly", 620 , 841	13
MPI_UNEQUAL, 298 , 307 , 336 , 823	"read_once", 620 , 841	14
MPI_UNIVERSE_SIZE, 494 , 515 , 516 , 826	"reverse_sequential", 620 , 841	15
MPI_UNSIGNED, 32 , 215 , 667 , 700 , 709 , 710 , 821	"same_disp_unit", 524 , 840 , 1008	16
MPI_UNSIGNED_CHAR, 32 , 215 , 217 , 667 , 821	"same_op", 524 , 841	17
MPI_UNSIGNED_LONG, 32 , 215 , 667 , 700 , 709 , 710 , 821	"same_op_no_op", 524 , 841	18
MPI_UNSIGNED_LONG_LONG, 32 , 215 , 667 , 700 , 709 , 710 , 821 , 1017	"same_size", 524 , 840 , 1006	19
MPI_UNSIGNED_SHORT, 32 , 215 , 667 , 821	"sequential", 620 , 841	20
MPI_UNWEIGHTED, 19 , 376 , 378 , 380 , 387 , 389 , 781 , 828 , 1009 , 1015	"soft", 464 , 498 , 503 , 840	21
MPI_VAL, 16 , 803	"striping_factor", 621 , 840	22
MPI_VERSION, 18 , 426 , 829	"striping_unit", 622 , 840	23
MPI_WCHAR, 32 , 217 , 363 , 665 , 667 , 821 , 1017	"thread_level", 464 , 474 , 840	24
MPI_WEIGHTS_EMPTY, 19 , 376 , 378 , 781 , 828 , 1009	"true", 325 , 535 , 620 , 841 , 1002	25
MPI_WIN_BASE, 535 , 536 , 813 , 826	"war", 586 , 841	26
MPI_WIN_CREATE_FLAVOR, 535 , 536 , 826	"waw", 586 , 841	27
MPI_WIN_DISP_UNIT, 535 , 536 , 826	"wdir", 464 , 502 , 840	28
MPI_WIN_DUP_FN, 351 , 825 , 1007	"write_mostly", 620 , 841	29
MPI_WIN_FLAVOR_ALLOCATE, 536 , 826	"write_once", 620 , 841	30
MPI_WIN_FLAVOR_CREATE, 536 , 826		31
MPI_WIN_FLAVOR_DYNAMIC, 536 , 826		32
MPI_WIN_FLAVOR_SHARED, 531 , 536 , 826		33
MPI_WIN_MODEL, 535 , 536 , 559 , 826		34
MPI_WIN_NULL, 364 , 535 , 824 , 1001		35
"MPI_WIN_NULL", 364 , 839		36
MPI_WIN_NULL_COPY_FN, 351 , 825 , 1007		37
MPI_WIN_NULL_DELETE_FN, 351 , 825 , 1007		38
MPI_WIN_SEPARATE, 536 , 559 , 560 , 579 , 826		39
MPI_WIN_SIZE, 535 , 536 , 826		40
MPI_WIN_UNIFIED, 536 , 559 , 580 , 588 , 826		41
MPI_WTIME_IS_GLOBAL, 427 , 428 , 451 , 810 , 823		42
		43
		44
		45
		46
		47
		48

MPI Declarations Index

This index refers to declarations needed in C and Fortran, such as address kind integers, handles, etc. The underlined page numbers is the “main” reference (sometimes there are more than one when key concepts are discussed in multiple areas). Fortran defined types are shown as `TYPE(name)`.

MPI_Aint, [19](#), [20](#), [33](#), [115](#), [118](#), [121](#), [125](#),
[134–137](#), [140–143](#), [153](#), [173](#), [175](#), [176](#),
[522](#), [525](#), [527](#), [531](#), [532](#), [536](#), [539](#), [542](#),
[545](#), [548](#), [550](#), [551](#), [553–555](#), [557](#), [664](#),
[669](#), [722](#), [782](#), [800](#), [810](#), [821](#), [831](#)
C example usage, [161](#)
MPI_Comm, [16](#), [30](#), [298](#), [305](#), [306](#), [308–310](#),
[313](#), [315](#), [318](#), [324–326](#), [335](#), [336](#), [338](#),
[340](#), [345](#), [346](#), [348–350](#), [478](#), [823](#), [824](#),
[831](#)
TYPE(MPI_Comm), [30](#), [313](#), [338](#), [768](#), [823](#),
[824](#), [832](#)
MPI_Count, [20](#), [21](#), [33](#), [721](#), [722](#), [800](#), [821](#), [831](#),
[1010](#)
MPI_Datatype, [115](#), [355](#), [790](#), [821–824](#), [831](#)
TYPE(MPI_Datatype), [115](#), [821–824](#), [832](#)
MPI_Errhandler, [434](#), [435–442](#), [804](#), [820](#), [824](#),
[831](#)
TYPE(MPI_Errhandler), [434](#), [820](#), [824](#), [832](#)
MPI_F08_status, [806](#), [829](#), [831](#), [1012](#)
MPI_File, [439](#), [611](#), [613](#), [615–617](#), [619](#), [620](#),
[622](#), [624](#), [628–634](#), [636–652](#), [654–661](#),
[664](#), [675](#), [676](#), [803](#), [824](#), [831](#)
TYPE(MPI_File), [611](#), [824](#), [832](#)
MPI_Fint, [803](#), [829](#), [831](#), [1016](#)
MPI_Group, [296](#), [297–302](#), [304](#), [336](#), [478](#), [536](#),
[565](#), [566](#), [617](#), [803](#), [824](#), [831](#)
TYPE(MPI_Group), [296](#), [824](#), [832](#)
MPI_Info, [429](#), [453](#), [454–457](#), [477](#), [480](#), [495](#),
[498](#), [499](#), [506](#), [510–512](#), [519](#), [537](#), [538](#),
[611](#), [614](#), [619](#), [620](#), [622](#), [747](#), [804](#), [823](#),
[824](#), [831](#), [1019](#)
TYPE(MPI_Info), [453](#), [823](#), [824](#), [832](#)
MPI_Message, [83](#), [804](#), [819](#), [824](#), [831](#), [1010](#)
TYPE(MPI_Message), [83](#), [819](#), [824](#), [832](#)
MPI_Offset, [20](#), [20](#), [33](#), [615](#), [616](#), [622](#), [624](#),
[628–634](#), [643–645](#), [651](#), [652](#), [654](#), [655](#),
[670](#), [678](#), [722](#), [800](#), [802](#), [821](#), [831](#)
MPI_Op, [212](#), [221](#), [225](#), [226](#), [228–231](#), [233–235](#),
[253](#), [254](#), [256–259](#), [276](#), [277](#), [279–282](#),
[545](#), [548](#), [550](#), [555](#), [557](#), [803](#), [824](#), [831](#)
TYPE(MPI_Op), [221](#), [824](#), [832](#)
MPI_Request, [60–64](#), [66](#), [68](#), [69](#), [70](#), [72–77](#), [79](#),
[88–95](#), [102–106](#), [443](#), [599](#), [603](#),
[631–634](#), [640–643](#), [647](#), [648](#), [783](#), [803](#),
[824](#), [831](#)
TYPE(MPI_Request), [68](#), [824](#), [832](#)
MPI_Session, [440](#), [441](#), [473](#), [804](#), [824](#), [831](#), [1005](#)
TYPE(MPI_Session), [473](#), [824](#), [832](#), [1005](#)
MPI_Status, [34](#), [37](#), [39–41](#), [68](#), [69](#), [72–77](#),
[79–81](#), [83](#), [85](#), [89](#), [147](#), [148](#), [600](#), [606](#),
[607](#), [628–630](#), [636–639](#), [646](#), [649](#), [650](#),
[655–659](#), [661](#), [757](#), [759](#), [805–807](#), [828](#),
[831](#), [1006](#), [1010](#), [1012](#)
TYPE(MPI_Status), [34](#), [37](#), [768](#), [769](#), [806](#), [807](#),
[828](#), [832](#), [1002](#), [1006](#), [1012](#)
MPI_T_cb_safety, [723](#), [729](#), [730](#), [831](#), [1005](#)
MPI_T_cvar_handle, [705](#), [706](#), [707](#), [829](#), [831](#)
MPI_T_enum, [700](#), [701](#), [702](#), [711](#), [725](#), [829](#), [831](#)
MPI_T_event_instance, [730](#), [733–735](#), [831](#), [1005](#)
MPI_T_event_registration, [727](#), [728–730](#), [732](#),
[831](#), [1005](#)
MPI_T_pvar_handle, [714](#), [715–717](#), [829–831](#)
MPI_T_pvar_session, [713](#), [714](#), [715–717](#), [829](#),
[831](#)
MPI_T_source_order, [721](#), [831](#), [1005](#)
MPI_Win, [351–353](#), [437](#), [438](#), [522](#), [525](#), [527](#),
[531](#), [535–539](#), [542](#), [545](#), [548](#), [550](#), [551](#),
[553–555](#), [557](#), [563](#), [565–568](#), [570](#), [571](#),
[573](#), [574](#), [803](#), [824](#), [831](#)
TYPE(MPI_Win), [522](#), [525](#), [527](#), [531](#), [824](#), [832](#)

MPI Callback Function Prototype Index

This index lists the C typedef names for callback routines, such as those used with attribute caching or user-defined reduction operations. Fortran example prototypes are given near the text of the C name.

COMM_COPY_ATTR_FUNCTION, [22](#), [23](#), [345](#),
[346](#), [758](#), [825](#), [837](#)
COMM_DELETE_ATTR_FUNCTION, [22](#), [345](#),
[346](#), [825](#), [837](#)
COMM_ERRHANDLER_FUNCTION, [435](#), [837](#)
COPY_FUNCTION, [22](#), [743](#), [744](#), [826](#), [839](#)

DATAREP_CONVERSION_FUNCTION, [669](#),
[671](#), [825](#), [838](#)
DATAREP_EXTENT_FUNCTION, [669](#), [670](#), [838](#)
DELETE_FUNCTION, [22](#), [743](#), [744](#), [826](#), [839](#)

FILE_ERRHANDLER_FUNCTION, [438](#), [439](#),
[838](#)

GREQUEST_CANCEL_FUNCTION, [600](#), [602](#),
[838](#)
GREQUEST_FREE_FUNCTION, [600](#), [601](#), [838](#)
GREQUEST_QUERY_FUNCTION, [600](#), [601](#),
[838](#)

MPI_Comm_copy_attr_function, [22](#), [23](#), [345](#),
[346](#), [758](#), [825](#), [832](#), [834](#)
MPI_Comm_delete_attr_function, [22](#), [345](#), [346](#),
[825](#), [832](#), [834](#)
MPI_Comm_errhandler_fn, [746](#), [1016](#)
MPI_Comm_errhandler_function, [22](#), [434](#), [435](#),
[746](#), [752](#), [833](#), [835](#), [1016](#)
MPI_COMM_NULL_COPY_FN, [825](#)
MPI_Copy_function, [22](#), [743](#), [826](#), [838](#)
MPI_Datarep_conversion_function, [666](#), [670](#),
[800](#), [825](#), [833](#), [836](#)
MPI_Datarep_conversion_function_c, [666](#),
[670](#), [671](#), [800](#), [825](#), [833](#), [836](#), [1003](#)
MPI_Datarep_extent_function, [666](#), [669](#), [670](#),
[833](#), [836](#)
MPI_Delete_function, [22](#), [743](#), [744](#), [826](#), [838](#)
MPI_File_errhandler_fn, [746](#), [1016](#)
MPI_File_errhandler_function, [438](#), [439](#), [746](#),
[833](#), [835](#), [1016](#)
MPI_Grequest_cancel_function, [600](#), [602](#), [833](#),
[836](#)
MPI_Grequest_free_function, [600](#), [601](#), [833](#), [836](#)
MPI_Grequest_query_function, [600](#), [600](#), [601](#),
[833](#), [835](#)

MPI_Handler_function, [22](#), [752](#), [1009](#)
MPI_Session_errhandler_function, [440](#), [833](#), [835](#),
[1005](#)
MPI_T_event_cb_function, [729](#), [730](#), [833](#), [1005](#)
MPI_T_event_dropped_cb_function, [732](#), [833](#),
[1005](#)
MPI_T_event_free_cb_function, [731](#), [833](#), [1005](#)
MPI_Type_copy_attr_function, [354](#), [355](#), [825](#),
[832](#), [835](#)
MPI_Type_delete_attr_function, [354](#), [355](#), [825](#),
[833](#), [835](#), [1013](#)
MPI_User_function, [221](#), [222](#), [226](#), [800](#), [832](#), [834](#)
MPI_User_function_c, [221](#), [222](#), [223](#), [800](#), [832](#),
[834](#), [1003](#)
MPI_Win_copy_attr_function, [351](#), [351](#), [352](#),
[825](#), [832](#), [834](#)
MPI_Win_delete_attr_function, [351](#), [351](#), [352](#),
[825](#), [832](#), [834](#)
MPI_Win_errhandler_fn, [746](#), [1016](#)
MPI_Win_errhandler_function, [436](#), [437](#), [746](#),
[833](#), [835](#), [1016](#)

SESSION_ERRHANDLER_FUNCTION, [440](#),
[441](#), [838](#), [1005](#)

TYPE_COPY_ATTR_FUNCTION, [354](#), [355](#),
[825](#), [837](#)
TYPE_DELETE_ATTR_FUNCTION, [354](#), [356](#),
[825](#), [837](#)

USER_FUNCTION, [222](#), [223](#), [836](#)

WIN_COPY_ATTR_FUNCTION, [351](#), [352](#), [825](#),
[837](#)
WIN_DELETE_ATTR_FUNCTION, [351](#), [352](#),
[825](#), [837](#)
WIN_ERRHANDLER_FUNCTION, [437](#), [837](#)

MPI Function Index

The underlined page numbers refer to the function definitions.

- MPI_ABORT, [223](#), [433](#), [463](#), [468](#), [485](#), [487](#), [518](#),
[699](#), [803](#), [1019](#)
- MPI_ACCUMULATE, [521](#), [539](#), [545](#), [547](#), [550](#),
[557](#), [560](#), [586](#), [592](#), [593](#), [1016](#), [1019](#),
[1020](#)
 - C example usage, [595](#)
 - Fortran example usage, [547](#)
 - Language-independent example usage, [592](#)
- MPI_Accumulate_c, [546](#), [1003](#)
- MPI_ADD_ERROR_CLASS, [446](#), [447](#)
- MPI_ADD_ERROR_CODE, [447](#)
- MPI_ADD_ERROR_STRING, [448](#)
- MPI_ADDRESS, [22](#), [751](#), [767](#), [1009](#)
- MPI_AINT_ADD, [24](#), [134](#), [136](#), [532](#), [1008](#)
 - C example usage, [595](#)
- MPI_AINT_DIFF, [24](#), [134](#), [136](#), [137](#), [532](#), [1008](#)
- MPI_ALLGATHER, [177](#), [181](#), [182](#), [202](#), [203](#),
[205](#), [207](#), [247](#)
 - C example usage, [205](#)
- MPI_Allgather_c, [202](#), [1003](#)
- MPI_ALLGATHER_INIT, [177](#), [181](#), [182](#), [269](#),
[1003](#)
 - MPI_Allgather_init_c, [269](#), [1003](#)
- MPI_ALLGATHERV, [177](#), [181](#), [182](#), [204](#), [205](#),
[249](#)
 - MPI_Allgatherv_c, [204](#), [1003](#)
- MPI_ALLGATHERV_INIT, [177](#), [181](#), [182](#), [270](#),
[1003](#)
 - MPI_Allgatherv_init_c, [271](#), [1003](#)
- MPI_ALLOC_MEM, [429](#), [430–432](#), [444](#),
[525–529](#), [531](#), [533](#), [541](#), [572](#), [768–770](#),
[781](#), [1004](#), [1006](#), [1013](#)
 - C example usage, [432](#), [595](#)
 - Fortran example usage, [431](#), [432](#)
- MPI_ALLOC_MEM_CPTR, [430](#), [1006](#)
- MPI_ALLREDUCE, [177](#), [180–182](#), [214](#), [222](#),
[226](#), [227](#), [255](#), [1017](#)
 - Fortran example usage, [228](#)
- MPI_Allreduce_c, [227](#), [1003](#)
- MPI_ALLREDUCE_INIT, [177](#), [181](#), [182](#), [277](#),
[1003](#)
 - MPI_Allreduce_init_c, [278](#), [1003](#)
- MPI_ALLTOALL, [177](#), [181](#), [182](#), [206](#), [207](#), [209](#),
[250](#), [1015](#)
 - C example usage, [288](#)
- MPI_Alltoall_c, [206](#), [1003](#)
- MPI_ALLTOALL_INIT, [177](#), [181](#), [182](#), [272](#), [1003](#)
 - MPI_Alltoall_init_c, [272](#), [1003](#)
- MPI_ALLTOALLV, [177](#), [181](#), [182](#), [207](#), [209](#), [212](#),
[251](#), [1015](#)
 - MPI_Alltoallv_c, [208](#), [1003](#)
- MPI_ALLTOALLV_INIT, [177](#), [181](#), [182](#), [273](#),
[1003](#)
 - MPI_Alltoallv_init_c, [274](#), [1003](#)
- MPI_ALLTOALLW, [177](#), [181](#), [182](#), [210](#), [211](#),
[212](#), [253](#), [1015](#)
 - MPI_Alltoallw_c, [210](#), [1003](#)
- MPI_ALLTOALLW_INIT, [177](#), [181](#), [182](#), [274](#),
[1003](#)
 - MPI_Alltoallw_init_c, [275](#), [1003](#)
- MPI_ATTR_DELETE, [22](#), [358](#), [744](#), [746](#)
- MPI_ATTR_GET, [22](#), [358](#), [745](#), [810](#), [811](#)
 - Fortran example usage, [812](#)
- MPI_ATTR_PUT, [22](#), [358](#), [745](#), [810](#), [811](#), [813](#)
- MPI_BARRIER, [177](#), [181](#), [182](#), [183](#), [184](#), [239](#),
[485](#), [583](#), [584](#), [680](#)
 - C example usage, [470](#), [591](#)
 - Fortran example usage, [484](#)
 - Language-independent example usage,
[582–585](#), [592](#), [593](#)
- MPI_BARRIER_INIT, [177](#), [181](#), [182](#), [262](#), [1003](#)
- MPI_BCAST, [14](#), [177](#), [181](#), [182](#), [184](#), [185](#), [214](#),
[240](#), [287](#)
 - C example usage, [185](#), [284–287](#), [328](#)
- MPI_Bcast_c, [184](#), [1003](#)
- MPI_BCAST_INIT, [14](#), [177](#), [181](#), [182](#), [262](#), [1003](#)
 - MPI_Bcast_init_c, [262](#), [1003](#)
- MPI_BSEND, [14](#), [48](#), [57](#)
 - Fortran example usage, [52](#)
- MPI_Bsend_c, [48](#), [1003](#)
- MPI_BSEND_INIT, [91](#), [95](#)
 - MPI_Bsend_init_c, [91](#), [1003](#)
- MPI_BUFFER_ATTACH, [55](#), [69](#)
 - C example usage, [56](#), [469](#)
- MPI_Buffer_attach_c, [55](#), [1003](#)
- MPI_BUFFER_DETACH, [55](#), [1003](#), [1012](#)
 - C example usage, [56](#)
- MPI_Buffer_detach_c, [55](#), [1003](#)
- MPI_CANCEL, [22](#), [51](#), [69](#), [80](#), [88](#), [89](#), [90](#), [238](#),
[261](#), [470](#), [552](#), [599](#), [602](#), [603](#), [747](#), [1003](#)

- C example usage, 470
- MPI_CART_COORDS, 369, [384](#), 385, 1001, 1018
- Fortran example usage, 390
- MPI_CART_CREATE, 334, 369, [370](#), [371–373](#), 393, 394, 782, 1006, 1018
- MPI_CART_GET, 369, [383](#), 1001, 1018
- Fortran example usage, 421
- MPI_CART_MAP, 369, [392](#), 393, 1011
- MPI_CART_RANK, 369, [384](#), 1018
- Fortran example usage, 390
- MPI_CART_SHIFT, 369, [389](#), 390, 394, 1018
- Fortran example usage, 390, 421
- Language-independent example usage, 401
- MPI_CART_SUB, 369, [391](#), 393, 1018
- Language-independent example usage, 391
- MPI_CARTDIM_GET, 369, [382](#), 383, 1018
- Language-independent example usage, 401
- MPI_CLOSE_PORT, [507](#), 511
- MPI_COMM_ACCEPT, [506](#), [508](#), 509, 516, 517
- C example usage, 513
- MPI_COMM_C2F, [803](#)
- MPI_COMM_CALL_ERRHANDLER, [448](#), 450
- MPI_COMM_COMPARE, [307](#), 336
- MPI_COMM_CONNECT, [445](#), [509](#), 516, 517
- C example usage, 513
- MPI_COMM_CREATE, 304, 307, [311](#), [312–316](#), 369, 1015
- C example usage, 312, 328, 329, 331
- MPI_COMM_CREATE_ERRHANDLER, 22, [434](#), 436, 751, 835, 837, 1013
- MPI_COMM_CREATE_FROM_GROUP, 305, 307, [323](#), 324, 339, 427, 1005
- C example usage, 480
- Fortran example usage, 484
- MPI_COMM_CREATE_GROUP, 305, 307, [313](#), [314–316](#), 323, 1011
- MPI_COMM_CREATE_KEYVAL, 22, 344, [345](#), 347, 357, 743, 809, 810, 834, 836, 1013, 1017
- C example usage, 358
- MPI_COMM_DELETE_ATTR, 22, 344, 347–349, 350, 358, 746
- MPI_COMM_DISCONNECT, 358, 475, 476, 485, 499, 517, [518](#), 519
- MPI_COMM_DUP, 298, 305, 307, [308](#), 309, 310, 312, 337, 340, 344, 346, 347, 350, 358, 365, 743, 753, 1003, 1011
- C example usage, 330
- MPI_COMM_DUP_FN, 22, [347](#), 348, 762, 825, 1007, 1013
- MPI_COMM_DUP_WITH_INFO, 305, 307, [309](#), 310, 324, 344, 346, 350, 358, 753, 1011
- MPI_COMM_F2C, [803](#)
- MPI_COMM_FREE, 305, 308, [324](#), 337, 340, 347, 348, 350, 358, 469, 472, 475, 485, 499, 517–519, 744
- MPI_COMM_FREE_KEYVAL, 22, 344, [348](#), 358, 744
- MPI_COMM_GET_ATTR, 22, 344, [349](#), 358, 427, 745, 764, 810, 811, 813
- C example usage, 358, 811
- Fortran example usage, 812
- MPI_COMM_GET_ERRHANDLER, 22, 434, 436, 751, 1019
- MPI_COMM_GET_INFO, 324, [326](#), 1004, 1011
- MPI_COMM_GET_NAME, [361](#), 362, 1001, 1017
- MPI_COMM_GET_PARENT, 361, 463, 469, 473, 496, [499](#)
- C example usage, 503
- MPI_COMM_GROUP, 17, 295, [298](#), 299, 304–306, 336, 434, 1019
- C example usage, 312, 331, 358
- MPI_COMM_IDUP, 305, 307, [309](#), 310, 334, 344, 346, 347, 350, 358, 753, 1003, 1006, 1011
- MPI_COMM_IDUP_WITH_INFO, 305, 307, [310](#), 324, 344, 346, 350, 358, 753, 1004
- MPI_COMM_JOIN, 473, [519](#), 520
- MPI_COMM_NULL_COPY_FN, 22, 23, [347](#), 348, 758, 809, 825, 1007, 1013
- MPI_COMM_NULL_DELETE_FN, 22, [347](#), 348, 825, 1007
- MPI_COMM_RANK, 14, [306](#), 336, 765
- C example usage, 29
- MPI_COMM_RANK_F08, 765
- MPI_COMM_REMOTE_GROUP, [337](#)
- MPI_COMM_REMOTE_SIZE, [336](#), 337
- C example usage, 316
- MPI_COMM_SET_ATTR, 22, 344, 347, [348](#), 358, 745, 764, 810, 811, 813
- C example usage, 358, 811
- Fortran example usage, 812
- MPI_COMM_SET_ERRHANDLER, 22, 434, 435, 751
- MPI_COMM_SET_INFO, 261, 324, [325](#), 326, 1004, 1006, 1011
- MPI_COMM_SET_NAME, 360
- MPI_COMM_SIZE, 305, 306, 336, 592
- MPI_COMM_SPAWN, 463, 464, 488, 493, 494, 495, 496–503, 515–517, 1005
- C example usage, 497, 503
- Fortran example usage, 497
- MPI_COMM_SPAWN_MULTIPLE, 463, 464, 488, 493, 494, [499](#), 500, 501, 516, 517, 1005
- C example usage, 502
- Fortran example usage, 502

- 1 MPI_COMM_SPLIT, [305](#), [307](#), [311](#), [312](#), [315](#),
2 [316](#), [317](#), [365](#), [369](#), [371](#), [373](#), [391](#), [393](#),
3 [394](#), [1015](#)
4 C example usage, [316](#), [341](#), [342](#)
- 5 MPI_COMM_SPLIT_TYPE, [305](#), [307](#), [318](#), [322](#),
6 [324](#), [1003](#), [1011](#)
7 C example usage, [320](#), [321](#)
- 8 MPI_COMM_TEST_INTER, [334](#), [335](#)
- 9 MPI_COMPARE_AND_SWAP, [521](#), [539](#), [551](#),
10 [592](#)
11 C example usage, [595](#)
12 Language-independent example usage, [593](#)
- 13 MPI_CONVERSION_FN_NULL, [672](#), [825](#), [1007](#)
- 14 MPI_CONVERSION_FN_NULL_C, [672](#), [800](#),
15 [825](#), [1003](#)
- 16 MPI_DIMS_CREATE, [369](#), [371](#), [372](#), [1004](#)
17 Fortran example usage, [421](#)
18 Language-independent example usage, [372](#)
- 19 MPI_DIST_GRAPH_CREATE, [324](#), [369](#), [374](#),
20 [376](#), [377–380](#), [389](#), [394](#), [1015](#)
21 C example usage, [380](#)
22 Language-independent example usage, [379](#)
- 23 MPI_DIST_GRAPH_CREATE_ADJACENT, [324](#),
24 [369](#), [374](#), [375](#), [376](#), [379](#), [380](#), [389](#), [394](#),
25 [1011](#), [1015](#)
26 Language-independent example usage, [379](#)
- 27 MPI_DIST_GRAPH_NEIGHBORS, [369](#), [387](#),
28 [388](#), [389](#), [394](#), [1011](#), [1015](#)
- 29 MPI_DIST_GRAPH_NEIGHBORS_COUNT,
30 [369](#), [387](#), [389](#), [1009](#), [1015](#)
- 31 MPI_DUP_FN, [22](#), [347](#), [744](#), [826](#)
- 32 MPI_ERRHANDLER_C2F, [486](#), [804](#)
- 33 MPI_ERRHANDLER_CREATE, [22](#), [751](#), [1009](#),
34 [1013](#)
- 35 MPI_ERRHANDLER_F2C, [486](#), [804](#)
- 36 MPI_ERRHANDLER_FREE, [434](#), [442](#), [469](#), [475](#),
37 [486](#), [1019](#)
- 38 MPI_ERRHANDLER_GET, [22](#), [751](#), [1009](#), [1019](#)
- 39 MPI_ERRHANDLER_SET, [22](#), [751](#), [1009](#)
- 40 MPI_ERROR_CLASS, [443](#), [446](#), [486](#)
- 41 MPI_ERROR_STRING, [442](#), [443](#), [446](#), [448](#), [486](#)
- 42 MPI_EXSCAN, [178](#), [181](#), [214](#), [222](#), [234](#), [235](#),
43 [260](#), [1015](#)
44 MPI_Exscan_c, [234](#), [1003](#)
- 45 MPI_EXSCAN_INIT, [178](#), [181](#), [282](#), [1003](#)
46 MPI_Exscan_init_c, [283](#), [1003](#)
- 47 MPI_F_SYNC_REG, [135](#), [593](#), [756](#), [771](#), [772](#),
48 [773](#), [791–795](#), [797](#), [1013](#)
Language-independent example usage, [593](#)
- MPI_FETCH_AND_OP, [521](#), [539](#), [547](#), [550](#)
- MPI_FILE_C2F, [803](#)
- MPI_FILE_CALL_ERRHANDLER, [449](#), [450](#)
- MPI_FILE_CLOSE, [519](#), [611](#), [612](#), [613](#), [614](#)
Fortran example usage, [637](#), [640](#)
- MPI_FILE_CREATE_ERRHANDLER, [434](#), [438](#),
[439](#), [835](#), [838](#), [1013](#)
- MPI_FILE_DELETE, [613](#), [614](#), [619](#), [621](#), [684](#)
- MPI_FILE_F2C, [803](#)
- MPI_FILE_GET_AMODE, [617](#), [618](#)
Fortran example usage, [618](#)
- MPI_FILE_GET_ATOMICITY, [675](#), [676](#)
- MPI_FILE_GET_BYTE_OFFSET, [636](#), [645](#), [652](#)
- MPI_FILE_GET_ERRHANDLER, [434](#), [439](#), [684](#),
[1019](#)
- MPI_FILE_GET_GROUP, [617](#)
- MPI_FILE_GET_INFO, [619](#), [620](#), [621](#), [1004](#),
[1020](#)
- MPI_FILE_GET_POSITION, [644](#)
- MPI_FILE_GET_POSITION_SHARED, [651](#),
[652](#), [676](#)
- MPI_FILE_GET_SIZE, [616](#), [617](#), [679](#)
- MPI_FILE_GET_TYPE_EXTENT, [663](#), [664](#),
[672](#), [1003](#)
MPI_File_get_type_extent_c, [664](#), [1003](#)
- MPI_FILE_GET_VIEW, [624](#), [625](#)
- MPI_FILE_IREAD, [625](#), [640](#), [653](#), [674](#)
Fortran example usage, [640](#)
MPI_File_iread_c, [640](#), [1003](#)
- MPI_FILE_IREAD_ALL, [625](#), [641](#), [642](#), [1008](#)
MPI_File_iread_all_c, [641](#), [1003](#)
- MPI_FILE_IREAD_AT, [625](#), [632](#)
MPI_File_iread_at_c, [632](#), [1003](#)
- MPI_FILE_IREAD_AT_ALL, [625](#), [633](#), [1008](#)
MPI_File_iread_at_all_c, [633](#), [1003](#)
- MPI_FILE_IREAD_SHARED, [625](#), [647](#), [648](#)
MPI_File_iread_shared_c, [648](#), [1003](#)
- MPI_FILE_IWRITE, [625](#), [642](#), [643](#)
MPI_File_iwrite_c, [642](#), [1003](#)
- MPI_FILE_IWRITE_ALL, [625](#), [643](#), [1008](#)
MPI_File_iwrite_all_c, [643](#), [1003](#)
- MPI_FILE_IWRITE_AT, [625](#), [634](#)
MPI_File_iwrite_at_c, [634](#), [1003](#)
- MPI_FILE_IWRITE_AT_ALL, [625](#), [635](#), [1008](#)
MPI_File_iwrite_at_all_c, [635](#), [1003](#)
- MPI_FILE_IWRITE_SHARED, [625](#), [648](#), [649](#)
MPI_File_iwrite_shared_c, [648](#), [1003](#)
- MPI_FILE_OPEN, [444](#), [492](#), [611](#), [612](#), [613](#), [619](#),
[621](#), [623](#), [645](#), [678](#), [679](#), [684](#), [685](#)
Fortran example usage, [637](#), [640](#)
- MPI_FILE_PREALLOCATE, [615](#), [616](#), [674](#), [679](#)
- MPI_FILE_READ, [625](#), [636](#), [637](#), [638](#), [640](#), [678](#),
[679](#)
Fortran example usage, [637](#)
MPI_File_read_c, [636](#), [1003](#)
- MPI_FILE_READ_ALL, [625](#), [637](#), [638](#), [642](#), [653](#),
[654](#)
MPI_File_read_all_c, [637](#), [1003](#)

MPI_FILE_READ_ALL_BEGIN, 14, 625, 653, 654, 657, 674, 797		
MPI_File_read_all_begin_c, 657, 1003		
MPI_FILE_READ_ALL_END, 625, 653, 654, 657, 674, 797		
MPI_FILE_READ_AT, 625, 628, 629, 630, 632		
MPI_File_read_at_c, 628, 1003		
MPI_FILE_READ_AT_ALL, 625, 629, 630, 633		
MPI_File_read_at_all_c, 629, 1003		
MPI_FILE_READ_AT_ALL_BEGIN, 14, 625, 654, 797		
MPI_File_read_at_all_begin_c, 654, 1003		
MPI_FILE_READ_AT_ALL_END, 625, 655, 797		
MPI_FILE_READ_ORDERED, 625, 650		
MPI_File_read_ordered_c, 650, 1003		
MPI_FILE_READ_ORDERED_BEGIN, 14, 625, 659, 797		
MPI_File_read_ordered_begin_c, 659, 1003		
MPI_FILE_READ_ORDERED_END, 625, 660, 797		
MPI_FILE_READ_SHARED, 625, 646, 648, 650		
MPI_File_read_shared_c, 646, 1003		
MPI_FILE_SEEK, 643, 644		
MPI_FILE_SEEK_SHARED, 651, 652, 676		
MPI_FILE_SET_ATOMICITY, 613, 674, 675		
C example usage, 680		
MPI_FILE_SET_ERRHANDLER, 434, 439, 684		
MPI_FILE_SET_INFO, 619, 621, 1004, 1020		
MPI_FILE_SET_SIZE, 615, 616, 674, 677, 679		
MPI_FILE_SET_VIEW, 129, 445, 612, 619, 621, 622, 623, 624, 645, 652, 662, 669, 678, 685, 1020		
Fortran example usage, 637, 640		
MPI_FILE_SYNC, 614, 626, 673–675, 676, 681, 682		
C example usage, 680		
MPI_FILE_WRITE, 625, 626, 638, 639, 640, 643, 678		
MPI_File_write_c, 638, 1003		
MPI_FILE_WRITE_ALL, 625, 639, 640, 643		
MPI_File_write_all_c, 639, 1003		
MPI_FILE_WRITE_ALL_BEGIN, 14, 625, 658, 783, 797		
MPI_File_write_all_begin_c, 658, 1003		
MPI_FILE_WRITE_ALL_END, 625, 658, 797		
MPI_FILE_WRITE_AT, 625, 626, 630, 631, 634		
MPI_File_write_at_c, 630, 1003		
MPI_FILE_WRITE_AT_ALL, 625, 631, 635		
MPI_File_write_at_all_c, 631, 1003		
MPI_FILE_WRITE_AT_ALL_BEGIN, 14, 625, 655, 797		
MPI_File_write_at_all_begin_c, 656, 1003		
MPI_FILE_WRITE_AT_ALL_END, 625, 656, 797		
MPI_FILE_WRITE_ORDERED, 625, 649, 650, 651		1
MPI_File_write_ordered_c, 651, 1003		2
MPI_FILE_WRITE_ORDERED_BEGIN, 14, 625, 660, 797		3
MPI_File_write_ordered_begin_c, 660, 1003		4
MPI_FILE_WRITE_ORDERED_END, 625, 661, 797		5
MPI_FILE_WRITE_SHARED, 625, 626, 647, 649, 651		6
MPI_File_write_shared_c, 647, 1003		7
MPI_FINALIZE, 18, 25, 26, 427, 432, 461, 468, 469–472, 476, 491, 518, 612, 695, 705, 718, 720, 802, 805, 806, 1005, 1011, 1019		8
C example usage, 469–471		9
MPI_FINALIZED, 471, 472, 486, 802, 1008		10
MPI_FREE_MEM, 430, 431, 444, 526, 528		11
C example usage, 595		12
Fortran example usage, 431, 432		13
MPI_GATHER, 177, 180–182, 186, 189, 190, 197, 203, 214, 242		14
C example usage, 172, 190, 194		15
MPI_Gather_c, 186, 1003		16
MPI_GATHER_INIT, 177, 181, 182, 263, 1003		17
MPI_Gather_init_c, 264, 1003		18
MPI_GATHERV, 177, 181, 182, 187, 189–191, 199, 205, 244		19
C example usage, 172, 191–194		20
MPI_Gatherv_c, 188, 1003		21
MPI_GATHERV_INIT, 177, 181, 182, 264, 1003		22
MPI_Gatherv_init_c, 265, 1003		23
MPI_GET, 521, 539, 542, 543, 550, 555, 560, 582, 585, 593, 794, 1019		24
C example usage, 589–591		25
Fortran example usage, 543, 545		26
Language-independent example usage, 582, 584		27
MPI_Get_c, 542, 1003		28
MPI_GET_ACCUMULATE, 521, 539, 547, 548, 550, 558, 586, 591, 592, 1007		29
C example usage, 595		30
Language-independent example usage, 592		31
MPI_Get_accumulate_c, 548, 1003		32
MPI_GET_ADDRESS, 22, 115, 134, 135, 136, 137, 150, 532, 751, 767, 785, 790, 808, 809		33
C example usage, 161, 163, 164, 171		34
Fortran example usage, 135, 786, 808, 814		35
MPI_GET_COUNT, 37, 38, 39, 68, 148, 149, 552, 607, 627, 1010, 1017		36
Fortran example usage, 149		37
MPI_Get_count_c, 38, 1003		38
MPI_GET_COUNT_C, 607		39

- 1 MPI_GET_ELEMENTS, [68](#), [147](#), [148](#), [149](#), [607](#),
2 [608](#), [627](#), [1010](#)
3 Fortran example usage, [149](#)
4 MPI_Get_elements_c, [147](#), [1003](#)
5 MPI_GET_ELEMENTS_C, [607](#)
6 MPI_GET_ELEMENTS_X, [68](#), [147](#), [148](#), [149](#),
7 [607](#), [627](#), [1010](#)
8 MPI_GET_LIBRARY_VERSION, [426](#), [486](#), [1006](#),
9 [1008](#), [1010](#)
10 MPI_GET_PROCESSOR_NAME, [428](#), [429](#), [1019](#)
11 MPI_GET_VERSION, [425](#), [426](#), [486](#), [771](#), [1008](#)
12 MPI_GRAPH_CREATE, [369](#), [372](#), [373](#), [374](#), [379](#),
13 [382](#), [386](#), [393](#), [394](#), [1018](#)
14 Language-independent example usage,
15 [373](#), [386](#), [387](#)
16 MPI_GRAPH_GET, [369](#), [382](#)
17 MPI_GRAPH_MAP, [369](#), [393](#), [394](#)
18 MPI_GRAPH_NEIGHBORS, [369](#), [385](#), [386](#), [394](#),
19 [1016](#)
20 Language-independent example usage, [386](#)
21 MPI_GRAPH_NEIGHBORS_COUNT, [369](#), [385](#),
22 [386](#), [1016](#)
23 Language-independent example usage, [386](#)
24 MPI_GRAPHDIMS_GET, [369](#), [381](#), [382](#)
25 MPI_GREQUEST_COMPLETE, [599](#), [601](#), [602](#),
26 [603](#)
27 C example usage, [603](#)
28 MPI_GREQUEST_START, [600](#), [835](#), [838](#), [1016](#)
29 C example usage, [603](#)
30 MPI_GROUP_C2F, [803](#)
31 MPI_GROUP_COMPARE, [297](#), [301](#)
32 MPI_GROUP_DIFFERENCE, [300](#)
33 MPI_GROUP_EXCL, [301](#), [303](#)
34 C example usage, [328](#)
35 MPI_GROUP_F2C, [803](#)
36 MPI_GROUP_FREE, [304](#), [305](#), [306](#), [434](#), [469](#),
37 [475](#), [1019](#)
38 C example usage, [312](#), [328](#), [329](#), [331](#)
39 MPI_GROUP_FROM_SESSION_PSET, [303](#),
40 [304](#), [1005](#)
41 C example usage, [480](#), [482](#)
42 Fortran example usage, [484](#)
43 MPI_GROUP_INCL, [300](#), [301](#), [302](#)
44 C example usage, [312](#), [329](#), [331](#)
45 MPI_GROUP_INTERSECTION, [299](#)
46 MPI_GROUP_RANGE_EXCL, [303](#)
47 MPI_GROUP_RANGE_INCL, [302](#)
48 MPI_GROUP_RANK, [296](#), [306](#)
MPI_GROUP_SIZE, [296](#), [305](#)
MPI_GROUP_TRANSLATE_RANKS, [297](#), [1017](#)
MPI_GROUP_UNION, [299](#)
MPI_IALLGATHER, [177](#), [181](#), [182](#), [246](#)
MPI_iallgather_c, [247](#), [1003](#)
MPI_IALLGATHERV, [177](#), [181](#), [182](#), [248](#), [1003](#)
MPI_IALLREDUCE, [177](#), [181](#), [182](#), [254](#)
C example usage, [289](#)
MPI_iallreduce_c, [255](#), [1003](#)
MPI_IALLTOALL, [177](#), [181](#), [182](#), [249](#)
C example usage, [288](#)
MPI_ialltoall_c, [249](#), [1003](#)
MPI_IALLTOALLV, [177](#), [181](#), [182](#), [250](#)
MPI_ialltoallv_c, [251](#), [1003](#)
MPI_IALLTOALLW, [177](#), [181](#), [182](#), [251](#)
MPI_ialltoallw_c, [252](#), [1003](#)
MPI_IBARRIER, [177](#), [181](#), [182](#), [238](#), [239](#), [287](#)
C example usage, [286](#)–[288](#)
MPI_IBCAST, [14](#), [177](#), [181](#), [182](#), [240](#), [241](#), [290](#)
C example usage, [241](#), [289](#), [290](#)
MPI_ibcast_c, [240](#), [1003](#)
MPI_IBSEND, [61](#), [68](#), [95](#)
MPI_ibsend_c, [61](#), [1003](#)
MPI_IXSCAN, [178](#), [181](#), [259](#)
MPI_ixscan_c, [259](#), [1003](#)
MPI_IGATHER, [177](#), [181](#), [182](#), [241](#)
MPI_igather_c, [241](#), [1003](#)
MPI_IGATHERV, [177](#), [181](#), [182](#), [242](#)
MPI_igatherv_c, [243](#), [1003](#)
MPI_IMPROBE, [14](#), [80](#), [83](#), [84](#), [85](#), [88](#), [492](#),
[1006](#), [1010](#)
MPI_IMRECV, [83](#)–[85](#), [87](#), [88](#), [1010](#)
MPI_imrecv_c, [87](#), [1003](#)
MPI_INEIGHBOR_ALLGATHER, [370](#), [406](#), [1011](#)
MPI_ineighbor_allgather_c, [407](#), [1003](#)
MPI_INEIGHBOR_ALLGATHERV, [370](#), [407](#),
[1011](#)
MPI_ineighbor_allgatherv_c, [408](#), [1003](#)
MPI_INEIGHBOR_ALLTOALL, [370](#), [409](#), [1011](#)
MPI_ineighbor_alltoall_c, [409](#), [1003](#)
MPI_INEIGHBOR_ALLTOALLV, [370](#), [410](#), [1011](#)
MPI_ineighbor_alltoallv_c, [410](#), [1003](#)
MPI_INEIGHBOR_ALLTOALLW, [370](#), [411](#), [1011](#)
MPI_ineighbor_alltoallw_c, [412](#), [1003](#)
MPI_INFO_C2F, [486](#), [804](#)
MPI_INFO_CREATE, [454](#), [486](#)
C example usage, [480](#)
MPI_INFO_CREATE_ENV, [458](#), [486](#), [1005](#)
MPI_INFO_DELETE, [444](#), [455](#), [457](#), [486](#)
MPI_INFO_DUP, [457](#), [486](#)
MPI_INFO_ENV
Language-independent example usage, [464](#)
MPI_INFO_F2C, [486](#), [804](#)
MPI_INFO_FREE, [327](#), [458](#), [469](#), [475](#), [480](#), [486](#),
[538](#), [620](#), [722](#), [726](#), [729](#), [731](#)
MPI_INFO_GET, [22](#), [747](#), [748](#), [1004](#), [1019](#)
MPI_INFO_GET_NKEYS, [453](#), [456](#), [457](#), [486](#),
[1019](#)
MPI_INFO_GET_NTHKEY, [453](#), [457](#), [486](#), [1019](#)

- MPI_INFO_GET_STRING, [22](#), [453](#), [455](#), [456](#),
[486](#), [747](#), [748](#), [1004](#)
- MPI_INFO_GET_VALUELEN, [22](#), [748](#), [1004](#),
[1019](#)
- MPI_INFO_SET, [454](#), [455](#), [457](#), [486](#), [747](#)
C example usage, [480](#)
- MPI_INIT, [18](#), [25](#), [26](#), [295](#), [427](#), [432](#), [458](#), [461](#),
[462](#), [463](#), [465–468](#), [471](#), [472](#), [485](#), [491](#),
[493](#), [496](#), [497](#), [499](#), [515](#), [516](#), [695](#), [698](#),
[705](#), [718](#), [802](#), [805](#), [806](#), [1005](#), [1010](#),
[1011](#), [1014](#), [1016](#)
C example usage, [29](#), [463](#)
- MPI_INIT_THREAD, [25](#), [295](#), [432](#), [458](#), [461](#),
[462](#), [465](#), [466–468](#), [471](#), [472](#), [474](#), [491](#),
[493](#), [516](#), [695](#), [698](#), [705](#), [802](#), [1010](#),
[1011](#), [1016](#)
- MPI_INITIALIZED, [471](#), [486](#), [802](#), [1008](#)
- MPI_INTERCOMM_CREATE, [305](#), [308](#), [314](#),
[337](#), [338](#), [339](#), [340](#), [1011](#)
C example usage, [341](#), [342](#)
- MPI_INTERCOMM_CREATE_FROM_GROUPS,
[305](#), [307](#), [308](#), [337](#), [338](#), [339](#), [427](#), [1005](#)
- MPI_INTERCOMM_MERGE, [307](#), [314](#), [334](#),
[337](#), [338](#), [340](#), [1013](#)
- MPI_IPROBE, [14](#), [38](#), [80](#), [81–85](#), [88](#), [492](#), [1010](#)
C example usage, [470](#)
- MPI_IRECV, [14](#), [64](#), [87](#), [784](#), [785](#), [788](#), [789](#)
C example usage, [288](#)
Fortran example usage, [70–72](#), [78](#)
MPI_Irecv_c, [64](#), [1003](#)
- MPI_IREDUCE, [177](#), [181](#), [182](#), [253](#), [254](#)
MPI_Ireduce_c, [253](#), [1003](#)
- MPI_IREDUCE_SCATTER, [178](#), [181](#), [182](#), [257](#)
MPI_Ireduce_scatter_c, [257](#), [1003](#)
- MPI_IREDUCE_SCATTER_BLOCK, [177](#), [181](#),
[182](#), [256](#)
MPI_Ireduce_scatter_block_c, [256](#), [1003](#)
- MPI_IRSEND, [63](#)
MPI_Irsend_c, [63](#), [1003](#)
- MPI_IS_THREAD_MAIN, [465](#), [467](#), [1008](#)
- MPI_ISCAN, [178](#), [181](#), [258](#)
MPI_Iscan_c, [258](#), [1003](#)
- MPI_ISCATTER, [177](#), [181](#), [182](#), [244](#)
MPI_Iscatter_c, [244](#), [1003](#)
- MPI_ISCATTERV, [177](#), [181](#), [182](#), [245](#)
MPI_Iscatterv_c, [246](#), [1003](#)
- MPI_ISEND, [14](#), [60](#), [95](#), [763](#), [764](#), [767](#), [783](#),
[784](#), [789](#)
Fortran example usage, [70](#), [71](#), [78](#)
MPI_Isend_c, [60](#), [1003](#)
- MPI_ISENDRECV, [65](#), [1003](#)
MPI_Isendrecv_c, [65](#), [1003](#)
- MPI_ISENDRECV_REPLACE, [66](#), [1003](#)
MPI_Isendrecv_replace_c, [66](#), [1003](#)
- MPI_ISSEND, [62](#) 1
MPI_Issend_c, [62](#), [1003](#) 2
- MPI_KEYVAL_CREATE, [22](#), [743](#), [745](#), [839](#) 3
- MPI_KEYVAL_FREE, [22](#), [358](#), [744](#) 4
- MPI_LOOKUP_NAME, [444](#), [506](#), [511](#), [512](#) 5
- MPI_MESSAGE_C2F, [804](#), [1010](#) 6
- MPI_MESSAGE_F2C, [804](#), [1010](#) 7
- MPI_MPROBE, [14](#), [80](#), [83](#), [84](#), [85](#), [88](#), [492](#), [1010](#) 8
- MPI_MRECV, [14](#), [83–85](#), [86](#), [87](#), [1010](#) 9
MPI_Mrecv_c, [86](#), [1003](#)
- MPI_NEIGHBOR_ALLGATHER, [370](#), [395](#), [397](#),
[399](#), [407](#), [1002](#), [1011](#) 10
Language-independent example usage, [397](#) 12
MPI_Neighbor_allgather_c, [395](#), [1003](#) 13
- MPI_NEIGHBOR_ALLGATHER_INIT, [413](#), [1003](#) 14
MPI_Neighbor_allgather_init_c, [414](#), [1003](#) 15
- MPI_NEIGHBOR_ALLGATHERV, [370](#), [397](#), [409](#),
[1002](#), [1011](#) 16
MPI_Neighbor_allgatherv_c, [398](#), [1003](#) 17
- MPI_NEIGHBOR_ALLGATHERV_INIT, [414](#),
[763](#), [1003](#), [1006](#) 18
MPI_Neighbor_allgatherv_init_c, [415](#), [1003](#) 19
- MPI_NEIGHBOR_ALLTOALL, [370](#), [399](#), [401](#),
[402](#), [410](#), [1002](#), [1011](#) 20
Language-independent example usage, [401](#) 22
MPI_Neighbor_alltoall_c, [400](#), [1003](#) 23
- MPI_NEIGHBOR_ALLTOALL_INIT, [416](#), [1003](#) 24
MPI_Neighbor_alltoall_init_c, [416](#), [1003](#) 25
- MPI_NEIGHBOR_ALLTOALLV, [370](#), [402](#), [411](#),
[1002](#), [1011](#) 26
MPI_Neighbor_alltoallv_c, [403](#), [1003](#) 28
- MPI_NEIGHBOR_ALLTOALLV_INIT, [417](#), [763](#),
[1003](#), [1006](#) 29
MPI_Neighbor_alltoallv_init_c, [418](#), [1003](#) 30
- MPI_NEIGHBOR_ALLTOALLW, [370](#), [404](#), [413](#),
[1002](#), [1011](#) 31
MPI_Neighbor_alltoallw_c, [405](#), [1003](#) 33
- MPI_NEIGHBOR_ALLTOALLW_INIT, [419](#), [763](#),
[1003](#), [1006](#) 34
MPI_Neighbor_alltoallw_init_c, [419](#), [1003](#) 35
- MPI_NULL_COPY_FN, [22](#), [347](#), [744](#), [826](#) 36
- MPI_NULL_DELETE_FN, [22](#), [347](#), [744](#), [826](#) 37
- MPI_OP_C2F, [804](#) 38
- MPI_OP_COMMUTATIVE, [229](#), [1015](#) 39
- MPI_OP_CREATE, [221](#), [222](#), [224](#), [762](#), [833](#),
[836](#), [1013](#) 40
C example usage, [225](#), [236](#) 42
Fortran example usage, [226](#) 43
MPI_Op_create_c, [221](#), [222](#), [801](#), [833](#), [1003](#) 44
- MPI_OP_F2C, [803](#) 45
- MPI_OP_FREE, [225](#), [469](#), [475](#) 45
- MPI_OPEN_PORT, [506](#), [508–513](#) 46
C example usage, [513](#) 47
- MPI_PACK, [58](#), [167](#), [169–171](#), [173](#), [666](#), [671](#) 48

- 1 C example usage, [171](#), [172](#)
2 MPI_Pack_c, [167](#), [1003](#)
3 MPI_PACK_EXTERNAL, [8](#), [173](#), [174](#), [777](#), [1017](#)
4 MPI_Pack_external_c, [174](#), [1003](#)
5 MPI_PACK_EXTERNAL_SIZE, [176](#)
6 MPI_Pack_external_size_c, [176](#), [1003](#)
7 MPI_PACK_SIZE, [57](#), [170](#), [1010](#)
8 C example usage, [172](#)
9 MPI_Pack_size_c, [170](#), [1003](#)
10 MPI_PARRIVED, [100](#), [106](#), [107](#), [1003](#)
11 C example usage, [111](#)
12 MPI_PCONTROL, [690](#), [691](#)
13 MPI_PREADY, [100–102](#), [104](#), [105–107](#), [1003](#)
14 C example usage, [100](#), [108](#), [109](#), [111](#)
15 MPI_PREADY_LIST, [105](#), [106](#), [1003](#)
16 MPI_PREADY_RANGE, [105](#), [106](#), [1003](#)
17 MPI_PRECV_INIT, [100](#), [103](#), [104](#), [108](#), [1003](#)
18 C example usage, [100](#), [108](#), [109](#), [111](#)
19 MPI_PROBE, [14](#), [35](#), [38](#), [39](#), [80](#), [81](#), [82](#), [83](#), [85](#),
20 [87](#), [88](#), [492](#), [1010](#)
21 Fortran example usage, [82](#)
22 MPI_PSEND_INIT, [100](#), [102](#), [103–105](#), [108](#),
23 [1003](#)
24 C example usage, [100](#), [108](#), [109](#), [111](#)
25 MPI_PUBLISH_NAME, [506](#), [510](#), [511](#), [512](#)
26 C example usage, [513](#)
27 MPI_PUT, [521](#), [539](#), [540](#), [543](#), [547](#), [554](#), [560](#),
28 [566](#), [576](#), [579](#), [583](#), [585](#), [593](#), [783](#), [794](#),
29 [1019](#)
30 C example usage, [566](#), [572](#), [589](#), [590](#)
31 Language-independent example usage,
32 [583](#), [585](#), [589](#)
33 MPI_Put_c, [540](#), [1003](#)
34 MPI_QUERY_THREAD, [467](#), [468](#), [491](#), [1008](#)
35 MPI_RACCUMULATE, [521](#), [539](#), [547](#), [550](#), [555](#),
36 [557](#)
37 MPI_Raccumulate_c, [556](#), [1003](#)
38 MPI_RECV, [14](#), [30](#), [34](#), [36](#), [38](#), [39](#), [81](#), [83](#), [84](#),
39 [87](#), [114](#), [146](#), [147](#), [169](#), [178](#), [187](#), [288](#),
40 [608](#), [680](#), [718](#), [790](#), [794](#)
41 C example usage, [29](#), [288](#)
42 Fortran example usage, [43](#), [44](#), [52–54](#), [72](#),
43 [82](#), [147](#)
44 MPI_Recv_c, [35](#), [1003](#)
45 MPI_RECV_INIT, [14](#), [94](#), [95](#), [104](#)
46 MPI_Recv_init_c, [94](#), [1003](#)
47 MPI_REDUCE, [177](#), [181](#), [182](#), [212](#), [213](#), [214](#),
48 [222–224](#), [227](#), [231](#), [232](#), [234](#), [235](#), [254](#),
[547](#), [550](#), [551](#), [1016](#)
C example usage, [219](#), [220](#), [225](#), [330](#)
Fortran example usage, [216](#), [217](#), [220](#), [226](#)
MPI_Reduce_c, [212](#), [1003](#)
MPI_REDUCE_INIT, [177](#), [181](#), [276](#), [1003](#)
MPI_Reduce_init_c, [277](#), [1003](#)
MPI_REDUCE_LOCAL, [214](#), [222](#), [228](#), [1013](#),
[1015](#)
MPI_Reduce_local_c, [228](#), [1003](#)
MPI_REDUCE_SCATTER, [177](#), [181](#), [182](#), [214](#),
[222](#), [231](#), [232](#), [258](#)
MPI_Reduce_scatter_c, [232](#), [1003](#)
MPI_REDUCE_SCATTER_BLOCK, [177](#), [181](#),
[182](#), [214](#), [222](#), [230](#), [231](#), [257](#), [1015](#)
MPI_Reduce_scatter_block_c, [230](#), [1003](#)
MPI_REDUCE_SCATTER_BLOCK_INIT, [177](#),
[181](#), [182](#), [279](#), [1003](#)
MPI_Reduce_scatter_block_init_c, [279](#),
[1003](#)
MPI_REDUCE_SCATTER_INIT, [178](#), [181](#), [182](#),
[280](#), [1003](#)
MPI_Reduce_scatter_init_c, [280](#), [1003](#)
MPI_REGISTER_DATAREP, [444](#), [666](#), [669–672](#),
[685](#), [836](#), [838](#)
MPI_Register_datarep_c, [666](#), [672](#), [801](#),
[836](#), [1003](#)
MPI_REQUEST_C2F, [803](#)
MPI_REQUEST_F2C, [803](#)
MPI_REQUEST_FREE, [70](#), [71](#), [88](#), [96](#), [238](#), [260](#),
[261](#), [469](#), [475](#), [552](#), [601–603](#), [1014](#)
C example usage, [469](#)
Fortran example usage, [71](#)
MPI_REQUEST_GET_STATUS, [39](#), [79](#), [80](#), [601](#),
[1002](#), [1014](#)
MPI_RGET, [521](#), [539](#), [554](#), [555](#)
C example usage, [594](#)
MPI_Rget_c, [554](#), [1003](#)
MPI_RGET_ACCUMULATE, [521](#), [539](#), [547](#), [550](#),
[557](#), [558](#), [559](#)
MPI_Rget_accumulate_c, [557](#), [1003](#)
MPI_RPUT, [521](#), [539](#), [553](#), [554](#)
C example usage, [594](#)
MPI_Rput_c, [553](#), [1003](#)
MPI_RSEND, [14](#), [50](#)
MPI_Rsend_c, [50](#), [1003](#)
MPI_RSEND_INIT, [93](#)
MPI_Rsend_init_c, [93](#), [1003](#)
MPI_SCAN, [178](#), [181](#), [214](#), [222](#), [233](#), [234](#), [235](#),
[259](#)
C example usage, [236](#)
MPI_Scan_c, [233](#), [1003](#)
MPI_SCAN_INIT, [178](#), [181](#), [281](#), [1003](#)
MPI_Scan_init_c, [281](#), [1003](#)
MPI_SCATTER, [177](#), [181](#), [182](#), [196](#), [197](#), [199](#),
[200](#), [231](#), [245](#)
C example usage, [200](#)
MPI_Scatter_c, [196](#), [1003](#)
MPI_SCATTER_INIT, [177](#), [181](#), [182](#), [266](#), [1003](#)
MPI_Scatter_init_c, [266](#), [1003](#)

MPI_SCATTERV, 177 , 181 , 182 , 198 , 199 , 200 , 233 , 246	Fortran example usage, 52 , 72	1
C example usage, 200 , 201	MPI_Ssend_c, 49 , 1003	2
MPI_Scatterv_c, 198 , 1003	MPI_SSEND_INIT, 92	3
MPI_SCATTERV_INIT, 177 , 181 , 182 , 267 , 1003	MPI_Ssend_init_c, 92 , 1003	4
MPI_Scatterv_init_c, 268 , 1003	MPI_START, 95 , 96 , 99 , 102 , 105 , 106 , 261 , 790	5
MPI_SEND, 14 , 29 , 30 , 31 , 39 , 44 , 114 , 146 , 167 , 288 , 612 , 680 , 692 , 790 , 791 , 794	MPI_STARTALL, 95 , 96 , 99 , 102 , 105 , 106 , 261 , 790	6
C example usage, 29 , 161 , 163 , 164 , 171 , 288 , 692	MPI_STATUS_C2F, 805	7
Fortran example usage, 43 , 44 , 53 , 54 , 72 , 82 , 147	MPI_STATUS_C2F08, 806 , 1012	8
MPI_Send_c, 30 , 1003	MPI_STATUS_F082C, 806 , 1012	9
MPI_SEND_INIT, 14 , 90 , 95 , 103	MPI_STATUS_F082F, 807 , 1002 , 1012	10
MPI_Send_init_c, 90 , 1003	MPI_STATUS_F2C, 805	11
MPI_SENDRECV, 40 , 96 , 389	MPI_STATUS_F2F08, 806 , 1002 , 1012	12
Fortran example usage, 158 – 160	MPI_STATUS_SET_CANCELLED, 607 , 1005	13
Language-independent example usage, 401	MPI_STATUS_SET_ELEMENTS, 606 , 607	14
MPI_Sendrecv_c, 40 , 1003	MPI_Status_set_elements_c, 606	15
MPI_SENDRECV_REPLACE, 41 , 96	MPI_STATUS_SET_ELEMENTS_C, 607	16
Fortran example usage, 390	MPI_STATUS_SET_ELEMENTS_X, 606 , 607 , 1010	17
MPI_Sendrecv_replace_c, 42 , 1003	MPI_T_CATEGORY_CHANGED, 739 , 1005	18
MPI_SESSION_C2F, 804 , 1005	MPI_T_CATEGORY_GET_CATEGORIES, 739 , 740 , 741	19
MPI_SESSION_CALL_ERRHANDLER, 450 , 486 , 1005	MPI_T_CATEGORY_GET_CVARS, 738 , 740 , 741	20
MPI_SESSION_CREATE_ERRHANDLER, 434 , 440 , 441 , 475 , 486 , 835 , 838 , 1005	MPI_T_CATEGORY_GET_EVENTS, 739 , 740 , 1005	21
MPI_SESSION_F2C, 804 , 1005	MPI_T_CATEGORY_GET_INDEX, 737 , 738 , 741 , 1008	22
MPI_SESSION_FINALIZE, 475 , 476 , 485 , 802 , 1005	MPI_T_CATEGORY_GET_INFO, 736 , 737 , 740 , 741 , 1007 , 1008	23
C example usage, 480 , 482	MPI_T_CATEGORY_GET_NUM, 736	24
Fortran example usage, 484	MPI_T_CATEGORY_GET_NUM_EVENTS, 737 , 1005	25
Language-independent example usage, 476	MPI_T_CATEGORY_GET_PVARS, 738 , 739 – 741	26
MPI_SESSION_GET_ERRHANDLER, 434 , 441 , 1005	MPI_T_CVAR_GET_INDEX, 704 , 741 , 1008	27
MPI_SESSION_GET_INFO, 474 , 479 , 480 , 1005	MPI_T_CVAR_GET_INFO, 700 , 702 , 703 , 706 , 707 , 740 , 741 , 1007 , 1008	28
MPI_SESSION_GET_NTH_PSET, 304 , 478 , 479 , 1005	C example usage, 705	29
C example usage, 482	MPI_T_CVAR_GET_NUM, 702 , 706	30
Fortran example usage, 484	MPI_T_CVAR_HANDLE_ALLOC, 700 , 706 , 707 , 741	31
MPI_SESSION_GET_NUM_PSETS, 477 , 478 , 1005	C example usage, 708	32
C example usage, 482	MPI_T_CVAR_HANDLE_FREE, 706 , 741	33
Fortran example usage, 484	C example usage, 708	34
MPI_SESSION_GET_PSET_INFO, 480 , 1005	MPI_T_CVAR_READ, 707 , 741	35
MPI_SESSION_INIT, 434 , 469 , 473 , 475 , 485 , 491 , 516 , 699 , 802 , 1005	C example usage, 708	36
C example usage, 480 , 482	MPI_T_CVAR_WRITE, 707 , 741 , 1002	37
Fortran example usage, 484	MPI_T_ENUM_GET_INFO, 700 , 741	38
MPI_SESSION_SET_ERRHANDLER, 434 , 441 , 1005	MPI_T_ENUM_GET_ITEM, 701 , 741	39
MPI_SIZEOF, 22 , 749 , 1005	MPI_T_EVENT_CALLBACK_GET_INFO, 730 , 731 , 1005	40
MPI_SSEND, 49	MPI_T_EVENT_CALLBACK_SET_INFO, 730 , 1005	41

- 1 MPI_T_EVENT_COPY, [724](#), [726](#), [734](#), [1005](#)
2 MPI_T_EVENT_GET_INDEX, [727](#), [741](#), [1005](#)
3 MPI_T_EVENT_GET_INFO, [700](#), [725](#), [726](#), [728](#),
4 [734](#), [740](#), [741](#), [1005](#)
5 MPI_T_EVENT_GET_NUM, [725](#), [1005](#)
6 MPI_T_EVENT_GET_SOURCE, [724](#), [735](#), [1005](#)
7 MPI_T_EVENT_GET_TIMESTAMP, [722](#), [724](#),
8 [734](#), [735](#), [1005](#)
9 MPI_T_EVENT_HANDLE_ALLOC, [728](#), [730](#),
10 [741](#), [1005](#)
11 MPI_T_EVENT_HANDLE_FREE, [731](#), [741](#),
12 [1005](#)
13 MPI_T_EVENT_HANDLE_GET_INFO, [728](#),
14 [729](#), [1005](#)
15 MPI_T_EVENT_HANDLE_SET_INFO, [728](#),
16 [1005](#)
17 MPI_T_EVENT_READ, [724](#), [726](#), [733](#), [734](#),
18 [1005](#)
19 MPI_T_EVENT_REGISTER_CALLBACK, [729](#),
20 [730](#), [1005](#)
21 MPI_T_EVENT_SET_DROPPED_HANDLER,
22 [732](#), [733](#), [1005](#)
23 MPI_T_FINALIZE, [699](#)
24 C example usage, [718](#)
25 MPI_T_INIT_THREAD, [698](#), [699](#)
26 C example usage, [718](#)
27 MPI_T_PVAR_GET_INDEX, [713](#), [741](#), [1008](#)
28 MPI_T_PVAR_GET_INFO, [700](#), [710](#), [711](#), [712](#),
29 [714](#)–[718](#), [740](#), [741](#), [1007](#), [1008](#)
30 C example usage, [718](#)
31 MPI_T_PVAR_GET_NUM, [711](#), [715](#)
32 MPI_T_PVAR_HANDLE_ALLOC, [700](#), [714](#),
33 [715](#)–[717](#), [741](#), [1002](#)
34 C example usage, [718](#)
35 MPI_T_PVAR_HANDLE_FREE, [715](#), [741](#), [1002](#),
36 [1007](#)
37 C example usage, [718](#)
38 MPI_T_PVAR_READ, [716](#), [718](#), [724](#), [741](#), [1007](#)
39 C example usage, [718](#)
40 MPI_T_PVAR_READRESET, [712](#), [717](#), [718](#),
41 [724](#), [741](#), [1007](#)
42 MPI_T_PVAR_RESET, [717](#), [718](#), [724](#), [741](#),
43 [1002](#), [1007](#)
44 MPI_T_PVAR_SESSION_CREATE, [713](#), [741](#)
45 C example usage, [718](#)
46 MPI_T_PVAR_SESSION_FREE, [714](#), [741](#)
47 MPI_T_PVAR_START, [715](#), [724](#), [741](#), [1002](#),
48 [1007](#)
C example usage, [718](#)
MPI_T_PVAR_STOP, [716](#), [724](#), [741](#), [1002](#), [1007](#)
MPI_T_PVAR_WRITE, [717](#), [724](#), [741](#), [1002](#),
[1007](#)
MPI_T_SOURCE_GET_INFO, [721](#), [722](#), [735](#),
[741](#), [1005](#)
MPI_T_SOURCE_GET_NUM, [721](#), [741](#), [1005](#)
MPI_T_SOURCE_GET_TIMESTAMP, [723](#), [724](#),
[741](#), [1005](#)
MPI_TEST, [39](#), [67](#), [68](#), [69](#), [71](#), [72](#), [74](#), [79](#), [80](#),
[88](#), [96](#), [99](#), [100](#), [106](#), [107](#), [261](#), [469](#),
[602](#), [603](#), [626](#), [627](#)
MPI_TEST_CANCELLED, [68](#), [69](#), [89](#), [601](#), [608](#),
[628](#)
C example usage, [470](#)
MPI_TESTALL, [72](#), [75](#), [76](#), [491](#), [601](#), [602](#), [605](#)
MPI_TESTANY, [72](#), [73](#), [74](#), [78](#), [491](#), [601](#), [602](#),
[605](#)
MPI_TESTSOME, [72](#), [77](#), [78](#), [491](#), [601](#), [602](#), [605](#)
MPI_TOPO_TEST, [369](#), [381](#)
MPI_TYPE_C2F, [803](#)
MPI_TYPE_COMMIT, [144](#), [804](#)
C example usage, [161](#), [163](#), [164](#), [171](#),
[190](#)–[194](#), [201](#), [236](#)
Fortran example usage, [144](#), [158](#)–[160](#), [543](#),
[786](#)
MPI_TYPE_CONTIGUOUS, [15](#), [115](#), [118](#), [138](#),
[152](#), [610](#), [664](#)
C example usage, [190](#)
Fortran example usage, [147](#), [149](#)
Language-independent example usage,
[116](#), [138](#)
MPI_Type_contiguous_c, [115](#), [1003](#)
MPI_TYPE_CREATE_DARRAY, [15](#), [38](#), [130](#),
[131](#), [152](#)
Fortran example usage, [134](#)
MPI_Type_create_darray_c, [130](#), [1003](#)
MPI_TYPE_CREATE_F90_COMPLEX, [15](#), [152](#),
[155](#), [216](#), [666](#), [756](#), [775](#), [776](#), [777](#)
MPI_TYPE_CREATE_F90_INTEGER, [15](#), [152](#),
[155](#), [215](#), [666](#), [756](#), [775](#), [776](#), [777](#)
MPI_TYPE_CREATE_F90_REAL, [15](#), [152](#), [155](#),
[215](#), [666](#), [756](#), [774](#), [775](#)–[777](#), [1015](#)
MPI_TYPE_CREATE_HINDEXED, [15](#), [22](#), [115](#),
[121](#), [122](#), [125](#), [126](#), [152](#), [751](#), [800](#)
MPI_Type_create_hindexed_c, [122](#), [1003](#)
MPI_TYPE_CREATE_HINDEXED_BLOCK, [15](#),
[115](#), [124](#), [152](#), [800](#), [1010](#)
MPI_Type_create_hindexed_block_c, [124](#),
[1003](#)
MPI_TYPE_CREATE_HVECTOR, [15](#), [22](#), [115](#),
[118](#), [152](#), [751](#)
C example usage, [161](#), [163](#)
Fortran example usage, [158](#), [160](#)
MPI_Type_create_hvector_c, [118](#), [1003](#)
MPI_TYPE_CREATE_INDEXED_BLOCK, [15](#),
[123](#), [124](#), [152](#)
Fortran example usage, [543](#)
MPI_Type_create_indexed_block_c, [123](#),
[1003](#)

- MPI_TYPE_CREATE_KEYVAL, [344](#), [354](#), [357](#),
[810](#), [834](#), [837](#), [1017](#)
- MPI_TYPE_CREATE_RESIZED, [22](#), [115](#), [138](#),
[141](#), [142](#), [152](#), [664](#), [752](#), [1013](#)
C example usage, [164](#)
Fortran example usage, [786](#)
MPI_Type_create_resized_c, [141](#), [1003](#)
- MPI_TYPE_CREATE_STRUCT, [15](#), [22](#), [115](#),
[125](#), [126](#), [139](#), [152](#), [211](#), [751](#), [800](#)
C example usage, [161](#), [163](#), [171](#), [193](#), [194](#),
[236](#)
Fortran example usage, [160](#), [786](#), [814](#)
Language-independent example usage,
[126](#), [138](#)
MPI_Type_create_struct_c, [125](#), [1003](#)
- MPI_TYPE_CREATE_SUBARRAY, [15](#), [18](#), [127](#),
[129](#), [131](#), [152](#)
C example usage, [687](#)
MPI_Type_create_subarray_c, [127](#), [1003](#)
- MPI_TYPE_DELETE_ATTR, [344](#), [357](#), [358](#),
[1013](#)
- MPI_TYPE_DUP, [15](#), [145](#), [146](#), [152](#), [1013](#)
- MPI_TYPE_DUP_FN, [355](#), [825](#), [1007](#)
- MPI_TYPE_EXTENT, [22](#), [751](#), [1009](#)
Fortran example usage, [543](#)
- MPI_TYPE_F2C, [803](#)
- MPI_TYPE_FREE, [145](#), [155](#), [356](#), [469](#), [475](#)
Fortran example usage, [543](#)
- MPI_TYPE_FREE_KEYVAL, [344](#), [356](#), [358](#)
- MPI_TYPE_GET_ATTR, [344](#), [357](#), [358](#), [764](#),
[810](#), [1013](#)
- MPI_TYPE_GET_CONTENTS, [151](#), [152](#), [153](#),
[154](#)–[156](#), [800](#)
C example usage, [165](#)
MPI_Type_get_contents_c, [153](#), [1003](#)
- MPI_TYPE_GET_ENVELOPE, [151](#), [152](#), [154](#),
[155](#), [776](#), [800](#)
C example usage, [165](#)
MPI_Type_get_envelope_c, [151](#), [1003](#)
- MPI_TYPE_GET_EXTENT, [22](#), [140](#), [143](#), [751](#),
[779](#), [808](#)
C example usage, [161](#)
Fortran example usage, [158](#), [160](#), [545](#), [547](#)
MPI_Type_get_extent_c, [140](#), [1003](#)
- MPI_TYPE_GET_EXTENT_X, [140](#), [1010](#)
- MPI_TYPE_GET_NAME, [363](#), [1001](#), [1013](#)
- MPI_TYPE_GET_TRUE_EXTENT, [142](#)
MPI_Type_get_true_extent_c, [142](#), [1003](#)
- MPI_TYPE_GET_TRUE_EXTENT_X, [142](#), [143](#),
[1010](#)
- MPI_TYPE_HINDEXED, [22](#), [751](#), [1009](#)
- MPI_TYPE_HVECTOR, [22](#), [751](#), [1009](#)
- MPI_TYPE_INDEXED, [15](#), [119](#), [120](#), [121](#), [123](#),
[152](#)
C example usage, [161](#), [163](#)
Fortran example usage, [159](#)
Language-independent example usage, [120](#)
MPI_Type_indexed_c, [120](#), [1003](#)
- MPI_TYPE_LB, [22](#), [751](#), [1009](#)
- MPI_TYPE_MATCH_SIZE, [756](#), [778](#), [779](#), [1013](#)
- MPI_TYPE_NULL_COPY_FN, [355](#), [825](#), [1007](#)
- MPI_TYPE_NULL_DELETE_FN, [355](#), [825](#),
[1007](#), [1013](#)
- MPI_TYPE_SET_ATTR, [344](#), [356](#), [358](#), [764](#),
[810](#), [813](#), [1013](#)
- MPI_TYPE_SET_NAME, [362](#), [1013](#)
- MPI_TYPE_SIZE, [137](#), [138](#), [1010](#)
C example usage, [692](#)
MPI_Type_size_c, [137](#), [1003](#)
- MPI_TYPE_SIZE_X, [137](#), [138](#), [1010](#)
- MPI_TYPE_STRUCT, [22](#), [751](#), [1009](#)
- MPI_TYPE_UB, [22](#), [751](#), [1009](#)
- MPI_TYPE_VECTOR, [15](#), [116](#), [117](#), [118](#), [121](#),
[152](#)
C example usage, [191](#), [192](#), [194](#), [201](#)
Fortran example usage, [158](#), [160](#)
Language-independent example usage, [117](#)
MPI_Type_vector_c, [117](#), [1003](#)
- MPI_UNPACK, [168](#), [169](#), [173](#), [671](#)
C example usage, [171](#), [172](#)
MPI_Unpack_c, [168](#), [1003](#)
- MPI_UNPACK_EXTERNAL, [8](#), [175](#), [777](#)
MPI_Unpack_external_c, [175](#), [1003](#)
- MPI_UNPUBLISH_NAME, [445](#), [511](#), [512](#)
C example usage, [513](#)
- MPI_USER_FUNCTION
Fortran example usage, [226](#)
- MPI_WAIT, [37](#), [39](#), [67](#), [68](#), [69](#)–[73](#), [75](#), [88](#), [96](#),
[99](#), [100](#), [106](#), [107](#), [237](#), [261](#), [288](#), [469](#),
[492](#), [599](#), [602](#), [603](#), [626](#), [627](#), [653](#), [674](#),
[675](#), [783](#), [789](#), [790](#), [793](#)
C example usage, [286](#)–[288](#)
Fortran example usage, [70](#)–[72](#), [78](#), [640](#)
- MPI_WAITALL, [72](#), [74](#), [75](#), [76](#), [238](#), [289](#), [491](#),
[492](#), [552](#), [601](#), [602](#), [605](#)
C example usage, [288](#), [289](#), [594](#)
- MPI_WAITANY, [51](#), [72](#), [73](#), [78](#), [491](#), [492](#), [601](#),
[602](#), [605](#)
C example usage, [594](#)
Fortran example usage, [78](#)
- MPI_WAITSOME, [72](#), [76](#), [77](#), [78](#), [491](#), [492](#), [601](#),
[602](#), [605](#)
Fortran example usage, [78](#)
- MPI_WIN_ALLOCATE, [522](#), [525](#), [526](#), [529](#), [535](#),
[536](#), [541](#), [572](#), [768](#)–[770](#), [1004](#), [1006](#)
MPI_Win_allocate_c, [525](#), [1003](#)
- MPI_WIN_ALLOCATE_CPTR, [527](#), [1006](#)
- MPI_WIN_ALLOCATE_SHARED, [319](#), [522](#), [527](#),

- 1 528, 530, 535, 536, 572, 769, 770,
2 1004–1006
- 3 MPI_Win_allocate_shared_c, [527](#), [1003](#)
- 4 MPI_WIN_ALLOCATE_SHARED_CPTR, [529](#),
5 1006
- 6 MPI_WIN_ATTACH, [531](#), [532](#), [533](#), [534](#), [572](#)
7 C example usage, [595](#)
- 8 MPI_WIN_C2F, [803](#)
- 9 MPI_WIN_CALL_ERRHANDLER, [449](#), [450](#)
- 10 MPI_WIN_COMPLETE, [535](#), [561](#), [566](#),
11 567–569, 578, 584
12 C example usage, [566](#), [590](#), [591](#)
13 Language-independent example usage,
14 [584](#), [585](#)
- 15 MPI_WIN_CREATE, [492](#), [522](#), [525–527](#), [529](#),
16 532, 533, 535, 536, 577
17 Fortran example usage, [543](#), [545](#), [547](#)
18 MPI_Win_create_c, [523](#), [1003](#)
- 19 MPI_WIN_CREATE_DYNAMIC, [445](#), [522](#), [531](#),
20 [532](#), [533–536](#), [577](#)
21 C example usage, [595](#)
- 22 MPI_WIN_CREATE_ERRHANDLER, [434](#), [436](#),
23 [438](#), [835](#), [837](#), [1013](#)
- 24 MPI_WIN_CREATE_KEYVAL, [344](#), [351](#), [357](#),
25 [810](#), [834](#), [837](#), [1017](#)
- 26 MPI_WIN_DELETE_ATTR, [344](#), [354](#), [358](#)
- 27 MPI_WIN_DETACH, [531](#), [534](#), [535](#)
28 C example usage, [595](#)
- 29 MPI_WIN_DUP_FN, [351](#), [825](#), [1007](#)
- 30 MPI_WIN_F2C, [803](#)
- 31 MPI_WIN_FENCE, [535](#), [543](#), [561](#), [563](#), [564](#),
32 [565](#), [575](#), [576](#), [578](#), [579](#), [582](#), [586](#), [794](#)
33 C example usage, [589](#)
34 Fortran example usage, [543](#), [545](#), [547](#)
35 Language-independent example usage, [589](#)
- 36 MPI_WIN_FLUSH, [530](#), [552](#), [554](#), [573](#), [578](#), [592](#),
37 [593](#)
38 C example usage, [595](#)
39 Language-independent example usage,
40 [583](#), [592](#), [593](#)
- 41 MPI_WIN_FLUSH_ALL, [552](#), [554](#), [573](#), [578](#)
42 Language-independent example usage, [592](#)
- 43 MPI_WIN_FLUSH_LOCAL, [552](#), [574](#), [578](#)
44 Language-independent example usage, [582](#)
- 45 MPI_WIN_FLUSH_LOCAL_ALL, [552](#), [574](#), [578](#)
- 46 MPI_WIN_FREE, [352](#), [469](#), [475](#), [519](#), [535](#), [1002](#)
47 Fortran example usage, [545](#), [547](#)
- 48 MPI_WIN_FREE_KEYVAL, [344](#), [352](#), [358](#)
- 49 MPI_WIN_GET_ATTR, [344](#), [353](#), [358](#), [536](#), [764](#),
50 [810](#), [813](#)
- 51 MPI_WIN_GET_ERRHANDLER, [434](#), [438](#), [1019](#)
- 52 MPI_WIN_GET_GROUP, [536](#), [537](#)
- 53 MPI_WIN_GET_INFO, [537](#), [538](#), [1004](#), [1011](#)
- 54 MPI_WIN_GET_NAME, [364](#), [1001](#)
- 55 MPI_WIN_LOCK, [524](#), [535](#), [561](#), [570](#), [571](#), [572](#),
56 [575](#), [576](#), [578](#), [582–584](#)
57 C example usage, [572](#)
58 Language-independent example usage,
59 [582–585](#)
- 60 MPI_WIN_LOCK_ALL, [524](#), [561](#), [570](#), [571](#), [572](#),
61 [575](#), [576](#), [578](#), [584](#), [592](#)
62 C example usage, [594](#), [595](#)
63 Language-independent example usage,
64 [583](#), [593](#)
- 65 MPI_WIN_NULL_COPY_FN, [351](#), [825](#), [1007](#)
- 66 MPI_WIN_NULL_DELETE_FN, [351](#), [825](#), [1007](#)
- 67 MPI_WIN_POST, [535](#), [561](#), [565](#), [566](#), [567–569](#),
68 [572](#), [575](#), [576](#), [578](#), [585](#), [586](#)
69 C example usage, [590](#), [591](#)
70 Language-independent example usage,
71 [584](#), [585](#)
- 72 MPI_WIN_SET_ATTR, [344](#), [353](#), [358](#), [536](#), [764](#),
73 [810](#), [813](#)
- 74 MPI_WIN_SET_ERRHANDLER, [434](#), [437](#)
- 75 MPI_WIN_SET_INFO, [537](#), [538](#), [1004](#), [1011](#)
- 76 MPI_WIN_SET_NAME, [363](#)
- 77 MPI_WIN_SHARED_QUERY, [528](#), [530](#), [769](#),
78 [770](#), [1006](#)
79 MPI_Win_shared_query_c, [530](#), [1003](#)
- 80 MPI_WIN_SHARED_QUERY_CPTR, [531](#), [1006](#)
- 81 MPI_WIN_START, [535](#), [561](#), [565](#), [566](#), [567](#),
82 [569](#), [575](#), [576](#), [585](#), [591](#)
83 C example usage, [566](#), [590](#), [591](#)
84 Language-independent example usage,
85 [584](#), [585](#)
- 86 MPI_WIN_SYNC, [574](#), [575](#), [578](#), [580](#), [581](#), [584](#),
87 [592](#), [593](#), [1001](#)
88 Language-independent example usage,
89 [582](#), [592](#), [593](#)
- 90 MPI_WIN_TEST, [568](#), [1001](#)
- 91 MPI_WIN_UNLOCK, [535](#), [554](#), [561](#), [571](#), [572](#),
92 [578](#), [582](#), [583](#)
93 C example usage, [572](#)
94 Language-independent example usage,
95 [582–585](#)
- 96 MPI_WIN_UNLOCK_ALL, [554](#), [561](#), [571](#), [578](#),
97 [582](#), [592](#)
98 C example usage, [594](#), [595](#)
99 Language-independent example usage, [583](#)
- 100 MPI_WIN_WAIT, [535](#), [561](#), [567](#), [568](#), [569](#), [572](#),
101 [578](#), [582](#), [584](#), [585](#)
102 C example usage, [590](#), [591](#)
103 Language-independent example usage,
104 [584](#), [585](#)
- 105 MPI_WTICK, [24](#), [451](#), [723](#)
- 106 MPI_WTIME, [14](#), [24](#), [428](#), [451](#), [710](#), [723](#)
107 C example usage, [451](#), [692](#)
- 108 MPIEXEC

Language-independent example usage,	1
464 , 489	2
mpiexec, 463–466 , 487 , 488 , 1005	3
mpirun, 487	4
PMPI_, 689 , 764	5
PMPI_AINT_ADD, 24	6
PMPI_AINT_DIFF, 24	7
PMPI_ISEND, 764 , 766 , 767	8
PMPI_WTICK, 24	9
PMPI_WTIME, 24	10
	11
	12
	13
	14
	15
	16
	17
	18
	19
	20
	21
	22
	23
	24
	25
	26
	27
	28
	29
	30
	31
	32
	33
	34
	35
	36
	37
	38
	39
	40
	41
	42
	43
	44
	45
	46
	47
	48