MPI: A Message-Passing Interface Standard Version 4.0

(Draft)

Unofficial, for comment only

Message Passing Interface Forum

December 22, 2020

This document describes the 2020 Draft Specification of the Message-Passing Interface $\mathbf{2}$ (MPI) standard, intended for comment. It is not an official version of the standard. The MPI standard includes point-to-point message-passing, collective communications, group and communicator concepts, process topologies, environmental management, process cre-ation and management, one-sided communications, extended collective operations, external interfaces, I/O, some miscellaneous topics, and multiple tool interfaces. Language bindings $\overline{7}$ for C and Fortran are defined. Historically, the evolution of the standards is from MPI-1.0 (May 5, 1994) to MPI-1.1 (June 12, 1995) to MPI-1.2 (July 18, 1997), with several clarifications and additions and published as part of the MPI-2 document, to MPI-2.0 (July 18, 1997), with new functionality, to MPI-1.3 (May 30, 2008), combining for historical reasons the documents 1.1 and 1.2 and some errata documents to one combined document, and to MPI-2.1 (June 23, 2008), combining the previous documents. Version MPI-2.2 (September 4, 2009) added additional clarifications and seven new routines. Version MPI-3.0 (September 21, 2012) is an extension of MPI-2.2. Version MPI-3.1 (June 4, 2015) adds clarifications and minor extensions to MPI-3.0. Comments. Please send comments on MPI to the MPI Forum as follows: 1. Subscribe to https://lists.mpi-forum.org/mailman/listinfo/mpi-comments 2. Send your comment to: mpi-comments@lists.mpi-forum.org, together with the version of the MPI standard and the page and line numbers on which you are commenting. Your comment will be forwarded to MPI Forum committee members for consideration. Messages sent from an unsubscribed e-mail address will not be considered.

©1993, 1994, 1995, 1996, 1997, 2008, 2009, 2012, 2015, 2020 University of Tennessee, Knoxville, Tennessee. Permission to copy without fee all or part of this material is granted, provided the University of Tennessee copyright notice and the title of this document appear, and notice is given that copying is by permission of the University of Tennessee.

Unofficial Draft for Comment Only

2020 Draft Specification, November, 2020. 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 a number of new features that will be present in the final MPI-4.0 document. 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 interfaces and parallel I/O. A miscellany chapter discusses items that do not fit elsewhere, in particular language interoperability.

 $\mathbf{5}$

 24

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

8

⁹ Version 1.1: June, 1995. Beginning in March, 1995, the Message-Passing Interface Forum
 ¹⁰ reconvened to correct errors and make clarifications in the MPI document of May 5, 1994,
 ¹¹ referred to below as Version 1.0. These discussions resulted in Version 1.1. The changes
 ¹² from Version 1.0 are minor. A version of this document with all changes marked is available.

13

48

¹⁴ Version 1.0: May, 1994. The Message-Passing Interface Forum, with participation from
 ¹⁵ over 40 organizations, has been meeting since January 1993 to discuss and define a set of
 ¹⁶ library interface standards for message passing. The Message-Passing Interface Forum is
 ¹⁷ not sanctioned or supported by any official standards organization.

¹⁸ The goal of the Message-Passing Interface, simply stated, is to develop a widely used ¹⁹ standard for writing message-passing programs. As such the interface should establish a ²⁰ practical, portable, efficient, and flexible standard for message-passing.

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

	draft was processed by LATEX on May 5, 1994.
24	
25	
26	
27	
28	
29	
30	
31	
32	
33	
34	
35	
36	
37	
38	
39	
10	
11	
12	
13	
14	
15	
16	
17	

Contents

Li	st of l	Figures				Х	viii
Li	st of 7	Tables					xx
A	cknow	ledgments					xxii
1	Intro	eduction 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 2020 Draft Specification					5
	1.9	Who Should Use This Standard?					5
	1.10	What Platforms Are Targets for Implementation?					5
	1.11	What Is Included in the Standard?					5
	1.12	What Is Not Included in the Standard?					6
	1.13	Organization of This Document		•	•	•	6
2	MPI	Terms and Conventions					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					13
		2.4.3 MPI Datatypes					15
	2.5	Datatypes					16
		2.5.1 Opaque Objects					16
		2.5.2 Array Arguments					18
		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					20
		2.5.7 File Offsets					20
		2.5.8 Counts					20
	2.6	Language Binding					21

		2.6.1	Deprecated and Removed Interfaces
		2.6.2	Fortran Binding Issues
		2.6.3	C Binding Issues
		2.6.4	Functions and Macros
	2.7	Proces	ses
	2.8	Error	Handling
	2.9		nentation Issues
		2.9.1	Independence of Basic Runtime Routines
		2.9.2	Interaction with Signals
	2.10	Examp	ples $\ldots \ldots 27$
3	Poin	t_to_Pe	pint Communication 29
J	3.1		uction $\dots \dots \dots$
	3.2	Blocki	ng Send and Receive Operations
	3.2	2 0 1	ng Send and Receive Operations 30 Blocking Send 30
		3.2.1 3.2.2	Message Data
		3.2.2 3.2.3	
		3.2.3 3.2.4	Message Envelope 33 Blocking Receive 34
			0
		3.2.5	
		3.2.6	0
		3.2.7	Send-Receive
	3.3		pe Matching and Data Conversion
		3.3.1	Type Matching Rules 43
			Type MPI_CHARACTER 45
		3.3.2	Data Conversion 46
	3.4		unication Modes
	3.5		tics of Point-to-Point Communication
	3.6		Allocation and Usage 56
		3.6.1	Model Implementation of Buffered Mode 58
	3.7	Nonble	ocking Communication
		3.7.1	Communication Request Objects
		3.7.2	Communication Initiation
		3.7.3	Communication Completion
		3.7.4	Semantics of Nonblocking Communications
		3.7.5	Multiple Completions 73
		3.7.6	Non-Destructive Test of status
	3.8	Probe	and Cancel
		3.8.1	Probe
		3.8.2	Matching Probe
		3.8.3	Matched Receives
		3.8.4	Cancel
	3.9		cent Communication Requests
	3.10		Processes
4	Part	itioned	Point-to-Point Communication 101
-1	4.1		$uction \dots \dots$
	4.1		tics of Partitioned Point-to-Point Communication
	4.4	4.2.1	Communication Initialization and Starting with Partitioning 102
		7.4.1	- Communication intranzation and starting with ratificining 104

		4.2.2	Communication Completion under Partitioning	108
		4.2.3	Semantics of Communications in Partitioned Mode	109
	4.3	Partitic	oned Communication Examples	110
		4.3.1	Partition Communication with Threads/Tasks Using OpenMP 4.0	
		4.9.9	or later	
		4.3.2	Send-only Partitioning Example with Tasks and OpenMP version	
		4.3.3	4.0 or later	
		4.3.3	Send and Receive Partitioning Example with OpenMP version 4.0 or later	
			or later	119
5	Data	atypes		117
	5.1	Derived	l Datatypes	117
		5.1.1	Type Constructors with Explicit Addresses	119
		5.1.2		
		5.1.3	Datatype Constructors	131
		5.1.4	Distributed Array Datatype Constructor	133
		5.1.5	Address and Size Functions	139
		5.1.6	Lower-Bound and Upper-Bound Markers	142
		5.1.7	Extent and Bounds of Datatypes	145
		5.1.8	True Extent of Datatypes	
		5.1.9	Commit and Free	148
		5.1.10	Duplicating a Datatype	150
		5.1.11	Use of General Datatypes in Communication	151
		5.1.12	Correct Use of Addresses	
		5.1.13	Decoding a Datatype	155
		5.1.14	Examples	
	5.2	Pack a	nd Unpack	172
	5.3		cal MPI_PACK and MPI_UNPACK	
6			Communication	185
	6.1		action and Overview	
	6.2	Comm	micator Argument	
		6.2.1	Specifics for Intra-Communicator Collective Operations	188
		6.2.2	Applying Collective Operations to Inter-Communicators	189
		6.2.3	Specifics for Inter-Communicator Collective Operations	190
	6.3	Barrier	Synchronization	191
	6.4	Broadc		
		6.4.1	Example using MPI_BCAST	193
	6.5	Gather		194
		6.5.1	Examples using MPI_GATHER, MPI_GATHERV	198
	6.6	Scatter		204
		6.6.1	Examples using MPI_SCATTER, MPI_SCATTERV	208
	6.7	Gather	-to-all	211
		6.7.1	Example using MPI_ALLGATHER	214
	6.8	All-to-A	All Scatter/Gather	215
	6.9	Global	Reduction Operations	221
		6.9.1	Reduce	222
		6.9.2	Predefined Reduction Operations	

		6.9.3	Signed Characters and Reductions	227
		6.9.4	MINLOC and MAXLOC	227
		6.9.5	User-Defined Reduction Operations	231
			Example of User-Defined Reduce	
		6.9.6	All-Reduce	236
		6.9.7	Process-Local Reduction	238
	6.10	Reduce	-Scatter	240
		6.10.1	MPI_REDUCE_SCATTER_BLOCK	240
		6.10.2	MPI_REDUCE_SCATTER	241
	6.11	Scan .		243
		6.11.1	Inclusive Scan	243
		6.11.2	Exclusive Scan	245
		6.11.3	Example using MPI_SCAN	246
	6.12	Nonblo	cking Collective Operations	248
		6.12.1	Nonblocking Barrier Synchronization	250
		6.12.2	Nonblocking Broadcast	251
			Example using MPI_IBCAST	251
		6.12.3	Nonblocking Gather	252
		6.12.4	Nonblocking Scatter	255
		6.12.5	Nonblocking Gather-to-all	258
		6.12.6	Nonblocking All-to-All Scatter/Gather	261
		6.12.7	Nonblocking Reduce	
		6.12.8	Nonblocking All-Reduce	268
		6.12.9	Nonblocking Reduce-Scatter with Equal Blocks	269
		6.12.10	Nonblocking Reduce-Scatter	271
		6.12.11	Nonblocking Inclusive Scan	272
		6.12.12	Nonblocking Exclusive Scan	273
	6.13	Persiste	ent Collective Operations	274
		6.13.1	Persistent Barrier Synchronization	275
		6.13.2	Persistent Broadcast	276
		6.13.3	Persistent Gather	277
		6.13.4	Persistent Scatter	281
		6.13.5	Persistent Gather-to-all	284
		6.13.6	Persistent All-to-All Scatter/Gather	287
		6.13.7	Persistent Reduce	292
		6.13.8	Persistent All-Reduce	293
		6.13.9	Persistent Reduce-Scatter with Equal Blocks	294
		6.13.10	Persistent Reduce-Scatter	296
		6.13.11	Persistent Inclusive Scan	297
		6.13.12	Persistent Exclusive Scan	298
	6.14	Correct	ness	299
7			ntexts, Communicators, and Caching	307
	7.1		ction	307
		7.1.1	Features Needed to Support Libraries	307
		7.1.2	MPI's Support for Libraries	308
	7.2		Concepts	310
		7.2.1	Groups	310

		7.2.2	Contexts	0
		7.2.3	Intra-Communicators	1
		7.2.4	Predefined Intra-Communicators	1
	7.3	Group	Management	2
		7.3.1	Group Accessors	2
		7.3.2	Group Constructors	
		7.3.3	Group Destructors	
	7.4		micator Management	
		7.4.1	Communicator Accessors	
		7.4.2	Communicator Constructors	
		7.4.3	Communicator Destructors	
		7.4.4	Communicator Info	
	7.5		ting Examples	
		7.5.1	Current Practice $\#1$	
		7.5.2	Current Practice $\#2$	
		7.5.3	(Approximate) Current Practice #3	
		7.5.4	Example $#4$	
		7.5.5	Library Example $\#1$	
		7.5.6	Library Example $\#2$	
	7.6	Inter-C	ommunication	
		7.6.1	Inter-Communicator Accessors	
		7.6.2	Inter-Communicator Operations	
		7.6.3	Inter-Communication Examples	
			Example 1: Three-Group "Pipeline"	8
			Example 2: Three-Group "Ring"	9
	7.7	Caching		1
		7.7.1	Functionality	1
		7.7.2	Communicators	2
		7.7.3	Windows	8
		7.7.4	Datatypes	2
		7.7.5	Error Class for Invalid Keyval	5
		7.7.6	Attributes Example	5
	7.8	Naming	g Objects	7
	7.9		izing the Loosely Synchronous Model	2
		7.9.1	Basic Statements	2
		7.9.2	Models of Execution	2
			Static Communicator Allocation	3
			Dynamic Communicator Allocation	3
			The General Case	3
8		ess Top		
	8.1		$\begin{array}{c} \text{uction} \dots \dots \dots \dots \dots \dots \dots \dots \dots $	
	8.2		Topologies 38 1 1	
	8.3		$\dim_{\mathcal{H}} \inf_{\mathcal{H}} MPI \dots \dots \dots \dots \dots \dots \dots \dots \dots $	
	8.4		ew of the Functions	
	8.5		$gy Constructors \dots \dots$	
		8.5.1	Cartesian Constructor	
		8.5.2	Cartesian Convenience Function: MPI_DIMS_CREATE	9

		8.5.3	Graph Constructor	390
		8.5.4	Distributed Graph Constructor	392
		8.5.5	Topology Inquiry Functions	399
		8.5.6	Cartesian Shift Coordinates	407
		8.5.7	Partitioning of Cartesian Structures	409
		8.5.8	Low-Level Topology Functions	410
	8.6	Neighb	orhood Collective Communication	412
		8.6.1	Neighborhood Gather	413
		8.6.2	Neighbor Alltoall	417
	8.7	Nonblo	cking Neighborhood Communication	425
		8.7.1	Nonblocking Neighborhood Gather	425
		8.7.2	Nonblocking Neighborhood Alltoall	428
	8.8	Persiste	ent Neighborhood Communication	432
		8.8.1	Persistent Neighborhood Gather	433
		8.8.2	Persistent Neighborhood Alltoall	436
	8.9	An Ap	plication Example	440
9			nmental Management	447
	9.1		nentation Information	447
		9.1.1	Version Inquiries	447
		9.1.2	Environmental Inquiries	449
			Tag Values	449
			Host Rank	449
			IO Rank	450
			Clock Synchronization	450
			Inquire Processor Name	450
	9.2		y Allocation	451
	9.3		Iandling	
		9.3.1	Error Handlers for Communicators	457
		9.3.2	Error Handlers for Windows	459
		9.3.3	Error Handlers for Files	461
		9.3.4	Error Handlers for Sessions	463
		9.3.5	Freeing Errorhandlers and Retrieving Error Strings	465
	9.4	Error C	Codes and Classes	466
	9.5	Error (Classes, Error Codes, and Error Handlers	469
	9.6	Timers	and Synchronization	473
10				
10	The	Info Ob	oject	475
11	Proc	ess Init	tialization, Creation, and Management	483
	11.1		iction	483
	11.2		orld Model	484
		11.2.1	Starting MPI Processes	484
		11.2.2	Finalizing MPI	490
		11.2.3	Determining Whether MPI Has Been Initialized When Using the	
			World Model	493
		11.2.4	Allowing User Functions at MPI Finalization	494
	11.3		ssions Model	495
		-		

	11.3.1	Session Creation and Destruction Methods	. 497
	11.3.2	Processes Sets	. 498
	11.3.3	Runtime Query Functions	. 499
	11.3.4	Sessions Model Examples	. 502
11.4	Commo	on Elements of Both Process Models	. 508
	11.4.1	MPI Functionality that is Always Available	. 508
	11.4.2	Aborting MPI Processes	
11.5	Portabl	e MPI Process Startup	
11.6		d Threads	
	11.6.1	General	
	11.6.2	Clarifications	
11.7	The Dy	namic Process Model	
	11.7.1	Starting Processes	
	11.7.2	The Runtime Environment	
11.8		Manager Interface	
11.0	11.8.1	Processes in MPI	. 517
	11.8.2	Starting Processes and Establishing Communication	
	11.8.3	Starting Multiple Executables and Establishing Communication	
	11.8.4	Reserved Keys	
	11.8.5	Spawn Example	
11.9		shing Communication	
11.0	11.9.1	Names, Addresses, Ports, and All That	
	11.9.2	Server Routines	
	11.9.2	Client Routines	. 532
	11.9.4	Name Publishing	. 534
	11.9.4 11.9.5	Reserved Key Values	
	11.9.6	Client/Server Examples	
11 10		Sunctionality	. 539
11.10	11 10 1	Unctionality	. 539
		Singleton MPI Initialization	
		MPI_APPNUM	
		Releasing Connections	
	11.10.5	Another Way to Establish MPI Communication	. 345
12 One-	Sided C	Communications	545
12.1		ction	
12.2		ation	
12.2	12.2.1	Window Creation	
	12.2.2	Window That Allocates Memory	
	12.2.2	Window That Allocates Shared Memory	
	12.2.4	Window of Dynamically Attached Memory	
	12.2.4 12.2.5	Window Destruction	
	12.2.6 12.2.6	Window Attributes	
	12.2.0 12.2.7	Window Info	
12.3		mication Calls	
12.0	12.3.1	Put	
	12.3.1 12.3.2	Get	
	12.3.2 12.3.3	Examples for Communication Calls	
	14.0.0	Examples for Communication Calls	. 500

	12.3.4	Accumulate Functions
		Accumulate Function
		Get Accumulate Function
		Fetch and Op Function
		Compare and Swap Function
	12.3.5	Request-based RMA Communication Operations 578
12.4	Memor	y Model
12.5	Synchr	onization Calls
	12.5.1	Fence
	12.5.2	General Active Target Synchronization
	12.5.3	Lock
	12.5.4	Flush and Sync
	12.5.5	Assertions
	12.5.6	Miscellaneous Clarifications
12.6	Error H	$\mathbf{Handling} \dots \dots \dots \dots \dots \dots \dots \dots \dots $
	12.6.1	Error Handlers
	12.6.2	Error Classes
12.7	-	tics and Correctness
	12.7.1	Atomicity
	12.7.2	Ordering
	12.7.3	Progress
	12.7.4	Registers and Compiler Optimizations
12.8		les \ldots
12.0	Lincinp	
13 Exte		terfaces 625
13.1	Introdu	action 625 lized Requests 625
13.2	Genera	lized Requests
	13.2.1	Examples
13.3	Associa	ating Information with Status
14 I/O		635
14.1		$action \dots \dots$
	14.1.1	Definitions
14.2		$anipulation \dots \dots$
	14.2.1	Opening a File
	14.2.2	Closing a File
	14.2.3	Deleting a File
	14.2.4	Resizing a File
	14.2.5	Preallocating Space for a File
	14.2.6	Querying the Size of a File
	14.2.7	Querying File Parameters
	14.2.8	File Info 645
		Reserved File Hints
14.3	File Vi	ews
14.4	Data A	ccess
	14.4.1	Data Access Routines
	14.4.1	Data Access noutlies
	14.4.1	Positioning

		Coordination	653
		Data Access Conventions	653
	14.4.2	Data Access with Explicit Offsets	654
	14.4.3	Data Access with Individual File Pointers	661
	14.4.4	Data Access with Shared File Pointers	
		Noncollective Operations	
		Collective Operations	
		Seek	
	14.4.5	Split Collective Data Access Routines	
14.5	File Int	eroperability	689
	14.5.1	Datatypes for File Interoperability	
	14.5.2	External Data Representation: "external32"	
	14.5.3	User-Defined Data Representations	696
	11.0.0	Extent Callback	
		Datarep Conversion Functions	698
	14.5.4	Matching Data Representations	700
14.6		ency and Semantics	
14.0	14.6.1	File Consistency	
	14.0.1 14.6.2	Random Access vs. Sequential Files	
	14.0.2 14.6.3		704
	14.0.3 14.6.4	Progress	
	14.0.4 14.6.5	Nonblocking Collective File Operations	
		•	
	14.6.6	Type Matching	
	14.6.7	Miscellaneous Clarifications	
	14.6.8	MPI_Offset Type	
	14.6.9	Logical vs. Physical File Layout	706
	14.0.10	File Size	707
	14.0.11	Examples	707
14 8		Asynchronous $1/0$	710
14.7		cor Handling	
14.8		or Classes	
14.9	-	les	
		Double Buffering with Split Collective I/O	
	14.9.2	Subarray Filetype Constructor	715
15 Tool	Suppor	•t	717
15.1		ction	717
$15.1 \\ 15.2$		g Interface	717
10.2	15.2.1	Requirements	717
	15.2.1 15.2.2	Discussion	718
	15.2.2 15.2.3	Logic of the Design	718
	15.2.3 15.2.4	Miscellaneous Control of Profiling	710
	15.2.4 15.2.5	MPI Library Implementation	719
	15.2.0 15.2.6	Complications	720 721
	10.2.0		721 721
		Multiple Counting	721
		Linker Oddities	722 722
	1597	Fortran Support Methods	
	15.2.7	Multiple Levels of Interception	722

15.3	The MF	PI Tool Information Interface
	15.3.1	Verbosity Levels
	15.3.2	Binding MPI Tool Information Interface Variables to MPI Objects
	15.3.3	Convention for Returning Strings
	15.3.4	Initialization and Finalization
	15.3.5	Datatype System
	15.3.6	Control Variables
		Control Variable Query Functions
		Handle Allocation and Deallocation
		Control Variable Access Functions
	15.3.7	Performance Variables
	10.0	Performance Variable Classes
		Performance Variable Query Functions
		Performance Experiment Sessions
		Handle Allocation and Deallocation
		Starting and Stopping of Performance Variables
		Performance Variable Access Functions
	15.3.8	
	10.0.0	Events Event Sources
		Callback Safety Requirements
		Event Type Query Functions
		Handle Allocation and Deallocation
		Handling Dropped Events
		Reading Event Data
		Reading Event Meta Data
	15.3.9	Variable Categorization
		Category Query Functions
		Category Member Query Functions
		Return Codes for the MPI Tool Information Interface
	15.3.11	Profiling Interface
		Interfaces
		ated since MPI-2.0
16.2	Depreca	ated since MPI-2.2
16.3	Depreca	ated since MPI-4.0
17 Rem	loved In	terfaces
17.1		ed MPI-1 Bindings
	17.1.1	Overview
	17.1.2	Removed MPI-1 Functions
	17.1.3	Removed MPI-1 Functions Removed MPI-1 Datatypes
	17.1.4	Removed MPI-1 Constants
	17.1.5	Removed MPI-1 Callback Prototypes
17.2		Bindings
11.4		
18 Back		ncompatibilities
18.1	Backwa	rd Incompatibilities Starting in MPI-4.0

9 Lang 19.1	guage B Support	t for Fortran	78 78
13.1	19.1.1	Overview	7
	19.1.1 19.1.2	Fortran Support Through the mpi_f08 Module	7
	19.1.2 19.1.3	Fortran Support Through the mpi_100 Module	
	19.1.0	Fortran Support Through the mpif.h Include File	7
	19.1.4 19.1.5	Interface Specifications, Procedure Names, and the Profiling Interface	
	19.1.6 19.1.6	MPI for Different Fortran Standard Versions	7
	19.1.0 19.1.7	Requirements on Fortran Compilers	7
	19.1.8	Additional Support for Fortran Register-Memory-Synchronization	8
	19.1.9	Additional Support for Fortran Numeric Intrinsic Types	8
	101110	Parameterized Datatypes with Specified Precision and Exponent	Ŭ
		Range	8
		Support for Size-specific MPI Datatypes	8
		Communication With Size-specific Types	
	19.1.10	Problems With Fortran Bindings for MPI	8
		Problems Due to Strong Typing	
		Problems Due to Data Copying and Sequence Association with Sub-	
		script Triplets	8
	19.1.13	Problems Due to Data Copying and Sequence Association with Vec-	
		tor Subscripts	8
	19.1.14	Special Constants	8
		Fortran Derived Types	8
	19.1.16	Optimization Problems, an Overview	8
	19.1.17	Problems with Code Movement and Register Optimization	8
		Nonblocking Operations	8
		Persistent Operations	8
		One-sided Communication	8
		MPI_BOTTOM and Combining Independent Variables in Datatypes	8
		Solutions	8
		The Fortran ASYNCHRONOUS Attribute	8
		Calling MPI_F_SYNC_REG	
		A User Defined Routine Instead of MPI_F_SYNC_REG	8
		Module Variables and COMMON Blocks	8
		The (Poorly Performing) Fortran VOLATILE Attribute	8
		The Fortran TARGET Attribute	8
	19.1.18		8
		Permanent Data Movement	8
10.0		Comparison with C	8
19.2	~ ~	t for Large Count and Large Byte Displacement	8
19.3		ge Interoperability	8
	19.3.1	Introduction	8
	19.3.2	Assumptions	8
	19.3.3	Initialization	8
	19.3.4	Transfer of Handles	8
	19.3.5	Status	8
	19.3.6	MPI Opaque Objects	8
		Datatypes	8

			Callback Functions	840
			Error Handlers	840
			Reduce Operations	840
		19.3.7	Attributes	841
		19.3.8	Extra-State	
		19.3.9	Constants	845
			Interlanguage Communication	846
Α	Lang		indings Summary	849
	A.1	Defined	Values and Handles	849
		A.1.1	Defined Constants	849
		A.1.2	Types	863
		A.1.3	Prototype Definitions	864
			C Bindings	864
			Fortran 2008 Bindings with the mpi_f08 Module	866
			Fortran Bindings with mpif.h or the mpi Module	869
		A.1.4	Deprecated Prototype Definitions	
		A.1.5	Info Keys	871
		A.1.6	Info Values	872
	A.2		ry of the Semantics of all OpRelated Routines	873
	A.3	C Bindi	ings	
		A.3.1	Point-to-Point Communication C Bindings	874
		A.3.2	Partitioned Communication C Bindings	877
		A.3.3	Datatypes C Bindings	878
		A.3.4	Collective Communication C Bindings	881
		A.3.5	Groups, Contexts, Communicators, and Caching C Bindings	889
		A.3.6	Process Topologies C Bindings	892
		A.3.7	MPI Environmental Management C Bindings	896
		A.3.8	The Info Object C Bindings	
		A.3.9	Process Creation and Management C Bindings	898
		A.3.10	One-Sided Communications C Bindings	899
		A.3.11	External Interfaces C Bindings	
			I/O C Bindings	
			Language Bindings C Bindings	907
		A.3.14	Tools / Profiling Interface C Bindings	907
		A.3.15	Tools / MPI Tool Information Interface C Bindings	907
		A.3.16	Deprecated C Bindings	910
	A.4	Fortran	2008 Bindings with the mpi_f08 Module	911
		A.4.1	Point-to-Point Communication Fortran 2008 Bindings	911
		A.4.2	Partitioned Communication Fortran 2008 Bindings	921
		A.4.3	Datatypes Fortran 2008 Bindings	922
		A.4.4	Collective Communication Fortran 2008 Bindings	930
		A.4.5	Groups, Contexts, Communicators, and Caching Fortran 2008 Bind-	
			ings	952
		A.4.6	Process Topologies Fortran 2008 Bindings	959
		A.4.7	MPI Environmental Management Fortran 2008 Bindings	969
		A.4.8	The Info Object Fortran 2008 Bindings	972
		A.4.9	Process Creation and Management Fortran 2008 Bindings	973

	A.4.10	One-Sided Communications Fortran 2008 Bindings	976	
	A.4.11	External Interfaces Fortran 2008 Bindings	984	
	A.4.12	I/O Fortran 2008 Bindings	984	
	A.4.13	Language Bindings Fortran 2008 Bindings	997	
	A.4.14	Tools / Profiling Interface Fortran 2008 Bindings	998	
	A.4.15	Deprecated Fortran 2008 Bindings	998	
A.5	Fortran	Bindings with mpif.h or the mpi Module	999	
	A.5.1	Point-to-Point Communication Fortran Bindings	999	
	A.5.2	Partitioned Communication Fortran Bindings	1002	
	A.5.3	Datatypes Fortran Bindings	1002	
	A.5.4	Collective Communication Fortran Bindings		
	A.5.5	Groups, Contexts, Communicators, and Caching Fortran Bindings	1011	
	A.5.6	Process Topologies Fortran Bindings	1015	
	A.5.7	MPI Environmental Management Fortran Bindings	1019	
	A.5.8	The Info Object Fortran Bindings	1021	
	A.5.9	Process Creation and Management Fortran Bindings	1022	
	A.5.10	One-Sided Communications Fortran Bindings	1023	
	A.5.11	External Interfaces Fortran Bindings	1028	
	A.5.12	I/O Fortran Bindings	1028	
	A.5.13	Language Bindings Fortran Bindings	1033	
	A.5.14	Tools / Profiling Interface Fortran Bindings	1033	
	A.5.15	Deprecated Fortran Bindings	1033	
	nge-Log		1035	
B.1	_	s from Version 3.1 to Version 4.0 \ldots		
	B.1.1	Fixes to Errata in Previous Versions of MPI		
	B.1.2	Changes in MPI-4.0		
B.2	<u> </u>	s from Version 3.0 to Version 3.1		
	B.2.1	Fixes to Errata in Previous Versions of MPI		
	B.2.2	Changes in MPI-3.1		
B.3		s from Version 2.2 to Version 3.0		
	B.3.1	Fixes to Errata in Previous Versions of MPI		
	B.3.2	Changes in MPI-3.0		
B.4		s from Version 2.1 to Version 2.2		
B.5	Change	s from Version 2.0 to Version 2.1	1049	
Diblig	nonhr		1055	
Bibliog	rapny		1055	
Genera	l Index		1061	
Genera	i indon		1001	
Examp	les Inde	x	1067	
MPI C	MPI Constant and Predefined Handle Index 107			
MPI D	MPI Declarations Index 107			
MPI C	allback	Function Prototype Index	1077	
	unction	Index	1078	

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	12
6.1	Collective communications an overview	187
6.2	Collective communications, an overview	190
6.3	Inter-communicator reduce-scatter	191
6.4	Gather example	198
6.5	Gatherv example with strides	199
6.6	Gatherv example, 2-dimensional	200
6.7	Gatherv example, 2-dimensional, subarrays with different sizes	201
6.8	Gatherv example, 2-dimensional, subarrays with different sizes and strides .	203
6.9	Scatter example	208
6.10	•	209
6.11		210
	Race conditions with point-to-point and collective communications	302
6.13	Overlapping Communicators Example	306
7.1	Inter-communicator creation using MPI_COMM_CREATE	329
7.2	Inter-communicator construction with MPI_COMM_SPLIT	333
7.3	Recursive communicator creation with MPI_COMM_SPLIT_TYPE	338
7.4	Three-group pipeline	358
7.5	Three-group ring	359
8.1	Neighborhood gather communication example	415
8.2	Cartesian neighborhood allgather example for 3 and 1 processes in a dimension	
8.3	Cartesian neighborhood alltoall example for 3 and 1 processes in a dimension	
8.4	Set-up of process structure for two-dimensional parallel Poisson solver	442
8.5	Communication routine with local data copying and sparse neighborhood	449
9 G	all-to-all	443
8.6	Communication routine with sparse neighborhood all-to-all-w and without local data copying	444
8.7	Two-dimensional parallel Poisson solver with persistent sparse neighborhood	444
0.1	all-to-all-w and without local data copying	445
		440
11.1	Session handle to communicator	496
11.2	Process set examples	499
10.1		
12.1	Schematic description of the public/private window operations in the	FOC
	MPI_WIN_SEPARATE memory model for two overlapping windows	586

12.2	Active target communication	589
12.3	Active target communication, with weak synchronization	590
12.4	Passive target communication	591
12.5	Active target communication with several processes	595
12.6	Symmetric communication	614
12.7	Deadlock situation	614
12.8	No deadlock	614
14.1	Etypes and filetypes	636
14.2	Partitioning a file among parallel processes	636
14.3	Displacements	649
14.4	Example array file layout	715
14.5	Example local array filetype for process 1	715
	Status conversion routines	

List of Tables

2.1	Deprecated and removed constructs	23
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	157
7.1	MPI_COMM_* Function Behavior (in Inter-Communication Mode)	353
9.1	Error classes (Part 1)	467
9.2	Error classes (Part 2)	468
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	509
12.1	C types of attribute value argument to $MPI_WIN_GET_ATTR$ and	
	MPI_WIN_SET_ATTR	560
12.2	Error classes in one-sided communication routines	604
14.1	Data access routines	651
	"external32" sizes of predefined datatypes	694
	"external32" sizes of optional datatypes	695
	"external32" sizes of C++ datatypes	695
14.5	I/O Error Classes	713
15.1	MPI tool information interface verbosity levels	724
15.2	Constants to identify associations of variables	725
15.3	MPI datatypes that can be used by the MPI tool information interface	728
	Scopes for control variables	732
	Hierarchy of safety requirement levels for event callback routines	752
15.6	List of MPI functions that when called from within a callback function may	
	not return MPI_T_ERR_NOT_ACCESSIBLE	753
15.7	Return codes used in functions of the MPI tool information interface	771
17.1	Removed MPI-1 functions and their replacements	779
17.2	Removed MPI-1 datatypes. The indicated routine may be used for changing	
	the lower and upper bound respectively	780
	Removed MPI-1 constants	780
17.4	Removed MPI-1 callback prototypes and their replacements	780

9.1 Specific Fortran procedure names and related calling conventions799.2 Occurrence of Fortran optimization problems81	

Acknowledgments

6

 $\overline{7}$

8

9 10 11

12

13

14

15

16

17

18 19

20 21

22 23

24

25 26

27

28 29

30 31

32 33

34

35

36 37

38 39

40

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

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

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

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

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

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

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

1	Greg Astfalk	Robert Babb	Ed Benson	Rajesh Bordawekar
2	Pete Bradley	Peter Brennan	Ron Brightwell	Maciej Brodowicz
3	Eric Brunner	Greg Burns	Margaret Cahir	Pang Chen
4	Ying Chen	Albert Cheng	Yong Cho	Joel Clark
5	Lyndon Clarke	Laurie Costello	Dennis Cottel	Jim Cownie
6	Zhenqian Cui	Suresh Damodaran-Kar	nal	Raja Daoud
7	Judith Devaney	David DiNucci	Doug Doefler	Jack Dongarra
8	Terry Dontje	Nathan Doss	Anne Elster	Mark Fallon
9	Karl Feind	Sam Fineberg	Craig Fischberg	Stephen Fleischman
10	Ian Foster	Hubertus Franke	Richard Frost	Al Geist
11	Robert George	David Greenberg	John Hagedorn	Kei Harada
12	Leslie Hart	Shane Hebert	Rolf Hempel	Tom Henderson
13	Alex Ho	Hans-Christian Hoppe	Joefon Jann	Terry Jones
14	Karl Kesselman	Koichi Konishi	Susan Kraus	Steve Kubica
15	Steve Landherr	Mario Lauria	Mark Law	Juan Leon
16	Lloyd Lewins	Ziyang Lu	Bob Madahar	Peter Madams
17	John May	Oliver McBryan	Brian McCandless	Tyce McLarty
18	Thom McMahon	Harish Nag	Nick Nevin	Jarek Nieplocha
19	Ron Oldfield	Peter Ossadnik	Steve Otto	Peter Pacheco
20	Yoonho Park	Perry Partow	Pratap Pattnaik	Elsie Pierce
21	Paul Pierce	Heidi Poxon	Jean-Pierre Prost	Boris Protopopov
22	James Pruyve	Rolf Rabenseifner	Joe Rieken	Peter Rigsbee
23	Tom Robey	Anna Rounbehler	Nobutoshi Sagawa	Arindam Saha
24	Eric Salo	Darren Sanders	Eric Sharakan	Andrew Sherman
25	Fred Shirley	Lance Shuler	A. Gordon Smith	Ian Stockdale
26	David Taylor	Stephen Taylor	Greg Tensa	Rajeev Thakur
27	Marydell Tholburn	Dick Treumann	Simon Tsang	Manuel Ujaldon
28	David Walker	Jerrell Watts	Klaus Wolf	Parkson Wong
29	Dave Wright			
30	2 010 111010			
31	The MPI Forum a	lso acknowledges and ap	preciates the valuable	e input from people via
32	e-mail and in person.			
33	The following inst	titutions supported the M	1PI-2 effort through t	ime and travel support
34	for the people listed a	bove.		
35	Argonne National	Laboratory		
36	Bolt, Beranek, an	ů		
37	California Institu			
38	Center for Compt	0.		
39	Convex Compute	_		
40	Cray Research	Corporation		
41	Digital Equipmen	t Corporation		
42	0	*		
43	Dolphin Interconnect Solutions, Inc. Edinburgh Parallel Computing Centre			
44				
45	General Electric (2 V	motion Tachnolog	
46	Hewlett-Packard	Research Center for Info	mation recimology	
40				
48	Hitachi Hughag Aircraft (Yamanana		
-10	Hughes Aircraft (Jompany		

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

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

Unofficial Draft for Comment Only

 41

1	MPI-1.3 and MPI-2.1:
2 3	The editors and organizers of the combined documents have been:
4	• Richard Graham, Convener and Meeting Chair
5 6	• Jack Dongarra, Steering Committee
7 8	• Al Geist, Steering Committee
9	• William Gropp, Steering Committee
10 11	• Rainer Keller, Merge of MPI-1.3
12 13	• Andrew Lumsdaine, Steering Committee
14 15	• Ewing Lusk, Steering Committee, MPI-1.1-Errata (Oct. 12, 1998) MPI-2.1-Errata Ballots 1, 2 (May 15, 2002)
16 17 18	 Rolf Rabenseifner, Steering Committee, Merge of MPI-2.1 and MPI-2.1-Errata Ballots 3, 4 (2008)
19 20 21	All chapters have been revisited to achieve a consistent MPI-2.1 text. Those who served as authors for the necessary modifications are:
22 23	• William Gropp, Front Matter, Introduction, and Bibliography
23 24	• Richard Graham, Point-to-Point Communication
25 26	Adam Moody, Collective Communication
27	• Richard Treumann, Groups, Contexts, and Communicators
28 29 30	• Jesper Larsson Träff, Process Topologies, Info-Object, and One-Sided Communica- tions
31 32	• George Bosilca, Environmental Management
33	• David Solt, Process Creation and Management
34 35	• Bronis R. de Supinski, External Interfaces, and Profiling
36 37	• Rajeev Thakur, I/O
38	• Jeffrey M. Squyres, Language Bindings and MPI-2.1 Secretary
39 40	• Rolf Rabenseifner, Deprecated Functions and Annex Change-Log
41 42	• Alexander Supalov and Denis Nagorny, Annex Language Bindings
43 44 45 46	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.
47	
48	

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

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

Pavan Balaji	Purushotham V. Bangalore	Brian Barrett	1	
Richard Barrett	Christian Bell	Robert Blackmore	2	
Gil Bloch	Ron Brightwell	Greg Bronevetsky	3	
Jeff Brown	Darius Buntinas	Jonathan Carter	4	
Nathan DeBardeleben	Terry Dontje	Gabor Dozsa	5	
Edric Ellis	Karl Feind	Edgar Gabriel	6	
Patrick Geoffray	Johann George	David Gingold	7	
David Goodell	Erez Haba	Robert Harrison	8	
Thomas Herault	Marc-André Hermanns	Steve Hodson	9	
Joshua Hursey	Yutaka Ishikawa	Bin Jia	10	
Hideyuki Jitsumoto	Terry Jones	Yann Kalemkarian	11	
Ranier Keller	Matthew Koop	Quincey Koziol	12	
Manojkumar Krishnan	Sameer Kumar	Miron Livny	13	
Andrew Lumsdaine	Miao Luo	Ewing Lusk	14	
Timothy I. Mattox	Kannan Narasimhan	Mark Pagel	15	
Avneesh Pant	Steve Poole	Howard Pritchard	16	
Craig Rasmussen	Hubert Ritzdorf	Rob Ross	17	
Martin Schulz	Pavel Shamis	Galen Shipman	18	
Christian Siebert	Anthony Skjellum	Brian Smith	19	
Naoki Sueyasu	Vinod Tipparaju	Keith Underwood	20	
Rolf Vandevaart	Abhinav Vishnu	Weikuan Yu	21	
			22	
The MPI Forum also acknow	vledges and appreciates the va	aluable input from people via	23	
e-mail and in person.			24	
The following institutions su	pported the MPI-2.2 effort thr	rough time and travel support	25	
for the people listed above.			26	
Argonne National Laborato	ry		27	
Auburn University			28	
Bull				
Bull Cisco Systems, Inc.				
Cray Inc.			31	
Forschungszentrum Jülich				
Fujitsu			33	
The HDF Group			34	
Hewlett-Packard				
International Business Machines				
Indiana University				
Institut National de Recherche en Informatique et Automatique (Inria)			38	
Institute for Advanced Science & Engineering Corporation				
Intel Corporation				
Lawrence Berkeley National	Laboratory		41	
Lawrence Livermore National Laboratory				
Los Alamos National Laboratory			43	
Mathworks			44	
Mellanox Technologies 4				
Microsoft 4				
Myricom			47	
NEC Corporation			48	

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

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

lettings of participated in the c-in-	an discussions and who	are not mentioned above.	3
Tatsuya Abe	Tomoya Adachi	Sadaf Alam	4
Reinhold Bader	Pavan Balaji	Purushotham V. Bangalore	5
Brian Barrett	Richard Barrett	Robert Blackmore	6
Aurelien Bouteiller	Ron Brightwell	Greg Bronevetsky	7
Jed Brown	Darius Buntinas	Devendar Bureddy	8
Arno Candel	George Carr	Mohamad Chaarawi	9
Raghunath Raja Chandrasekar	James Dinan	Terry Dontje	10
Edgar Gabriel	Balazs Gerofi	Brice Goglin	11
David Goodell	Manjunath Gorentla	Erez Haba	12
Jeff Hammond	Thomas Herault	Marc-André Hermanns	13
Jennifer Herrett-Skjellum	Nathan Hjelm	Atsushi Hori	14
Joshua Hursey	Marty Itzkowitz	Yutaka Ishikawa	15
Nysal Jan	Bin Jia	Hideyuki Jitsumoto	16
Yann Kalemkarian	Krishna Kandalla	Takahiro Kawashima	17
Chulho Kim	Dries Kimpe	Christof Klausecker	18
Alice Koniges	Quincey Koziol	Dieter Kranzlmueller	19
Manojkumar Krishnan	Sameer Kumar	Eric Lantz	20
Jay Lofstead	Bill Long	Andrew Lumsdaine	21
Miao Luo	Ewing Lusk	Adam Moody	22
Nick M. Maclaren	Amith Mamidala	Guillaume Mercier	23
Scott McMillan	Douglas Miller	Kathryn Mohror	24
Tim Murray	Tomotake Nakamura	Takeshi Nanri	25
Steve Oyanagi	Mark Pagel	Swann Perarnau	26
Sreeram Potluri	Howard Pritchard	Rolf Riesen	27
Hubert Ritzdorf	Kuninobu Sasaki	Timo Schneider	28
Martin Schulz	Gilad Shainer	Christian Siebert	29
Anthony Skjellum	Brian Smith	Marc Snir	30
Raffaele Giuseppe Solca	Shinji Sumimoto	Alexander Supalov	31
Sayantan Sur	Masamichi Takagi	Fabian Tillier	32
Vinod Tipparaju	Jesper Larsson Träff	Richard Treumann	33
Keith Underwood	Rolf Vandevaart	Anh Vo	34
Abhinav Vishnu	Min Xie	Enqiang Zhou	35
			36

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

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

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

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

36

37

38

39

40

41

42

 $1 \\ 2$

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

• Rolf Rabenseifner, Steering Committee, Terms and Definitions, and Fortran Bindings, Deprecated Functions, Annex Change-Log, and Annex Language Bindings				
• Richard L. Graham Ste	• Richard L. Graham, Steering Committee, Meeting Convener			
			4 5	
• Jeffrey M. Squyres, Lang	guage Bindings and MP	I-3.1 Secretary	6	
• Daniel Holmes, Point-to-	Point Communication		7	
• George Bosilca, Datatyp	es and Environmental M	Management	8 9	
• Torsten Hoefler, Collecti	ve Communication and	Process Topologies	10	
,			11	
• Pavan Balaji, Groups, C	ontexts, and Communic	cators, and External Interfaces	12	
• Jeff Hammond, The Info	o Object		13 14	
• David Solt, Process Crea	ation and Management		15	
,	ation and Management		16	
• Quincey Koziol, I/O			17	
• Kathryn Mohror, Tool S	upport		18 19	
• Rajeev Thakur, One-Sid	od Communications		20	
• Rajeev Thakur, One-Sid	eu Communications		21	
The following list includes some of the active participants who attended MPI Forum				
meetings or participated in the e-mail discussions.				
Charles Archer	Pavan Balaji	Purushotham V. Bangalore	24 25	
Brian Barrett	Wesley Bland	Michael Blocksome	26	
George Bosilca	Aurelien Bouteiller	Devendar Bureddy	27	
Yohann Burette	Mohamad Chaarawi	Alexey Cheptsov	28	
James Dinan	Dmitry Durnov	Thomas Francois	29	
Edgar Gabriel	Todd Gamblin	Balazs Gerofi	30	
Paddy Gillies	David Goodell	Manjunath Gorentla Venkata	31	
Richard L. Graham	Ryan E. Grant	William Gropp	32	
Khaled Hamidouche	Jeff Hammond	Amin Hassani	33	
Marc-André Hermanns	Nathan Hjelm	Torsten Hoefler	34	
Daniel Holmes	Atsushi Hori	Yutaka Ishikawa	35	
Hideyuki Jitsumoto	Jithin Jose	Krishna Kandalla	36	
Christos Kavouklis	Takahiro Kawashima	Chulho Kim	37	
Michael Knobloch	Alice Koniges	Quincey Koziol	38	
Sameer Kumar	Joshua Ladd	Ignacio Laguna	39	
Huiwei Lu	Guillaume Mercier	Kathryn Mohror	40	
Adam Moody	Tomotake Nakamura	Takeshi Nanri	41	
Steve Oyanagi	Antonio J. Pena	Sreeram Potluri	42	
Howard Pritchard	Rolf Rabenseifner	Nicholas Radcliffe	43	

Raghunath Raja

Christian Siebert

Kento Sato

David Solt

Craig Rasmussen

Anthony Skjellum

Jeffrey M. Squyres

Martin Schulz

Ken Raffenetti

Sangmin Seo

Brian Smith

Davide Rossetti

xxxiii

44

45

46

47 48

1	Hari Subramoni Sl	ninji Sumimoto	Alexander Supalov	
2	Bronis R. de Supinski – Sa	ayantan Sur	Masamichi Takagi	
3	Keita Teranishi R	ajeev Thakur	Fabian Tillier	
4	Yuichi Tsujita G	eoffroy Vallée	Rolf vandeVaart	
5	-	erome Vienne	Venkat Vishwanath	
6	-	useyin S. Yildiz	Junchao Zhang	
7	Xin Zhao			
8				
9	The MPI Forum also acknowledges	and appreciates t	the valuable input from people via	
10	e-mail and in person.			
11	The following institutions supported	d the MPI-3.1 $efforthere = 10^{-3}$	rt through time and travel support	
12	for the people listed above.			
13	Argonne National Laboratory			
14	Auburn University			
15	Cisco Systems, Inc.			
16	Cray			
17	EPCC, The University of Edinburg	sh		
18	ETH Zurich	511		
19	Forschungszentrum Jülich			
20	Fujitsu			
21	German Research School for Simul	ation Sciences		
22	The HDF Group	ation belefices		
23	International Business Machines			
24	Institut National de Recherche en Informatique et Automatique (Inria)			
25	Intel Corporation		(inita)	
26	Kyushu University	X	*	
27	Lawrence Berkeley National Labora	atory		
28	Lawrence Livermore National Laboratory			
29	Lawrence Livermore National Laboratory Lenovo			
30	Los Alamos National Laboratory			
31	Mellanox Technologies, Inc.			
32	Mellanox Technologies, Inc. Microsoft Corporation			
33	NEC Corporation			
34	NVIDIA Corporation			
35	Oak Ridge National Laboratory			
36	The Ohio State University			
37	RIKEN AICS			
38	Sandia National Laboratories			
39	Texas Advanced Computing Center	r		
40	Tokyo Institute of Technology	L		
41	University of Alabama at Birmingh	nam		
42	University of Houston	10111		
43	University of Illinois at Urbana-Ch	ampaign		
44	University of Oregon	ampagn		
45	University of Stuttgart, High Perfo	rmance Computi	ng Center Stuttgart (HLRS)	
46	University of Tennessee, Knoxville		ng Contor Statugart (IIIIto)	
47	University of Tokyo			
48	Chiverbing of TORyO			

MPI-4.0:

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

Unofficial Draft for Comment Only

xxxv

¹ **Tools** Kathryn Mohror and Marc-André Hermanns

 $\mathbf{2}$

3

4

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

•	0 1	1	
5		Julien Adam	Abdelhalim Amer
6		Charles Archer	Ammar Ahmad Awan
7		Pavan Balaji	Marc Gamell Balmana
8		Purushotham V. Bangalore	Mohammadreza Bayatpour
9		Jean-Baptiste Besnard	Claudia Blaas-Schenner
10		Wesley Bland	Gil Bloch
11		George Bosilca	Aurelien Bouteiller
12		Ben Bratu	Alexander Calvert
13		Nicholas Chaimov	Sourav Chakraborty
14		Steffen Christgau	Ching-Hsiang Chu
15		Mikhail Chuvelev	James Clark
16		Carsten Clauss	Isaias Alberto Compres Urena
17		Giuseppe Congiu	Brandon Cook
18		James Custer	Anna Daly
19		Hoang-Vu Dang	James Dinan
20		Matthew Dosanjh	Murali Emani
21		Christian Engelmann	Noah Evans
22		Ana Gainaru	Esthela Gallardo
23		Balazs Gerofi	Salvatore Di Girolamo
24		Brice Goglin	Manjunath Gorentla Venkata
25		Richard Graham	Ryan E. Grant
26		Stanley Graves	William Gropp
27		Siegmar Gross	Taylor Groves
28 29		Yanfei Guo	Khaled Hamidouche
29 30		Jeff Hammond	Marc-André Hermanns
31		Nathan Hjelm	Torsten Hoefler
32		Daniel Holmes	Atsushi Hori
33		Josh Hursey	Ilya Ivanov
34		Julien Jaeger	Emmanuel Jeannot
35		Sylvain Jeaugey	Jithin Jose
36		Krishna Kandalla	Takahiro Kawashima
37		Chulho Kim	Michael Knobloch
38		Alice Koniges	Sameer Kumar
39		Kim Kyunghun	Ignacio Laguna Peralta
40		Stefan Lankes	Tonglin Li
41		Xioyi Lu	Kavitha Madhu
42		Alexey Malhanov	Ryan Marshall
43		William Marts	Guillaume Mercier
44		Ali Mohammed	Kathryn Mohror
45		Takeshi Nanri	Thomas Naughton
46		Christoph Niethammer	Takafumi Nose
47			
48			

Le	ena Oden	Steve Oyanagi	Guillaume Papauré	1
Iv	ry Peng	Antonio Peña	Simon Pickartz	2
A	rtem Polyakov	Sreeram Potluri	Howard Pritchard	3
Μ	lartina Prugger	Marc Pérache	Rolf Rabenseifner	4
N	icholas Radcliffe	Ken Raffenetti	Craig Rasmussen	5
Se	oren Rasmussen	Hubert Ritzdorf	Sergio Rivas-Gomez	6
	avide Rossetti	Martin Ruefenacht	Amit Ruhela	7
	hit Schonbein	Joseph Schuchart	Martin Schulz	8
	angmin Seo	Sameh Sharkawi	Sameer Shende	9
	lin Si	Anthony Skjellum	Brian Smith	10
	avid Solt	Jeffrey M. Squyres	Srinivas Sridharan	11
	ari Subramoni	Nawrin Sultana	Shinji Sumimoto	12
			5	
	ayantan Sur	Hugo Taboada	Keita Teranishi	13
	ajeev Thakur	Keith Underwood	Geoffroy Vallee	14
	kshay Venkatesh	Jerome Vienne	Anh-Vo	15
	ıstin Wozniak	Junchao Zhang	Dong Zhong	16
H	ui Zhou			17
				18
	1 1 1	1 1 .		19
		edges and appreciates	the valuable input from people via	20
e-mail and in pe				21
		ported the MPI-4.0 eff	fort through time and travel support	22
for the people lis	sted above.			23
ATOS				24
Argonne Na	ational Laboratory			25
Arm	J		*	26
Auburn Uni	iversity			27
	Supercomputing Co	enter		28
CEA				29
Cisco System	ms Inc			30
Cray Inc.				31
0	University Of Edi	nhurgh		32
ETH Zürich	v	mourgi		33
Fujitsu	1			34
	ersity of Applied S	aionaoa		35
	· · · ·	Simulation Sciences		36
		Simulation Sciences		37
	ckard Enterprise			38
	al Business Machir			39
		e en Informatique et	Automatique (Inria)	
Intel Corpo				40
		r, Forschungszentrum	h Jülich	41
-	Institute Of Tech	nology		42
Kyushu Uni	-			43
	erkeley National L	-		44
	ivermore National	Laboratory		45
Lenovo				46
Los Alamos	National Laborat	ory		47
Mellanox Te	echnologies, Inc.			48

xxxvii

1	Microsoft Corporation
2	NEC Corporation
3	NVIDIA Corporation
4	Oak Ridge National Laboratory
5	PAR-TEC
6	Paratools, Inc.
7	RIKEN AICS (R-CCS as of 2017)
8	RWTH Aachen University
9	Rutgers University
10	Sandia National Laboratories
11	Silicon Graphics, Inc.
12	Technical University of Munich
13	The HDF Group
14	The Ohio State University
15	Texas Advanced Computing Center
16	Tokyo Institute of Technology
17	University of Alabama at Birmingham
18	University of Basel, Switzerland
19	University of Houston
20	University of Illinois at Urbana-Champaign and the National Center for Supercomput-
21	ing Applications
22	University of Innsbruck
23	University of Oregon
24	University of Potsdam
25	University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)
26	University of Tennessee, Chattanooga
27	University of Tennessee, Knoxville
28	University of Texas at El Paso
29	University of Tokyo
30	VSC Research Center, TU Wien
31	
32	
33	
34	
35	
36	
37	
38	
39	
40	*
41	
42	
43	
44	
45	
46	
47	
48	

Chapter 1

Introduction to MPI

1.1 Overview and Goals

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

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

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

A complete list of goals follows.

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

Unofficial Draft for Comment Only

 24

 31

 $45 \\ 46$

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

1.2 Background of MPI-1.0

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

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

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

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

- 44
- $45 \\ 46$

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

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

Unofficial Draft for Comment Only

1

2

3

4

5 6

> 7 8

9 10

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

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

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

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

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

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

1.4 Background of MPI-1.3 and MPI-2.1

After the release of MPI-2.0, the MPI Forum kept working on errata and clarifications for42both standard documents (MPI-1.1 and MPI-2.0). The short document "Errata for MPI-1.1"43was released October 12, 1998. On July 5, 2001, a first ballot of errata and clarifications for44MPI-2.0 was released, and a second ballot was voted on May 22, 2002. Both votes were done45electronically. Both ballots were combined into one document: "Errata for MPI-2," May4615, 2002. This errata process was then interrupted, but the Forum and its e-mail reflectors47kept working on new requests for clarification.48

Unofficial Draft for Comment Only

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

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

1516

1718

19

20

2122

23

 24 25

26

2728

29

30 31

32 33

34

35

36

37

Background of MPI-2.2 1.5

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

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

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

1.6Background of MPI-3.0

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

38 39 40

41

Background of MPI-3.1 1.7

MPI-3.1 is a minor update to the MPI standard. Most of the updates are corrections 42and clarifications to the standard, especially for the Fortran bindings. New functions added 43 include routines to manipulate MPI_Aint values in a portable manner, nonblocking collective 44 I/O routines, and routines to get the index value by name for MPI_T performance and 45control variables. A general index was also added. 46

- 47
- 48

1.8 Background of 2020 Draft Specification

The 2020 draft specification is expected to become the MPI-4.0 specification once all features have been merged. MPI-4.0 is a major update to the MPI standard. This update includes a number of new features which will be present in the final MPI-4.0 document. The largest changes are the addition of large-count versions of many routines to address the limitations of using an int or INTEGER for the count parameter, persistent collectives, partitioned communications, an alternative way to initialize MPI, application info assertions, and improvements to the definitions of error handling. In addition, there are a number of smaller improvements and corrections.

1.9 Who Should Use This Standard?

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

1.10 What Platforms Are Targets for Implementation?

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

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

1.11 What Is Included in the Standard?

The standard includes:

- Point-to-point communication,
- Datatypes,

1 2

3

4

5

6

7

8

9

10 11 12

13 14

15

16

17

18

19

20

21 22

23 24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41 42 43

44 45

46 47

1	Collective operations,
2 3	Process groups,
4 5	Communication contexts,
	Process topologies,
7 8	Environmental management and inquiry,
9 10	The Info object,
11	Process creation and management,
12 13	One-sided communication,
14 15	External interfaces,
15 16	Parallel file I/O,
17 18	Language bindings for Fortran and C,
19 20	Tool support.
25 26 27 28 29 30 31 32 33 34 34 35 exter	 2 What Is Not Included in the Standard? standard does not specify: Operations that require more operating system support than is currently standard; for example, interrupt-driven receives, remote execution, or active messages, Program construction tools, Debugging facilities. There are many features that have been considered and not included in this standard. happened for a number of reasons, one of which is the time constraint that was self-osed in finishing the standard. Features that are not included can always be offered as nsions by specific implementations. Perhaps future versions of MPI will address some nese issues.
³⁸ 1.13	3 Organization of This Document
⁴⁰ The	following is a list of the remaining chapters in this document, along with a brief ription of each.
44	Chapter 2, MPI Terms and Conventions, explains notational terms and conventions used throughout the MPI document.
45 46 47 48	Chapter 3, Point-to-Point Communication, defines the basic, pairwise communication subset of MPI. <i>Send</i> and <i>receive</i> are found here, along with many associated functions designed to make basic communication powerful and efficient.

Unofficial Draft for Comment Only

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

Unofficial Draft for Comment Only

 $44 \\ 45$

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

Chapter 2

MPI Terms and Conventions

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

2.1 Document Notation

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

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

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

2.2 Naming Conventions

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

- 1. In C 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.

Unofficial Draft for Comment Only

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.

Procedure Specification 2.3

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.
- 1617 18

19

26

27

28 29

33

34

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

20There is one special case—if an argument is a handle to an opaque object (these terms 21are defined in Section 2.5.1), and the object is updated by the procedure call, then the 22argument is marked INOUT or OUT. It is marked this way even though the handle itself is 23not modified—we use the INOUT or OUT attribute to denote that what the handle *references* 24 is updated. 25

> 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 30 is to be used, but does not provide a rigorous classification that can be translated directly 31 into all language bindings (e.g., INTENT in Fortran 90 bindings or const in C bindings). 32 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 pro-35 cesses and OUT by other processes. Such an argument is, syntactically, an INOUT argument 36 and is marked as such, although, semantically, it is not used in one call both for input and 37 for output on a single process. 38

Another frequent situation arises when an argument value is needed only by a subset 39 of the processes. When an argument is not significant at a process then an arbitrary value 40 can be passed as an argument. 41

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

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

1

2

3

4

5

6

7

8 9 10

11 12

1314

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

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

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

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

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

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

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

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

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

2.4 Semantic Terms

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

2.4.1 MPI Operations

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

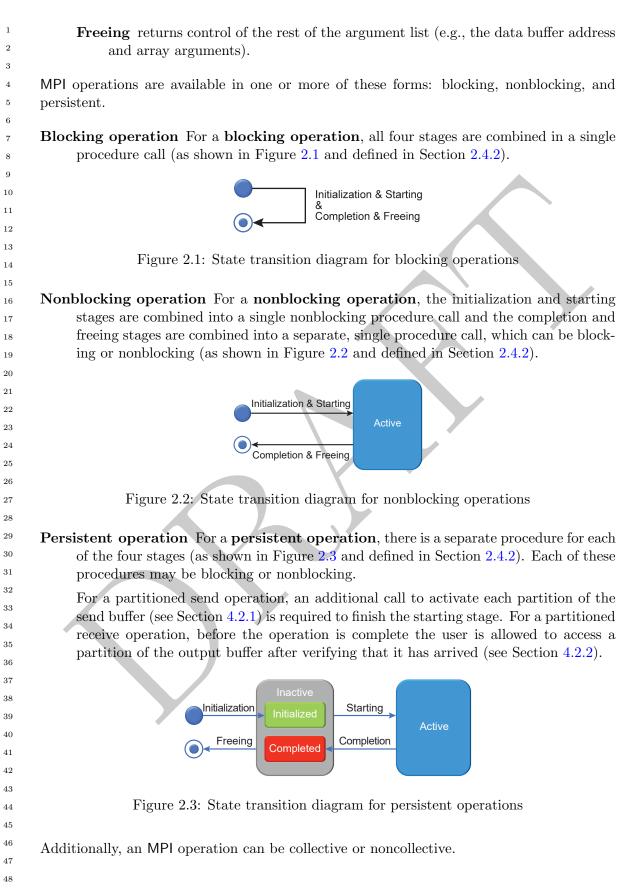
Note that **initiation** refers to the combination of the initialization and starting stages.

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

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

Unofficial Draft for Comment Only

 $\frac{4}{5}$



CHAPTER 2. MPI TERMS AND CONVENTIONS

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 non-local—defined as follows:

Non-local procedure An MPI procedure is non-local if returning may require, during its execution, some specific semantically-related MPI procedure to be called on another MPI process.

Local procedure An MPI procedure is local if it is not non-local.

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.

Unofficial Draft for Comment Only

 $\mathbf{2}$

 24

 31

 $41 \\ 42$

1 2 3	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.
4 5 6 7 8 9 10	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.
11 12 13	Incomplete procedure An MPI procedure is called incomplete if it is not a completing procedure.
14 15 16 17 18 19	 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.
20 21	Blocking procedure An MPI procedure is blocking if it is not nonblocking.
22 23 24 25 26	Advice to users. Note that for operation-related MPI procedures, in most cases incomplete procedures are local and completing procedures are non-local. Exceptions are noted where such procedures are defined. In many cases an additional prefix letter I as an abbreviation of the words incomplete and immediate marks nonblocking procedures in the procedure name.
27	Some categorization examples are listed below.
28 29	Nonblocking procedures:
30 31 32	• incomplete and local: MPI_ISEND, MPI_IRECV, MPI_IBCAST, MPI_IMPROBE, MPI_SEND_INIT, MPI_RECV_INIT,
33	Blocking procedures:
34	• completing and non-local: MPI_SEND, MPI_RECV, MPI_BCAST,
35 36	• incomplete and non-local: MPI_MPROBE, MPI_BCAST_INIT,,
37	MPI_FILE_{READ WRITE}_{AT_ALL ALL ORDERED}_BEGIN.
38	• completing and local: MPI_BSEND, MPI_RSEND, MPI_MRECV.
39 40	MPI procedures that are not MPI operation-related:
41	• MPI_COMM_RANK, MPI_WTIME, MPI_PROBE, MPI_IPROBE,
42 43	(End of advice to users.)
44	(Ena of autoce to asers.)
45 46	Collective procedure An MPI procedure is collective if all processes in a group or groups of MPI processes need to invoke the procedure.
47 48	Initialization procedures of collective operations over the same process group must be executed in the same order by all members of the process group.

Unofficial Draft for Comment Only

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

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

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

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

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

2.4.3 MPI Datatypes

For datatypes, the following terms are defined:

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

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

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

Unofficial Draft for Comment Only

1 2

3

4

5

6

 $\overline{7}$

8

9 10

11

12

13

14

15

15 16 17

18

19 20

21

22

23

24

25 26

27

45

46

2.5 Datatypes

2.5.1 Opaque Objects

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

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

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

INTEGER :: MPI_VAL

END TYPE MPI_Comm

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

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

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

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

44 45

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

2 3 4

5

6

7

8

9

10

20

21

22

23 24

25

26

27 28

29

30

31

32

33

34

35

36

37

38 39

40

41

42

43

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

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

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

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

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

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

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

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

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

Unofficial Draft for Comment Only

1 2

3

4

 $\mathbf{5}$

6 7

8 9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24 25

26

27 28

29

30

31

32

33

34

35

36

37

38

2.5.2 Array Arguments

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

11 1213

1

2

3

4

5

6

7

8

9

2.5.3 State

14MPI procedures use at various places arguments with state types. The values of such a 15datatype are all identified by names, and no operation is defined on them. For example, 16the MPI_TYPE_CREATE_SUBARRAY routine has a state argument order with values 17MPI_ORDER_C and MPI_ORDER_FORTRAN.

18 19

20

29

2.5.4 Named Constants

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

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

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

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

- 42MPI_MAX_PROCESSOR_NAME
- MPI_MAX_LIBRARY_VERSION_STRING 43
- MPI_MAX_ERROR_STRING 44
- MPI_MAX_DATAREP_STRING 45
- MPI_MAX_INFO_KEY 46
- 47 MPI_MAX_INFO_VAL
- MPI_MAX_OBJECT_NAME 48

MPI_MAX_PORT_NAME
MPI_VERSION
MPI_SUBVERSION
MPI_F_STATUS_SIZE (C only)
MPI_STATUS_SIZE (Fortran only)
MPI_ADDRESS_KIND (Fortran only)
MPI_COUNT_KIND (Fortran only)
MPI_INTEGER_KIND (Fortran only)
MPI_OFFSET_KIND (Fortran only)
MPI_SUBARRAYS_SUPPORTED (Fortran only)
MPI_ASYNC_PROTECTS_NONBLOCKING (Fortran only)
The constants that cannot be used in initialization expressions or assignments in Fortran
are as follows:
MPI_BOTTOM
MPI_STATUS_IGNORE
MPI_STATUSES_IGNORE
MPI_ERRCODES_IGNORE
MPI_IN_PLACE

MPI_IN_PLACE MPI_ARGV_NULL MPI_ARGVS_NULL MPI_UNWEIGHTED MPI_WEIGHTS_EMPTY

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

2.5.5 Choice

MPI functions sometimes use arguments with a *choice* (or union) data type. Distinct calls to the same routine may pass by reference actual arguments of different types. The mechanism for providing such arguments will differ from language to language. For Fortran with the include file mpif.h or the mpi module, the document uses <type> to represent a choice variable; with the Fortran mpi_f08 module, such arguments are declared with the Fortran 2008 + 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 non-standard compiler-dependent method that has the quality of the call mechanism in the implicit Fortran interfaces, or with the method defined for the mpi_f08 module. See details in Section 19.1.1. (End of advice to implementors.)

 24

 31

2.5.6 Absolute Addresses and Relative Address Displacements

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

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

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

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

2.5.8 Counts

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

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

1

2

14

15

16

17

18

19

20

21

22

23 24

25

33

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

Rationale. Count values need to be large enough to encode any value used for expressing element counts, strides, offsets, indexes, displacements, typemaps in memory, typemaps in file views, etc. Prior to MPI-4.0, many MPI routines used int in C and INTEGER in Fortran as the type for *count* arguments. To avoid breaking backward compatibility, this version of the standard continues to support int in C as well as INTEGER in Fortran in such routines. In addition, this version of the standard supports using MPI_Count in C (via separate "_c"suffixed procedures) as well as INTEGER(KIND=MPI_COUNT_KIND) in Fortran (via polymorphic interfaces in newer MPI Fortran bindings (USE mpi_f08)) in such routines. See Section 19.2 for a full explanation. (*End of rationale.*)

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

There are cases where MPI_UNDEFINED can be returned in a **large count** OUT parameter. Per Table A.1.1 (page 851), 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

Unofficial Draft for Comment Only

 $\mathbf{5}$

 $\overline{7}$

 24

 31

 $46 \\ 47$

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

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

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

15 16

17

36

37

38

39

40

41

42

43 44

45

2.6.2 Fortran Binding Issues

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

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

All MPI Fortran subroutines have a return code in the last argument. With USE mpi_f08, this last argument is declared as OPTIONAL, except for user-defined callback functions (e.g., COMM_COPY_ATTR_FUNCTION) and their predefined callbacks (e.g.,

MPI_NULL_COPY_FN). A few MPI operations which are functions do not have the return code argument. The return code value for successful completion is MPI_SUCCESS. Other error codes are implementation dependent; see the error codes in Chapter 9 and Annex A.

³⁴ Constants representing the maximum length of a string are one smaller in Fortran than
 ³⁵ 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).

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

Unofficial Draft for Comment Only

2.6. LANGUAGE BINDING

Deprecated or removed	deprecated	removed	Replacement
construct	since	since	
MPI_ADDRESS	MPI-2.0	MPI-3.0	MPI_GET_ADDRESS
MPI_TYPE_HINDEXED	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_HINDEXED
MPI_TYPE_HVECTOR	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_HVECTOR
MPI_TYPE_STRUCT	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_STRUCT
MPI_TYPE_EXTENT	MPI-2.0	MPI-3.0	MPI_TYPE_GET_EXTENT
MPI_TYPE_UB	MPI-2.0	MPI-3.0	MPI_TYPE_GET_EXTENT
MPI_TYPE_LB	MPI-2.0	MPI-3.0	MPI_TYPE_GET_EXTENT
MPI_LB ¹	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_RESIZED
MPI_UB ¹	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_RESIZED
MPI_ERRHANDLER_CREATE	MPI-2.0	MPI-3.0	MPI_COMM_CREATE_ERRHANDLER
MPI_ERRHANDLER_GET	MPI-2.0	MPI-3.0	MPI_COMM_GET_ERRHANDLER
MPI_ERRHANDLER_SET	MPI-2.0	MPI-3.0	MPI_COMM_SET_ERRHANDLER
MPI_Handler_function ²	MPI-2.0	MPI-3.0	MPI_Comm_errhandler_function ²
MPI_KEYVAL_CREATE	MPI-2.0		MPI_COMM_CREATE_KEYVAL
MPI_KEYVAL_FREE	MPI-2.0		MPI_COMM_FREE_KEYVAL
MPI_DUP_FN ³	MPI-2.0		MPI_COMM_DUP_FN ³
MPI_NULL_COPY_FN ³	MPI-2.0		MPI_COMM_NULL_COPY_FN ³
MPI_NULL_DELETE_FN ³	MPI-2.0		MPI_COMM_NULL_DELETE_FN ³
MPI_Copy_function ²	MPI-2.0		MPI_Comm_copy_attr_function ²
COPY_FUNCTION ³	MPI-2.0		COMM_COPY_ATTR_FUNCTION ³
$MPI_Delete_function^2$	MPI-2.0		MPI_Comm_delete_attr_function ²
DELETE_FUNCTION ³	MPI-2.0		COMM_DELETE_ATTR_FUNCTION ³
MPI_ATTR_DELETE	MPI-2.0		MPI_COMM_DELETE_ATTR
MPI_ATTR_GET	MPI-2.0		MPI_COMM_GET_ATTR
MPI_ATTR_PUT	MPI-2.0		MPI_COMM_SET_ATTR
MPI_COMBINER_HVECTOR_INTEGER ⁴	-	MPI-3.0	MPI_COMBINER_HVECTOR ⁴
MPI_COMBINER_HINDEXED_INTEGER ⁴	-	MPI-3.0	MPI_COMBINER_HINDEXED ⁴
MPI_COMBINER_STRUCT_INTEGER ⁴	-	MPI-3.0	MPI_COMBINER_STRUCT ⁴
MPI:	MPI-2.2	MPI-3.0	C language binding
MPI_CANCEL for send requests	MPI-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() return	ns the size in l	bits instead	of bytes: see Section 16.3.
Other entries are regular MPI routines.			
e their entities are regular init i foutilites.			

Table 2.1: Deprecated and removed constructs

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.

Unofficial Draft for Comment Only

Type declarations are provided for handles to each category of opaque objects.

² Array arguments are indexed from zero.

- ³ Logical flags are integers with value 0 meaning "false" and a non-zero value meaning
 ⁴ "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.

12 13 14

15 16

17

18 19

20 21

1

5

6 7

8

9

10

11

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 26calls are used. The interaction of an MPI program with other possible means of commu-27nication, I/O, and process management is not specified. Unless otherwise stated in the 28specification of the standard, MPI places no requirements on the result of its interaction 29 with external mechanisms that provide similar or equivalent functionality. This includes, 30 but is not limited to, interactions with external mechanisms for process control, shared and 31 remote memory access, file system access and control, interprocess communication, process 32 signaling, and terminal I/O. High quality implementations should strive to make the results 33 of such interactions intuitive to users, and attempt to document restrictions where deemed 34necessary. 35

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.)

39 40

36

37

38

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

41 42 43

44

2.8 Error Handling

MPI provides the user with reliable message transmission. A message sent is always re ceived correctly, and the user does not need to check for transmission errors, time-outs,
 or other error conditions. In other words, MPI does not provide mechanisms for dealing
 with transmission failures in the communication system. If the MPI implementation is

Unofficial Draft for Comment Only

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

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

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

17 In C and Fortran, almost all MPI calls return a code that indicates successful completion 18 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 1920causes the parallel computation to abort, except for file operations. However, MPI provides 21mechanisms for users to change this default and to handle recoverable errors. The user may specify that no error is fatal, and handle error codes returned by MPI calls by himself or 22herself. Also, the user may provide his or her own error-handling routines, which will be 23invoked whenever an MPI call returns abnormally. The MPI error handling facilities are 24 25described 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 MPI_COMM_SELF is not initialized (i.e., before MPI_INIT / MPI_INIT_THREAD or after MPI_FINALIZE) the error raises the **initial error handler** (set during the launch operation, see 11.8.4).

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 ⁴⁵ has occurred. The desired behavior is that a relevant error code be returned, and the effect ⁴⁶ of the error be localized to the greatest possible extent. E.g., it is highly desirable that an ⁴⁷ erroneous receive call will not cause any part of the receiver's memory to be overwritten, ⁴⁸

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6 7

8 9

10

11

12

13

14 15

16

26

27 28

29 30

31

32 33

34

35

36 37

38 39

40 41

42

¹ beyond the area specified for receiving the message.

² Implementations may go beyond this document in supporting in a meaningful manner ³ MPI calls that are defined here to be erroneous. For example, MPI specifies strict type ⁴ matching rules between matching send and receive operations: it is erroneous to send a ⁵ floating point variable and receive an integer. Implementations may go beyond these type ⁶ matching rules, and provide automatic type conversion in such situations. It will be helpful ⁷ to generate warnings for such non-conforming behavior.

8 9

10

11 12

13

14

15

16

17 18

19

28

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

2.9 Implementation Issues

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

2.9.1 Independence of Basic Runtime Routines

MPI programs require that library routines that are part of the basic language environment
 (such as write in Fortran and printf and malloc in ISO C) and are executed after
 MPI_INIT and before MPI_FINALIZE operate independently and that their completion is

²³ independent of the action of other processes in an MPI program.

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

```
<sup>29</sup> int rank;
```

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

```
31 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

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

```
33 MPI_Finalize();
```

³⁴ The corresponding Fortran programs are also expected to complete.

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

40 MPI_Comm_rank(MPI_COMM_WORLD, &rank);

41 printf("Output from task rank %d\n", rank);

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

46

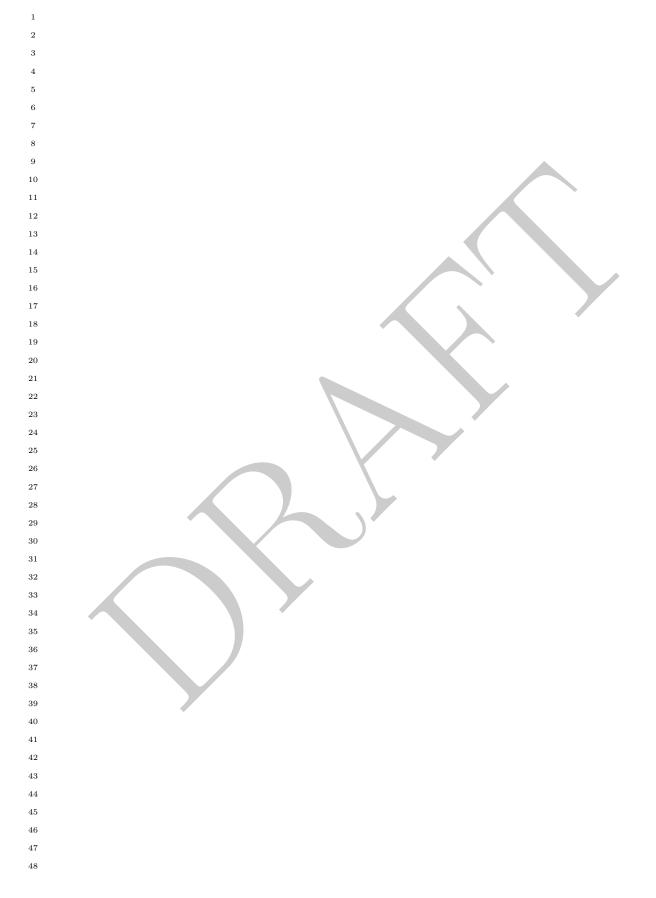
- 47
- 48

2.9.2 Interaction with Signals

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

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

2.10 Examples

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

Chapter 3

Point-to-Point Communication

Introduction 3.1

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"
                                                                                   22
int main(int argc, char *argv[])
                                                                                   23
{
  char message[20];
  int myrank;
  MPI_Status status;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
                                                                                   29
  if (myrank == 0)
                       /* code for process zero */
                                                                                   30
  ſ
      strcpy(message,"Hello, there");
      MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
                                                                                   33
  }
                                                                                   34
  else if (myrank == 1) /* code for process one */
                                                                                   35
  {
                                                                                   36
      MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
                                                                                   37
      printf("received :%s:\n", message);
  }
 MPI_Finalize();
  return 0;
}
                                                                                   42
```

In Example 3.1, process zero (myrank = 0) sends a message to process one using the 44send operation MPI_SEND. The operation specifies a send buffer in the sender memory 45from which the message data is taken. In the example above, the send buffer consists of 46the storage containing the variable **message** in the memory of process zero. The location, 47size and type of the send buffer are specified by the first three parameters of the send 48

1 $\mathbf{2}$ 3

5

6

1516

17

18

19

2021

> 24 25

> 26

27

28

31

32

38

39

40 41

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

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

19 20

21 22

23

24 25 26

27

39

40 41

42

43

3.2 Blocking Send and Receive Operations

3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

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

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

44 Fortran 2008 binding

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

Unofficial Draft for Comment Only

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

<type> BUF(*)

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

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

3.2.2 Message Data

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

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

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

Table 3.1: Predefined MPI datatypes corresponding to Fortran datatypes

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

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

 24

CHAPTER 3. POINT-TO-POINT COMMUNICATION

		1
1	MPI datatype	C datatype
2	MPI_CHAR	char
3		(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
2		(treated as integral value)
3	MPI_UNSIGNED_SHORT	unsigned short int
1	MPI_UNSIGNED	unsigned int
5	MPI_UNSIGNED_LONG	unsigned long int
3	MPI_UNSIGNED_LONG_LONG	unsigned long long int
7	MPI_FLOAT	float
3	MPI_DOUBLE	double
)	MPI_LONG_DOUBLE	long double
)	MPI_WCHAR	wchar_t
L		(defined in <stddef.h>)</stddef.h>
2		(treated as printable character)
3	MPI_C_BOOL	_Bool
Ļ	MPI_INT8_T	int8_t
5	MPI_INT16_T	int16_t
6	MPI_INT32_T	int32_t
7	MPI_INT64_T	int64_t
3	MPI_UINT8_T	uint8_t
)	MPI_UINT16_T	uint16_t
)	MPI_UINT32_T	uint32_t
	MPI_UINT64_T	uint64_t
2	MPI_C_COMPLEX	float _Complex
3	MPI_C_FLOAT_COMPLEX (as a synonym)	float _Complex
1	MPI_C_DOUBLE_COMPLEX	double _Complex
5	MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
3	MPI_BYTE	
7	 MPI_PACKED	
3		1
)		
)	Table $\overline{3.2}$: Predefined MPI datatypes of	corresponding to C datatypes

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.

⁴⁴ MPI requires support of these datatypes, which match the basic datatypes of Fortran ⁴⁵ and ISO C. Additional MPI datatypes should be provided if the host language has additional ⁴⁶ datatypes¹: MPI_DOUBLE_COMPLEX for double precision complex in Fortran declared to

47 48

 $^1\mathrm{These}$ types, such as DOUBLE COMPLEX and INTEGER*4, are not specified by any Fortran standard

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

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

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 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 838 and 846 for information on interlanguage communication with these types.

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

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

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

3.2.3 Message Envelope

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

but are extensions commonly accepted by Fortran compilers.

Unofficial Draft for Comment Only

1	source
2	destination
3	tag
4	communicator
5	
6	The message source is implicitly determined by the identity of the message sender. The
7	other fields are specified by arguments in the send operation.
8	The message destination is specified by the dest argument.
9	The integer-valued message tag is specified by the tag argument. This integer can be
10	used by the program to distinguish different types of messages. The range of valid tag
11	values is $0, \ldots, UB$, where the value of UB is implementation dependent. It can be found by
12	querying the value of the attribute MPI_TAG_UB, as described in Chapter 9. MPI requires
13	that UB be no less than 32767.
14	The comm argument specifies the communicator that is used for the send operation.
15	Communicators are explained in Chapter 7; below is a brief summary of their usage.
16	A communicator specifies the communication context for a communication operation.
17	Each communication context provides a separate "communication universe": messages are
18	always received within the context they were sent, and messages sent in different contexts do not interfere.
19	The communicator also specifies the set of processes that share this communication
20	context. This process group is ordered and processes are identified by their rank within
21	this group. Thus, the range of valid values for dest is $0, \ldots, n-1 \cup \{MPI_PROC_NULL\}$, where
22 23	n is the number of processes in the group. (If the communicator is an inter-communicator,
23 24	then destinations are identified by their rank in the remote group. See Chapter 7.)
24 25	When using the World Model (see Section 11.1), a predefined communicator
26	MPI_COMM_WORLD is provided by MPI. It allows communication with all processes that
27	are accessible after MPI initialization and processes are identified by their rank in the group
28	of MPI_COMM_WORLD.
29	
30	Advice to users. Users that are comfortable with the notion of a flat name space
31	for processes, and a single communication context, as offered by most existing com-
32	munication libraries, need only use the World Model for MPI initialization, and the
33	predefined variable MPI_COMM_WORLD as the comm argument. This will allow com-
34	munication with all the processes available at initialization time.
35	Users may define new communicators, as explained in Chapter 7. Communicators
36	provide an important encapsulation mechanism for libraries and modules. They allow
37	modules to have their own disjoint communication universe and their own process
38	numbering scheme. (End of advice to users.)
39	
40	Advice to implementors. The message envelope would normally be encoded by a
41	fixed-length message header. However, the actual encoding is implementation depen-
42	dent. Some of the information (e.g., source or destination) may be implicit, and need
43	not be explicitly carried by messages. Also, processes may be identified by relative
44	ranks, or absolute ids, etc. (End of advice to implementors.)
45	2.2.4 Placking Proving
46 47	3.2.4 Blocking Receive

 $\frac{47}{48}$ The syntax of the blocking receive operation is given below.

MPI_R	ECV(buf, count, datatyp	pe, source, tag, comm, status)	1
OUT	buf	initial address of receive buffer (choice)	2
IN	count	number of elements in receive buffer (non-negative integer)	3 4 5
IN	datatype	datatype of each receive buffer element (handle)	6
IN	source	rank of source or MPI_ANY_SOURCE (integer)	7
IN	tag	message tag or MPI_ANY_TAG (integer)	8 9
IN	comm	communicator (handle)	9 10
OUT		status object (status)	11
001	Status	status object (status)	12
C bin	ding		13
	0	nt count, MPI_Datatype datatype, int source,	14
		Comm comm, MPI_Status *status)	15 16
int MI	PT Recy c(void *buf.	MPI_Count count, MPI_Datatype datatype,	10
		int tag, MPI_Comm comm, MPI_Status *status)	18
Tranta			19
	an 2008 binding	type, source, tag, comm, status, ierror)	20
	PE(*), DIMENSION(21
		: count, source, tag	22
		NTENT(IN) :: datatype	23
TY	PE(MPI_Comm), INTEN	T(IN) :: comm	24 25
	<pre>/PE(MPI_Status) :: s</pre>		26
II	NTEGER, OPTIONAL, IN	TENT(OUT) :: ierror	27
MPI_Re	ecv(buf, count, data	type, source, tag, comm, status, ierror)	28
	<pre>YPE(*), DIMENSION(</pre>		29
II	NTEGER(KIND=MPI_COUN	T_KIND), INTENT(IN) :: count	30
		NTENT(IN) :: datatype	31
	NTEGER, INTENT(IN) :	5	32
	PE(MPI_Comm), INTEN		33 34
	YPE(MPI_Status) :: s NTEGER, OPTIONAL, IN		34
11	NIEGER, OPIIONAL, IN	IENI(UUI) :: IEIIOF	36
	an binding		37
		TYPE, SOURCE, TAG, COMM, STATUS, IERROR)	38
	TECED COUNT DATATY		39
11	IERROR	PE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),	40
	IERUUR		41
T	he blocking semantics of	f this call are described in Section 3.4.	42

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.

Unofficial Draft for Comment Only

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

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

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

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

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

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

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

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

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

45 46 3.2.5 Return Status

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

Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8

9

10

11

12

13

34

35

36

37

38

39

40

 $41 \\ 42$

43

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

In C, status is a structure that contains three fields named MPI_SOURCE, MPI_TAG, and MPI_ERROR; the structure may contain additional fields. Thus,

status.MPI_SOURCE, status.MPI_TAG, and status.MPI_ERROR contain the source, tag, and error code, respectively, of the received message.

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

With Fortran USE mpi_f08, status is defined as the Fortran BIND(C) derived type TYPE(MPI_Status) containing three public INTEGER fields named MPI_SOURCE, MPI_TAG, and MPI_ERROR. TYPE(MPI_Status) may contain additional, implementation-specific fields. Thus, status%MPI_SOURCE, status%MPI_TAG, and status%MPI_ERROR contain the source, tag, and error code of a received message respectively. Additionally, within both the mpi and the mpi_f08 modules, the constants MPI_STATUS_SIZE, MPI_SOURCE, MPI_TAG, MPI_ERROR, and TYPE(MPI_Status) are defined to allow conversion between both status representations. Conversion routines are provided in Section 19.3.5.

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

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

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

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

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

 $44 \\ 45$

 $\mathbf{2}$

 $\mathbf{5}$

 24

1	MPI_GET_	_COUNT(status, datatype, cour	nt)
2	IN	status	return status of receive operation (status)
$\frac{3}{4}$	IN	datatype	datatype of each receive buffer entry (handle)
4 5	OUT	count	number of received entries (integer)
6	001	count	number of received entities (integer)
7	C binding	z	
8			*status, MPI_Datatype datatype,
9 10		<pre>int *count)</pre>	
10 11 12	int MPI_G	et_count_c(const MPI_Stat MPI_Count *count)	us *status, MPI_Datatype datatype,
13 14 15 16 17 18 19	MPI_Get_c TYPE(TYPE(INTEG	2008 binding count(status, datatype, co MPI_Status), INTENT(IN) : MPI_Datatype), INTENT(IN) ER, INTENT(OUT) :: count ER, OPTIONAL, INTENT(OUT)	: status :: datatype
20 21 22 23 24	TYPE(TYPE(INTEG	count(status, datatype, co MPI_Status), INTENT(IN) MPI_Datatype), INTENT(IN) ER(KIND=MPI_COUNT_KIND), ER, OPTIONAL, INTENT(OUT)	: status :: datatype INTENT(OUT) :: count
25 26 27 28		COUNT (STATUS, DATATYPE, CO	DUNT, IERROR) C), DATATYPE, COUNT, IERROR
29 30 31 32 33 34	not bytes.) that set th parameter.	The datatype argument should e status variable. If the numbe , then MPI_GET_COUNT sets	ed. (Again, we count <i>entries</i> , each of type <i>datatype</i> , d match the argument provided by the receive call or of entries received exceeds the limits of the count the value of count to MPI_UNDEFINED. There are can be set to MPI_UNDEFINED; see Section 5.1.11.
34 35	Ratio	onale. Some message-passing	g libraries use INOUT count, tag and source argu-
36			becify the selection criteria for incoming messages
37			ues of the received message. The use of a separate
38			at are often attached with INOUT argument (e.g.,
39			as the tag in a receive). Some libraries use calls
40 41		x v	essage received." This is not thread safe.
41			MPI_GET_COUNT so as to improve performance.
43		8 8	out counting the number of elements it contains, needed. Also, this allows the same function to be
44			r MPI_IPROBE. With a status from MPI_PROBE
45			es are allowed as in a call to MPI_RECV to receive
46	this	message. (End of rationale.)	
47			
48			

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

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

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

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

3.2.6 Passing MPI_STATUS_IGNORE for Status

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

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

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

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

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

Unofficial Draft for Comment Only

 $\overline{7}$

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

3.2.7 Send-Receive

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

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

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

25		source, reev	
26	IN	sendbuf	initial address of send buffer (choice)
27	IN	sendcount	number of elements in send buffer (non-negative
28			integer)
29 30	IN	sendtype	type of elements in send buffer (handle)
31	IN	dest	rank of destination (integer)
32	IN	sendtag	send tag (integer)
33 34	Ουτ	recvbuf	initial address of receive buffer (choice)
35	IN	recvcount	number of elements in receive buffer (non-negative
36			integer)
37	IN	recvtype	type of elements receive buffer element (handle)
38 39	IN	source	rank of source or MPI_ANY_SOURCE (integer)
40	IN	recvtag	receive tag or MPI_ANY_TAG (integer)
41	IN	comm	communicator (handle)
42 43	OUT	status	status object (status)
44			
45	C bindi	0	
46	int MPI		<pre>void *sendbuf, int sendcount, MPI_Datatype sendtype,</pre>
47		int dest,	int sendtag, void *recvbuf, int recvcount,
48			

5 6

7

22 23

1 MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, 2 MPI_Status *status) 3 int MPI_Sendrecv_c(const void *sendbuf, MPI_Count sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, 5 MPI_Count recvcount, MPI_Datatype recvtype, int source, 6 int recvtag, MPI_Comm comm, MPI_Status *status) Fortran 2008 binding 9 MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, 10 recvcount, recvtype, source, recvtag, comm, status, ierror) 11 TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source, 1213 recvtag 14TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 15TYPE(*), DIMENSION(..) :: recvbuf 16TYPE(MPI_Comm), INTENT(IN) :: comm 17TYPE(MPI_Status) :: status 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 19 MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, 20recvcount, recvtype, source, recvtag, comm, status, ierror) 21TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf 22 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount 23TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 24INTEGER, INTENT(IN) :: dest, sendtag, source, recvtag 25TYPE(*), DIMENSION(..) :: recvbuf 26TYPE(MPI_Comm), INTENT(IN) :: comm 27TYPE(MPI_Status) :: status 28INTEGER, OPTIONAL, INTENT(OUT) :: ierror 29 30 Fortran binding 31MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF, 32 RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR) 33 <type> SENDBUF(*), RECVBUF(*) 34 INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE, 35 SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR 36 Execute a blocking send and receive operation. Both send and receive use the same 37 communicator, but possibly different tags. The send buffer and receive buffers must be 38 disjoint, and may have different lengths and datatypes. 39 The semantics of a send-receive operation is what would be obtained if the caller forked 40 two concurrent threads, one to execute the send, and one to execute the receive, followed 41 by a join of these two threads. 4243 44

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

INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS (MPI_STATUS_SIZE), IERROR

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

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

3.3 Datatype Matching and Data Conversion

3.3.1 Type Matching Rules

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

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

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

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

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

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

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

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

 31

```
1
        • Communication of untyped values (e.g., of datatype MPI_BYTE), where both sender
\mathbf{2}
           and receiver use the datatype MPI_BYTE. In this case, there are no requirements on
3
           the types of the corresponding entries in the sender and the receiver programs, nor is
4
          it required that they be the same.
5
        • Communication involving packed data, where MPI_PACKED is used.
6
7
          The following examples illustrate the first two cases.
8
9
     Example 3.2 Sender and receiver specify matching types.
10
     CALL MPI_COMM_RANK(comm, rank, ierr)
11
     IF (rank .EQ. 0) THEN
12
         CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
13
     ELSE IF (rank .EQ. 1) THEN
14
        CALL MPI_RECV(b(1), 15, MPI_REAL, 0, tag, comm, status, ierr)
15
     END IF
16
17
         This code is correct if both a and b are real arrays of size > 10. (In Fortran, it might be
18
     correct to use this code even if a or b have size < 10: e.g., when a(1) can be equivalenced
19
     to an array with ten reals.)
20
21
     Example 3.3 Sender and receiver do not specify matching types.
22
23
     /* ----- THIS EXAMPLE IS ERRONEOUS ---
                                                                ---- */
^{24}
     CALL MPI_COMM_RANK(comm, rank, ierr)
25
     IF (rank .EQ. 0) THEN
26
         CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
27
     ELSE IF (rank .EQ. 1) THEN
28
        CALL MPI_RECV(b(1), 40, MPI_BYTE, 0, tag, comm, status, ierr)
29
     END IF
30
31
          This code is erroneous, since sender and receiver do not provide matching datatype
32
     arguments.
33
34
     Example 3.4 Sender and receiver specify communication of untyped values.
35
     CALL MPI_COMM_RANK(comm, rank, ierr)
36
     IF (rank .EQ. 0) THEN
37
         CALL MPI_SEND(a(1), 40, MPI_BYTE, 1, tag, comm, ierr)
38
     ELSE IF (rank .EQ. 1) THEN
39
        CALL MPI_RECV(b(1), 60, MPI_BYTE, 0, tag, comm, status, ierr)
40
     END IF
41
42
          This code is correct, irrespective of the type and size of a and b (unless this results in
43
     an out of bounds memory access).
44
45
           Advice to users. If a buffer of type MPI_BYTE is passed as an argument to MPI_SEND,
46
           then MPI will send the data stored at contiguous locations, starting from the address
47
           indicated by the buf argument. This may have unexpected results when the data
48
          layout is not as a casual user would expect it to be. For example, some Fortran
```

CHAPTER 3. POINT-TO-POINT COMMUNICATION

44

Unofficial Draft for Comment Only

 31

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

Type MPI_CHARACTER

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

Example 3.5 Transfer of Fortran CHARACTERs.

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

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

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

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

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

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

```
47
48
```

3.3.2 Data Conversion

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

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

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

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

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

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

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

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

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

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

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

48

38

1

2

3

4

5 6

7

8

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

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

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

3.4 Communication Modes

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

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

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

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

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

There are three additional communication modes.

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\overline{7}$

 24

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

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 exe cution of any MPI procedure in any other MPI process.

 $13 \\ 14$

15

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.*)

16A send that uses the **synchronous** mode can be started whether or not a matching 17receive was posted. However, the send will complete successfully only if a matching receive is 18 posted, and the receive operation has started to receive the message sent by the synchronous 19send. Thus, the completion of a synchronous send not only indicates that the send buffer 20can be reused, but it also indicates that the receiver has reached a certain point in its 21execution, namely that it has started executing the matching receive. If both sends and 22receives are blocking operations then the use of the synchronous mode provides synchronous 23communication semantics: a communication does not complete at either end before both 24 processes rendezvous at the communication. A send executed in this mode is non-local.

25A send that uses the **ready** communication mode may be started *only* if the matching 26receive is already posted. Otherwise, the operation is erroneous and its outcome is unde-27fined. On some systems, this allows the removal of a hand-shake operation that is otherwise 28required and results in improved performance. The completion of the send operation does 29not depend on the status of a matching receive, and merely indicates that the send buffer 30 can be reused. A send operation that uses the ready mode has the same semantics as a 31 standard send operation, or a synchronous send operation; it is merely that the sender 32 provides additional information to the system (namely that a matching receive is already 33 posted), that can save some overhead. In a correct program, therefore, a ready send could 34be replaced by a standard send with no effect on the behavior of the program other than 35 performance. 36

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.

40 41

37

38

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

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

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

C binding

int	MPI_Bsend(const	void	*buf,	int	count,	MPI_Datatype	datatype,	int	dest,
	int ta	g, MP	I_Comm	com	m)				

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

Fortran 2008 binding

Fortrail 2008 binding	10
MPI_Bsend(buf, count, datatype, dest, tag, comm, ierror)	19
TYPE(*), DIMENSION(), INTENT(IN) :: buf	20
INTEGER, INTENT(IN) :: count, dest, tag	21
TYPE(MPI_Datatype), INTENT(IN) :: datatype	22
TYPE(MPI_Comm), INTENT(IN) :: comm	23
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	24
,,,,,,,	25
MPI_Bsend(buf, count, datatype, dest, tag, comm, ierror)	26
TYPE(*), DIMENSION(), INTENT(IN) :: buf	27
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count	28
TYPE(MPI_Datatype), INTENT(IN) :: datatype	29
INTEGER, INTENT(IN) :: dest, tag	30
TYPE(MPI_Comm), INTENT(IN) :: comm	31
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	32
	33
Fortran binding	34
MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)	35
<type> BUF(*)</type>	36
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR	37
Send in buffered mode.	38
	39
$\overline{\mathbf{v}}$	

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

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

	· · · ·	- ,	
IN	buf	initial address of send buffer (choice)	2
IN	count	number of elements in send buffer (non-negative	3 4
		$\operatorname{integer})$	5
IN	datatype	datatype of each send buffer element (handle)	6
IN	dest	rank of destination (integer)	7
		raine of destination (meeser)	8
IN	tag	message tag (integer)	9
IN	comm	communicator (handle)	10
			11

C binding

<pre>int MPI_Rsend(const v</pre>	oid *buf,	int count,	MPI_Datatype	datatype, int	dest,
int tag	, MPI_Comm	comm)			
int MDT Drand a (and at		MDT Comme	MDT T		

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

Fortran 2008 binding

For trail 2008 binding	19
MPI_Rsend(buf, count, datatype, dest, tag, comm, ierror)	20
TYPE(*), DIMENSION(), INTENT(IN) :: buf	
INTEGER, INTENT(IN) :: count, dest, tag	21
TYPE(MPI_Datatype), INTENT(IN) :: datatype	22
TYPE(MPI_Comm), INTENT(IN) :: comm	23
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	24
	25
MPI_Rsend(buf, count, datatype, dest, tag, comm, ierror)	26
TYPE(*), DIMENSION(), INTENT(IN) :: buf	27
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count	28
TYPE(MPI_Datatype), INTENT(IN) :: datatype	29
INTEGER, INTENT(IN) :: dest, tag	30
TYPE(MPI_Comm), INTENT(IN) :: comm	31
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	32
	33
Fortran binding	34
MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)	35
<type> BUF(*)</type>	36
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR	37
	57

Send in ready mode.

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

In a multithreaded implementation of MPI, the system may de-schedule a thread that 43is blocked on a send or receive operation, and schedule another thread for execution in 44the same address space. In such a case it is the user's responsibility not to modify a 45communication buffer until the communication completes. Otherwise, the outcome of the 46computation is undefined. 47

Unofficial Draft for Comment Only

1

12

13 1415

16

17 18

38

39

40

 41

42

- 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,
- ⁴ for standard sends. The programmer can signal his or her preference for blocking the ⁵ sender until a matching receive occurs by using the synchronous send mode.
- A possible communication protocol for the various communication modes is outlined
 below.
- ⁹ *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.
- ¹⁶ buffered send: The sender copies the message into a buffer and then sends it with a ¹⁷ 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.*)

28 29 30

 31

32

33 34

25

26

27

3.5 Semantics of Point-to-Point Communication

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

Order Messages are *non-overtaking*: If a sender sends two messages in succession to the 35 same destination, and both match the same receive, then this operation cannot receive the 36 second message if the first one is still pending. If a receiver posts two receives in succession, 37 and both match the same message, then the second receive operation cannot be satisfied 38 by this message, if the first one is still pending. This requirement facilitates matching of 39 sends to receives. It guarantees that message-passing code is deterministic, if processes are 40 single-threaded and the wildcard MPI_ANY_SOURCE is not used in receives. (Some of the 41 calls described later, such as MPI_CANCEL or MPI_WAITANY, are additional sources of 42nondeterminism.) 43

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

Unofficial Draft for Comment Only

any order. Similarly, if two receive operations that are logically concurrent receive two successively sent messages, then the two messages can match the two receives in either order.

Example 3.6 An example of non-overtaking messages.

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

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

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

Example 3.7 An example of two, intertwined matching pairs.

```
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.

FairnessMPI makes no guarantee of *fairness* in the handling of communication. Suppose44that a send is posted. Then it is possible that the destination process repeatedly posts a45receive that matches this send, yet the message is never received, because it is each time46overtaken by another message, sent from another source. Similarly, suppose that a receive47was posted by a multithreaded process. Then it is possible that messages that match this48

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $41 \\ 42$

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.

4

 $\mathbf{5}$ Resource limitations Any pending communication operation consumes system resources 6 that are limited. Errors may occur when lack of resources prevent the execution of an MPI 7call. A quality implementation will use a (small) fixed amount of resources for each pending 8 send in the ready or synchronous mode and for each pending receive. However, buffer space 9 may be consumed to store messages sent in standard mode, and must be consumed to store 10 messages sent in buffered mode, when no matching receive is available. The amount of space 11available for buffering will be much smaller than program data memory on many systems. 12Then, it will be easy to write programs that overrun available buffer space.

¹³ MPI allows the user to provide buffer memory for messages sent in the buffered mode. ¹⁴ Furthermore, MPI specifies a detailed operational model for the use of this buffer. An MPI ¹⁵ implementation is required to do no worse than implied by this model. This allows users to ¹⁶ avoid buffer overflows when they use buffered sends. Buffer allocation and use is described ¹⁷ in Section 3.6.

18 A buffered send operation that cannot complete because of a lack of buffer space is 19erroneous. When such a situation is detected, an error is signaled that may cause the 20program to terminate abnormally. On the other hand, a standard send operation that 21cannot complete because of lack of buffer space will merely block, waiting for buffer space 22to become available or for a matching receive to be posted. This behavior is preferable in 23many situations. Consider a situation where a producer repeatedly produces new values 24 and sends them to a consumer. Assume that the producer produces new values faster 25than the consumer can consume them. If buffered sends are used, then a buffer overflow 26will result. Additional synchronization has to be added to the program so as to prevent 27this from occurring. If standard sends are used, then the producer will be automatically 28throttled, 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.

31 32

33

Example 3.8 An exchange of messages.

34 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.

 $_{46}^{45}$ **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)
ELSE IF (rank .EQ. 1) THEN
   CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
   CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

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

Example 3.10 An exchange that relies on buffering.

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

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

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

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

Many programmers prefer to have more leeway and opt to use the "unsafe" programming style shown in Example 3.10. In such cases, the use of standard sends is likely ⁴⁵ to provide the best compromise between performance and robustness: quality implementations will provide sufficient buffering so that "common practice" programs will ⁴⁷ not deadlock. The buffered send mode can be used for programs that require more ⁴⁸

Unofficial Draft for Comment Only

1 2

3

6

9 10

11

12

13

14

15 16

17

18

19 20

21

22 23

24 25

26

27

28

29

30

31 32

33

34

35

36

37

38

39

40

41

42

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.

16MPI_BUFFER_ATTACH(buffer, size) 17IN buffer initial buffer address (choice) 18 19IN size buffer size, in bytes (non-negative integer) 2021C binding 22int MPI_Buffer_attach(void *buffer, int size) 23int MPI_Buffer_attach_c(void *buffer, MPI_Count size) 24 25Fortran 2008 binding 26MPI_Buffer_attach(buffer, size, ierror) 27TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer 28INTEGER, INTENT(IN) :: size 29 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 30 MPI_Buffer_attach(buffer, size, ierror) 31 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer 32 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: size 33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 34 35 Fortran binding 36 MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR) 37 <type> BUFFER(*) 38 INTEGER SIZE, IERROR 39 Provides to MPI a buffer in the user's memory to be used for buffering outgoing mes-40

⁴¹ sages. The buffer is used only by messages sent in buffered mode. Only one buffer can be ⁴² attached to a process at a time. In C, buffer is the starting address of a memory region. In ⁴³ Fortran, one can pass the first element of a memory region or a whole array, which must be ⁴⁴ 'simply contiguous' (for 'simply contiguous,' see also Section 19.1.12).

- 45 46
- 47
- 48

1

2

3

4

5

6

7

8

9 10

11 12

13

MPI BUI	FER_DETACH(buffer	addr. size)	1
OUT		initial buffer address (choice)	2
OUT	size	buffer size, in bytes (integer)	3 4
	0.20		5
C bindi	ng		6
int MPI	_Buffer_detach(voi	d *buffer_addr, int *size)	7
int MPI	_Buffer_detach_c(v	oid *buffer_addr, MPI_Count *size)	8 9
Fortran	2008 binding		10
		addr, size, ierror)	11
		_C_BINDING, ONLY : C_PTR	12 13
	GER, INTENT(OUT)	UT) :: buffer_addr	13
	-	TENT(OUT) :: ierror	15
MPT Buff	fer detach(buffer	addr, size, ierror)	16
		_C_BINDING, ONLY : C_PTR	17 18
		UT) :: buffer_addr	19
		T_KIND), INTENT(OUT) :: size	20
	GER, UPIIUNAL, IN	TENT(OUT) :: ierror	21
	binding		22 23
	<pre>'ER_DETACH(BUFFER be> BUFFER_ADDR(*)</pre>	ADDR, SIZE, IERROR)	23
• •	EGER SIZE, IERROR		25
Deta	och the buffer current	ly associated with MPI. The call returns the address and the	26
		This operation will block until all messages currently in the	27 28
		Upon return of this function, the user may reuse or deallocate	29
	e taken by the buffer.		30
If th MPI_UND		d buffer cannot be represented in size, it is set to	31
	LIMED.		32
Exampl	e 3.11 Calls to attac	ch and detach buffers.	33 34
#define	BUFFSIZE 10000		35
int size	e;		36
char *bı			37
		BUFFSIZE), BUFFSIZE);	38 39
	fer_detach(&buff,	s can now be used by MPI_Bsend */ &size):	40
	er size reduced to		41
	fer_attach(buff, s		42
/* Buffe	er of 10000 bytes	available again */	43 44
Ads	vice to users. Even	though the C functions MPI_Buffer_attach and	44
		have a first argument of type void*, these arguments are	46
use	d differently: A point	er to the buffer is passed to MPI_Buffer_attach; the address	47

of the pointer is passed to MPI_Buffer_detach, so that this call can return the pointer Unofficial Draft for Comment Only

value. In Fortran with the mpi module or mpif.h, the type of the buffer_addr argument is wrongly defined and the argument is therefore unused. In Fortran with the mpi_f08 module, the address of the buffer is returned as TYPE(C_PTR), see also Example 9.1 about the use of C_PTR pointers. (*End of advice to users.*)

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

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

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

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

2526

27

28

29

30

 31

32

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.*)

33 34 35

36

37

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.
- 47 48

44 45

46

1

2

3

4

5 6

7

8 9

- Compute the number, n, of bytes needed to store an entry for the new message. An upper bound on n can be computed as follows: A call to the function MPI_PACK_SIZE(count, datatype, comm, size), with the count, datatype and comm arguments used in the MPI_BSEND call, returns an upper bound on the amount of space needed to buffer the message data (see Section 5.2). The MPI constant MPI_BSEND_OVERHEAD provides an upper bound on the additional space consumed by the entry (e.g., for pointers or envelope information).
- Find the next contiguous empty space of n bytes in buffer (space following queue tail, or space at start of buffer if queue tail is too close to end of buffer). If space is not found then raise buffer overflow error.
- Append to end of PME queue in contiguous space the new entry that contains request handle, next pointer and packed message data; MPI_PACK is used to pack data.
- Post nonblocking send (standard mode) for packed data.
- Return

3.7 Nonblocking Communication

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

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

Unofficial Draft for Comment Only

 24

1 Nonblocking send start calls can use the same four modes as blocking sends: *standard*, $\mathbf{2}$ buffered, synchronous and ready. These carry the same meaning. Sends of all modes, ready 3 excepted, can be started whether a matching receive has been posted or not; a nonblocking 4 **ready** send can be started only if a matching receive is posted. In all cases, the send start 5call is local: it returns immediately, irrespective of the status of other processes. If the call 6 causes some system resource to be exhausted, then it will fail and return an error code. 7 Quality implementations of MPI should ensure that this happens only in "pathological" 8 cases. That is, an MPI implementation should be able to support a large number of pending 9 nonblocking operations.

¹⁰ The send-complete call returns when data has been copied out of the send buffer. It ¹¹ may carry additional meaning, depending on the send mode.

¹² If the send mode is **synchronous**, then the send can complete only if a matching receive ¹³ has started. That is, a receive has been posted, and has been matched with the send. In ¹⁴ this case, the send-complete call is non-local. Note that a synchronous, nonblocking send ¹⁵ may complete, if matched by a nonblocking receive, before the receive complete call occurs. ¹⁶ (It can complete as soon as the sender "knows" the transfer will complete, but before the ¹⁷ receiver "knows" the transfer will complete.)

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

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

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

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

32 Nonblocking sends in the buffered and ready modes have a more limited impact, e.g., 33 the blocking version of buffered send is capable of completing regardless of when a 34matching receive call is made. However, separating the start from the completion 35 of these sends still gives some opportunity for optimization within the MPI library. 36 For example, starting a buffered send gives an implementation more flexibility in 37 determining if and how the message is buffered. There are also advantages for both 38 nonblocking buffered and ready modes when data copying can be done concurrently 39 with computation. 40

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.*)

48

25

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.

in r_io_ive/ built, dutatype, dest, dag, comm, request)			
IN	buf	initial address of send buffer (choice)	19
IN	count	number of elements in send buffer (non-negative	20
	count	integer)	21
			22
IN	datatype	datatype of each send buffer element (handle)	23
IN	dest	rank of destination (integer)	24
IN	tag	message tag (integer)	25 26
IN	comm	communicator (handle)	20
			21
OUT	request	communication request (handle)	20
			30
\mathbf{C} binding	g		31
int MPI_1	<pre>int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>		
<pre>int tag, MPI_Comm comm, MPI_Request *request)</pre>			33
int MPI_Isend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,			34
-	int dest, int tag, MPI_Comm comm, MPI_Request *request)		35
			36
Fortran 2008 binding			37
	MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror)		
	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf		
INTEGER, INTENT(IN) :: count, dest, tag			40
TYPE(MPI_Datatype), INTENT(IN) :: datatype			41
	TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request		
	•	÷	43
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror		
MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror)			45

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

М ype, TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype

Unofficial Draft for Comment Only

1 $\mathbf{2}$

3

4

 $\mathbf{5}$

6

7

8

9 10

11 12

13

14

15

1617

18

46 47

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

Fortran binding MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)			1 2 3
51	<type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR</type>		
			4 5
Start	a buffered mode, nonblock	ing send.	6
			7 8
MPI_ISSE	MPI_ISSEND(buf, count, datatype, dest, tag, comm, request)		
IN	buf	initial address of send buffer (choice)	9 10
IN	count	number of elements in send buffer (non-negative integer)	11 12
IN	datatype	datatype of each send buffer element (handle)	13
IN	dest	rank of destination (integer)	14 15
IN	tag	message tag (integer)	16
IN	comm	communicator (handle)	17
OUT	request	communication request (handle)	18
001	request	communication request (nandle)	19 20
C bindin	ıg		20 21
	0	int count, MPI_Datatype datatype, int dest,	22
	int tag, MPI_Comm	comm, MPI_Request *request)	23
int MPI_	Issend_c(const void *bu	if, MPI_Count count, MPI_Datatype datatype,	24
int dest, int tag, MPI_Comm comm, MPI_Request *request)			25 26
Fortran 2008 binding			27
	MPI_Issend(buf, count, datatype, dest, tag, comm, request, ierror)		
	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf		
INTEGER, INTENT(IN) :: count, dest, tag			30
	<pre>(MPI_Datatype), INTENT((MPI_Comm), INTENT(IN)</pre>		31 32
	(MPI_Request), INTENT(IN)		33
INTEGER, OPTIONAL, INTENT(OUT) :: ierror			34
		e, dest, tag, comm, request, ierror) TENT(IN), ASYNCHRONOUS :: buf	36
	GER(KIND=MPI_COUNT_KIND		37
	(MPI_Datatype), INTENT(-	38
	GER, INTENT(IN) :: dest	01	39 40
TYPE	TYPE(MPI_Comm), INTENT(IN) :: comm		
	(MPI_Request), INTENT(C	•	41 42
INTE	GER, OPTIONAL, INTENT(C	DUT) :: ierror	43
Fortran	Fortran binding		
	MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)		
<type> BUF(*) 4</type>			46
INTE	INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 47		
	48		

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

3.7. NONBLOCKING COMMUNICATION

MPI_IRECV(buf, count, datatype, source, tag, comm, request) ¹				
OUT	buf	initial address of receive buffer (choice)	2 3	
IN	count	number of elements in receive buffer (non-negative	4	
		integer)	5	
IN	datatype	datatype of each receive buffer element (handle)	6	
IN	source	rank of source or MPI_ANY_SOURCE (integer)	7 8	
IN	tag	message tag or MPI_ANY_TAG (integer)	9	
IN	comm	communicator (handle)	10	
OUT	request	communication request (handle)	11	
			12 13	
C bin	•		14	
int M		<pre>int count, MPI_Datatype datatype, int source, [_Comm comm, MPI_Request *request)</pre>	15 16	
int M	PI_Irecv_c(void *buf	, MPI_Count count, MPI_Datatype datatype,	17	
	int source,	<pre>int tag, MPI_Comm comm, MPI_Request *request)</pre>	18 19	
Fortra	an 2008 binding		20	
		atype, source, tag, comm, request, ierror)	21	
), ASYNCHRONOUS :: buf	22	
	INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm			
	TYPE(MPI_Request), INTENT(OUT) :: request			
INTEGER, OPTIONAL, INTENT(OUT) :: ierror			27	
MPI_I:	recv(buf, count, dat	atype, source, tag, comm, request, ierror)	28	
), ASYNCHRONOUS :: buf	29 30	
		T_KIND), INTENT(IN) :: count	31	
	TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(IN) :: source, tag			
	TYPE(MPI_Comm), INTENT(IN) :: comm			
	TYPE(MPI_Request), INTENT(OUT) :: request			
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
Fortran binding			36	
		ATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)	37 38	
	<pre><type> BUF(*)</type></pre>			
I	NTEGER COUNT, DATATY	PE, SOURCE, TAG, COMM, REQUEST, IERROR	40	
St	art a nonblocking recei	ve.	41	
5			42	
			43 44	
			45 46	

1 2	MPI_ISEN	DRECV(sendbuf, sendcount, se source, recvtag, comm, r	endtype, dest, sendtag, recvbuf, recvcount, recvtype, equest)
3	IN	sendbuf	initial address of send buffer (choice)
4 5 6	IN	sendcount	number of elements in send buffer (non-negative integer)
7	IN	sendtype	datatype of each send buffer element (handle)
8	IN	dest	rank of destination (integer)
9 10	IN	sendtag	send tag (integer)
11	OUT	recvbuf	initial address of receive buffer (choice)
12 13 14	IN	recvcount	number of elements in receive buffer (non-negative integer)
14 15	IN	recvtype	datatype of each receive buffer element (handle)
16	IN	source	rank of source or MPI_ANY_SOURCE (integer)
17	IN	recvtag	receive tag or MPI_ANY_TAG (integer)
18 19	IN	comm	communicator (handle)
20	OUT	request	communication request (handle)
21		·	
22 23	C binding	-	
23	int MPI_1	Isendrecv(const void *sen	
25	<pre>MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag,</pre>		
26 27	MPI_Comm comm, MPI_Request *request)		
28	int MPI_Isendrecv_c(const void *sendbuf, MPI_Count sendcount,		
29	NDT Determine and have sint last and the model in the second state		
30			MPI_Datatype recvtype, int source,
31 32			m comm, MPI_Request *request)
33		2008 binding	
34	<pre>MPI_Isendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, request, ierror)</pre>		
35	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf		
36 37	INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source,		
38	recvtag		
39	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype		
40	TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf TYPE(MPI_Comm), INTENT(IN) :: comm		
41	TYPE(MPI_Comm), INTENT(IN) comm TYPE(MPI_Request), INTENT(OUT) :: request		
42 43	INTEGER, OPTIONAL, INTENT(OUT) :: ierror		
43	MPI_Isend	lrecv(sendbuf, sendcount,	sendtype, dest, sendtag, recvbuf,
45		recvcount, recvtype,	source, recvtag, comm, request, ierror)
46			T(IN), ASYNCHRONOUS :: sendbuf
47		GER(KIND=MPI_COUNT_KIND), (MPI_Datatype), INTENT(IN	INTENT(IN) :: sendcount, recvcount
48	11FE((m r_bacacype), INIENI(IN	/ senatype, recytype

Unofficial Draft for Comment Only

		t, sendtag, source, recvtag	1	
TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf			2 3	
	TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request			
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
Fortran k		NT SENDTYPE DEST SENDTAG RECUBILE	7 8	
	MPI_ISENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, REQUEST, IERROR)			
<pre><type> SENDBUF(*), RECVBUF(*)</type></pre>				
INTEC		E, DEST, SENDTAG, RECVCOUNT, RECVTYPE,	11	
	SOURCE, RECVTAG,	COMM, REQUEST, IERROR	12	
Initia	te a nonblocking communi	cation request for a <i>send and receive</i> operation.	13 14	
			15	
MPI_ISEN		unt, datatype, dest, sendtag, source, recvtag, comm,	16	
	request)		17	
INOUT	buf	initial address of send and receive buffer (choice)	18 19	
IN	count	number of elements in send and receive buffer	20	
		(non-negative integer)	21	
IN	datatype	type of elements in send and receive buffer (handle)	22	
IN	dest	rank of destination (integer)	23	
IN	sendtag	send message tag (integer)	24 25	
IN	source	rank of source or MPI_{ANY}_{SOURCE} (integer)	26	
IN	recvtag	receive message tag or MPI_ANY_TAG (integer)	27	
IN	comm	communicator (handle)	28 29	
OUT	request	communication request (handle)	30	
			31	
C bindin	5	which intervent MDI Detetoring detetoring	32 33	
int MPI_J	-	*buf, int count, MPI_Datatype datatype, dtag, int source, int recvtag, MPI_Comm comm,	34	
	MPI_Request *requ		35	
			36	
int MPI_J	-	<pre>id *buf, MPI_Count count, type, int dest, int sendtag, int source,</pre>	37	
		Comm comm, MPI_Request *request)	38 39	
Fontnon (
	2008 binding	nt, datatype, dest, sendtag, source, recvtag,	40 41	
11 1_10010	comm, request, ie		42	
TYPE	(*), DIMENSION(), AS		43	
		nt, dest, sendtag, source, recvtag	44	
	(MPI_Datatype), INTENT		45 46	
	(MPI_Comm), INTENT(IN)		40 47	
TYPE(MPI_Request), INTENT(OUT) :: request47INTEGER, OPTIONAL, INTENT(OUT) :: ierror48				
,,,,,				

Unofficial Draft for Comment Only

1	<pre>MPI_Isendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,</pre>
2	comm, request, ierror)
3	TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf
4	INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
5	TYPE(MPI_Datatype), INTENT(IN) :: datatype
6	INTEGER, INTENT(IN) :: dest, sendtag, source, recvtag
7	TYPE(MPI_Comm), INTENT(IN) :: comm
8	TYPE(MPI_Request), INTENT(OUT) :: request
9	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10	
11	Fortran binding
12	MPI_ISENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
13	COMM, REQUEST, IERROR)
	<type> BUF(*)</type>
14	INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, REQUEST,
15	IERROR
16	IERROR
17	Initiate a nonblocking communication request for a send and receive operation. The
18	same buffer is used both for the send and for the receive, so that the message sent is replaced
19	by the message received.
20	These calls allocate a communication request object and associate it with the request
21	
22	handle (the argument request). The request can be used later to query the status of the
	communication or wait for its completion.
23	A nonblocking send call indicates that the system may start copying data out of the
24	send buffer. The sender should not modify any part of the send buffer after a nonblocking
25	send operation is called, until the send completes.
26	A nonblocking receive call indicates that the system may start writing data into the re-
27	ceive buffer. The receiver should not access any part of the receive buffer after a nonblocking
28	receive operation is called, until the receive completes.
29	
30	Advice to users. To prevent problems with the argument copying and register
31	optimization done by Fortran compilers, please note the hints in Sections 19.1.10–
32	
	19.1.20. (End of advice to users.)
33	
34	3.7.3 Communication Completion
35	The functions MPI_WAIT and MPI_TEST are used to complete a nonblocking communica-
36	
37	tion. The completion of a send operation indicates that the sender is now free to update
38	the locations in the send buffer (the send operation itself leaves the content of the send
39	buffer unchanged). It does not indicate that the message has been received, rather, it may
40	have been buffered by the communication subsystem. However, if a synchronous mode
41	send was used, the completion of the send operation indicates that a matching receive was
42	initiated, and that the message will eventually be received by this matching receive.
43	The completion of a receive operation indicates that the receive buffer contains the
43	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,
45	that the send was initiated).
46	We shall use the following terminology: A null handle is a handle with value
47	we shan use the following terminology. A num nanche is a nanche with value

48 MPI_REQUEST_NULL. A persistent request and the handle to it are **inactive** if the request

is not associated with any ongoing communication (see Section 3.9). A handle is active if it is neither null nor inactive. An empty status is a status which is set to return tag = MPI_ANY_TAG, source = MPI_ANY_SOURCE, error = MPI_SUCCESS, and is also internally configured so that calls to MPI_GET_COUNT, MPI_GET_ELEMENTS, and MPI_GET_ELEMENTS_X return count = 0 and MPI_TEST_CANCELLED returns false. We set a status variable to empty when the value returned by it is not significant. Status is set in this way so as to prevent errors due to accesses of stale information.

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

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

		18
MPI_WAIT(request, status)		19
INOUT request	request (handle)	20
OUT status	status object (status)	21
oor status	status object (status)	22
		23
C binding		24
int MPI_Wait(MPI_Request *request, MPI_Status *status)		
Fortran 2008 binding		
PI_Wait(request, status, ierror)		
TYPE(MPI_Request), INTENT(INOUT) :: request		
TYPE(MPI_Status) :: status		29
INTEGER, OPTIONAL, INTENT(OUT)	:: ierror	30
		31
Fortran binding		32
MPI_WAIT(REQUEST, STATUS, IERROR)		33
INTEGER REQUEST, STATUS(MPI_ST	TATUS_SIZE), IERROR	34

A call to MPI_WAIT returns when the operation identified by request is complete. If the request is an active persistent request, it is marked inactive. Any other type of request is deallocated and the request handle is set to MPI_REQUEST_NULL. MPI_WAIT is a non-local operation.

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

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

Advice to users. Successful return of MPI_WAIT after a MPI_IBSEND implies that the user send buffer can be reused—i.e., data has been sent out or copied into a buffer 47 attached with MPI_BUFFER_ATTACH. Note that, at this point, we can no longer 48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6 7

8 9

10

11

12

13

14 15

16

17

35

36

37

38

39

40

41

42

43

44

1 2 3 4 5	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). (<i>End of advice to users.</i>)			
6 7 8	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. (<i>End of advice to implementors.</i>)			
9 10 11 12	MPI_TEST(request, flag, status)			
13	INOUT request c	communication request (handle)		
14	OUT flag t	rue if operation completed (logical)		
15 16	OUT status s	tatus object (status)		
17 18	C binding			
19	int MPI_Test(MPI_Request *request, i	int *flag, MPI_Status *status)		
20	Fortran 2008 binding			
21	MPI_Test(request, flag, status, ier	cor)		
22	TYPE(MPI_Request), INTENT(INOUT)			
23	LOGICAL, INTENT(OUT) :: flag			
24	TYPE(MPI_Status) :: status			
25	INTEGER, OPTIONAL, INTENT(OUT) :	:: ierror		
26 27	Fortron hinding			
28				
29	INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR			
30	LOGICAL FLAG			
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45	A call to MPI_TEST returns flag = true if the operation identified by request is complete. In such a case, the status object is set to contain information on the completed operation. If the request is an active persistent request, it is marked as inactive. Any other type of request is deallocated and the request handle is set to MPI_REQUEST_NULL. The call returns flag = false if the operation identified by request is not complete. In this case, the value of the status object is undefined. MPI_TEST is a local operation. The return status object for a receive operation carries information that can be accessed as described in Section 3.2.5. The status object for a send operation carries information that can be accessed by a call to MPI_TEST_CANCELLED (see Section 3.8). One is allowed to call MPI_TEST with a null or inactive request argument. In such a case the operation returns with flag = true and empty status. The functions MPI_WAIT and MPI_TEST can be used to complete any request-based nonblocking or persistent operation.			
45 46 47 48	schedule alternative activities within	nonblocking MPI_TEST call allows the user to a single thread of execution. An event-driven th periodic calls to MPI_TEST. (<i>End of advice to</i>		

Example 3.12 Simple usage of nonblocking operations and MPI_WAIT.

Example 3.12 Shiple usage of honolocking operations and with _wATT.	
CALL MPI_COMM_RANK(comm, rank, ierr)	2 3
IF (rank .EQ. 0) THEN	4
CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)	5
**** do some computation to mask latency ****	6
CALL MPI_WAIT(request, status, ierr)	7
ELSE IF (rank .EQ. 1) THEN	8
CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr)	9
**** do some computation to mask latency ****	10
CALL MPI_WAIT(request, status, ierr)	11
END IF	12
A request object can be freed using the following MPI procedure	13
A request object can be freed using the following MPI procedure.	14
	15
MPI_REQUEST_FREE(request)	16
INOUT request communication request (handle)	17
	18
C binding	19
int MPI_Request_free(MPI_Request *request)	20
	21
Fortran 2008 binding	22
MPI_Request_free(request, ierror)	23
TYPE(MPI_Request), INTENT(INOUT) :: request	24
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	25 26
Fortran binding	20 27
MPI_REQUEST_FREE(REQUEST, IERROR)	28
INTEGER REQUEST, IERROR	20
	30
MPI_REQUEST_FREE is a local operation. Upon successful return,	31
MPI_REQUEST_FREE sets request to MPI_REQUEST_NULL. For an inactive	32
request representing any type of MPI operation, MPI_REQUEST_FREE shall do the freeing	33
stage of the associated operation during its execution.	34
For a request representing a nonblocking point-to-point or a persistent point-to-point	35
operation, it is permitted (although strongly discouraged) to call MPI_REQUEST_FREE	36
when the request is active. In this special case, MPI_REQUEST_FREE will only mark the	37
request for freeing and MPI will actually do the freeing stage of the associated operation	38
later.	39
The use of this routine for generalized requests is described in Section 13.2.	40
Calling MPI_REQUEST_FREE with an active request representing any other type of	41
MPI operation (e.g., any partitioned operation (see Chapter 4), any collective operation $(1, 1, 2)$	42
(see Chapter 6), any I/O operation (see Chapter 14), or any request-based RMA operation	43
(see Chapter 12)) is erroneous.	44
Rationale. For point-to-point operations, the MPI_REQUEST_FREE mechanism is	45
provided for reasons of performance and convenience on the sending side. (End of	46
provided for reasons of performance and convenience on the sending side. (End of	47

Unofficial Draft for Comment Only

rationale.)

1

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.*)

```
9
     Example 3.13 An example using MPI_REQUEST_FREE.
10
11
     CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
12
     IF (rank .EQ. 0) THEN
13
        DO i=1,n
14
           CALL MPI_ISEND(outval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)
15
           CALL MPI_REQUEST_FREE(req, ierr)
16
           CALL MPI_IRECV(inval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)
17
           CALL MPI_WAIT(req, status, ierr)
18
        END DO
19
     ELSE IF (rank .EQ. 1) THEN
20
        CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
21
        CALL MPI_WAIT(req, status, ierr)
22
        DO I=1,n-1
23
           CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
24
           CALL MPI_REQUEST_FREE(req, ierr)
25
           CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
26
           CALL MPI_WAIT(req, status, ierr)
27
        END DO
28
        CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
29
        CALL MPI_WAIT(req, status, ierr)
30
     END IF
^{31}
32
```

```
3.7.4 Semantics of Nonblocking Communications
```

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

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

⁴¹ **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

Unofficial Draft for Comment Only

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

CHAPTER 3. POINT-TO-POINT COMMUNICATION

	(
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 MPI_UNDEFINED if none completed (integer)
OUT	flag	true if one of the operations is complete (logical)
OUT	status	status object (status)

MPI_TESTANY(count, array_of_requests, index, flag, status)

C binding

Fortran 2008 binding

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

Fortran binding

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

LOGICAL FLAG

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

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

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

 24

1 MPI_WAITALL(count, array_of_requests, array_of_statuses) 2 IN count list length (non-negative integer) 3 INOUT array_of_requests array of requests (array of handles) 4 5OUT array_of_statuses array of status objects (array of status) 6 7 C binding 8 int MPI_Waitall(int count, MPI_Request array_of_requests[], 9 MPI_Status array_of_statuses[]) 10 Fortran 2008 binding 11 MPI_Waitall(count, array_of_requests, array_of_statuses, ierror) 12INTEGER, INTENT(IN) :: count 13 TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count) 14TYPE(MPI_Status) :: array_of_statuses(*) 15INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1617 Fortran binding 18 MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR) 19INTEGER COUNT, ARRAY_OF_REQUESTS(*), ARRAY_OF_STATUSES(MPI_STATUS_SIZE, 20*), IERROR 21Blocks until all communication operations associated with active handles in the list 22 complete, and return the status of all these operations (this includes the case where no 23handle in the list is active). Both arrays have the same number of valid entries. The 24 i-th entry in array_of_statuses is set to the return status of the i-th operation. Active 25persistent requests are marked inactive. Requests of any other type are deallocated and the 26corresponding handles in the array are set to MPI_REQUEST_NULL. The list may contain 27null or inactive handles. The call sets to empty the status of each such entry. 28The error-free execution of MPI_WAITALL has the same effect as the execution of 29 MPI_WAIT for each of the array elements in some arbitrary order. MPI_WAITALL with an 30 array of length one is equivalent to MPI_WAIT. 31 When one or more of the communications completed by a call to MPI_WAITALL fail, 32 it is desirable to return specific information on each communication. The function 33 34MPI_WAITALL will return in such case the error code MPI_ERR_IN_STATUS and will set the error field of each status to a specific error code. This code will be MPI_SUCCESS, if the 35 specific communication completed; it will be another specific error code, if it failed; or it can 36 be MPI_ERR_PENDING if it has neither failed nor completed. The function MPI_WAITALL 37 will return MPI_SUCCESS if no request had an error, or will return another error code if it 38 failed for other reasons (such as invalid arguments). In such cases, it will not update the 39 error fields of the statuses. 40 41 Rationale. This design streamlines error handling in the application. The application 42code need only test the (single) function result to determine if an error has occurred. It 43 needs to check each individual status only when an error occurred. (End of rationale.) 44 4546 47 48

MPI_TEST	ALL(count, array_of_requests,	flag, array_of_statuses)	1
IN	count	list length (non-negative integer)	2
INOUT	array_of_requests	array of requests (array of handles)	3 4
OUT	flag	true if all of the operations are complete (logical)	5
OUT	array_of_statuses	array of status objects (array of status)	6
001	anay_01_statuses	array of status objects (array of status)	7
C binding	g		8 9
		<pre>quest array_of_requests[], int *flag,</pre>	9 10
	MPI_Status array_of_	statuses[])	11
Fortran 2	2008 binding		12
	0	sts, flag, array_of_statuses, ierror)	13
	ER, INTENT(IN) :: count		14 15
	MPI_Request), INTENT(INO CAL, INTENT(OUT) :: flag	UT) :: array_of_requests(count)	16
	(MPI_Status) :: array_of_:	statuses(*)	17
	ER, OPTIONAL, INTENT(OUT)		18
Fortran k			19
	6	STS, FLAG, ARRAY_OF_STATUSES, IERROR)	20 21
		STS(*), ARRAY_OF_STATUSES(MPI_STATUS_SIZE,	21
	*), IERROR		23
LOGIC	CAL FLAG		24
Retur	ns flag = true if all communic	cations associated with active handles in the array	25
_		here no handle in the list is active). In this case, each	26
		ve request is set to the status of the corresponding	27 28
-		e marked inactive. Requests of any other type are dles in the array are set to MPI_REQUEST_NULL.	29
		null or inactive handle is set to empty.	30
		no request is modified and the values of the status	31
	undefined. This is a local op		32
		ecution of MPI_TESTALL are handled in the same	33 34
manner as	errors in MPI_WAITALL.		35
			36
			37
			38
			39 40
			40
			42
			43
			44
			45
			46 47
			47

1 2	MPI_WAIT	SOME(incount, array_of_requ	ests, outcount, array_of_indices, array_of_statuses)		
3	IN	incount	length of array_of_requests (non-negative integer)		
4 5	INOUT	array_of_requests	array of requests (array of handles)		
6	OUT	outcount	number of completed requests (integer)		
7 8 9	OUT	array_of_indices	array of indices of operations that completed (array of integers)		
10 11	Ουτ	array_of_statuses	array of status objects for operations that completed (array of status)		
12	C hinding	•			
13 14 15 16	C binding int MPI_W	-			
17 18 19 20 21 22 23 24 25	MPI_Waits INTEG TYPE(INTEG TYPE(array_of_statuses, i ER, INTENT(IN) :: incoun	t UT) :: array_of_requests(incount) unt, array_of_indices(*) statuses(*)		
26 27 28 29 30	Fortran binding MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES, ARRAY_OF_STATUSES, IERROR) INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*), ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *), IERROR				
31 32 33 34 35 36 37 38 39 40 41	completed. have comp indices of t from zero in array_of_st requests ar and the ass If the = MPI_UNI	Returns in outcount the num leted. Returns in the first o chese operations (index within n C and from one in Fortran). atuses the status for these co re marked as inactive. Any ot sociated handle is set to MPI_ list contains no active handle DEFINED.	s, then the call returns immediately with outcount		
42 43 44 45 46 47 48	it is desiral outcount, a all commu MPI_ERR_II success or	ble to return specific informa mray_of_indices and array_of_ nications that have succeeden N_STATUS and the error field to indicate the specific error	ications completed by MPI_WAITSOME fails, then tion on each communication. The arguments statuses will be adjusted to indicate completion of ed or failed. The call will return the error code d of each status returned will be set to indicate that occurred. The call will return MPI_SUCCESS will return another error code if it failed for other		

Unofficial Draft for Comment Only

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

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

_	(;) = - 1	· · · · · · · · · · · · · · · · · · ·	
			6 7
IN	incount	length of array_of_requests (non-negative integer)	8
INOUT	array_of_requests	array of requests (array of handles)	9
OUT	outcount	number of completed requests (integer)	10
OUT	array_of_indices	array of indices of operations that completed (array of integers)	11 12
OUT	array_of_statuses	array of status objects for operations that completed (array of status)	13 14 15
			16
C binding			17
int MPI_T		_Request array_of_requests[],	18
	int *outcount, int a	-	19
	MPI_Status array_of_	statuses[])	20
Fortran 2	008 binding		21
MPI_Tests	ome(incount, array_of_red	quests, outcount, array_of_indices,	22
array_of_statuses, ierror)			
	ER, INTENT(IN) :: incount		24
	-	<pre>JT) :: array_of_requests(incount)</pre>	25
	ER, INTENT(OUT) :: outcou	•	26 27
	MPI_Status) :: array_of_s		28
INIEG	ER, OPTIONAL, INTENT(OUT)) :: lerror	29
Fortran b	oinding		30
MPI_TESTS		QUESTS, OUTCOUNT, ARRAY_OF_INDICES,	31
	ARRAY_OF_STATUSES, I		32
INTEG		JESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),	33
	ARRAY_OF_STATUSES(M	PI_STATUS_SIZE, *), IERROR	34
Behav	es like MPI_WAITSOME, exce	pt that it returns immediately. If no operation has	35
completed	it returns $outcount = 0$. If the	re is no active handle in the list it returns outcount	36
$=$ MPI_UN			37
		on, which returns immediately, whereas	38
		munication completes, if it was passed a list that	39 40
		the calls fulfill a fairness requirement: If a request	40 41
		t of requests passed to MPI_WAITSOME or	42
		has been posted, then the receive will eventually	43
succeed, unless the send is satisfied by another receive; and similarly for send requests.			

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

Advice to users. The use of MPI_TESTSOME is likely to be more efficient than the use of MPI_TESTANY. The former returns information on all completed communications,

Unofficial Draft for Comment Only

1 2

 $\frac{3}{4}$

5

44

45

46

```
1
           with the latter, a new call is required for each communication that completes.
\mathbf{2}
           A server with multiple clients can use MPI_WAITSOME so as not to starve any client.
3
           Clients send messages to the server with service requests. The server calls
4
           MPI_WAITSOME with one receive request for each client, and then handles all receives
5
           that completed. If a call to MPI_WAITANY is used instead, then one client could starve
6
           while requests from another client always sneak in first. (End of advice to users.)
7
8
                                    MPI_TESTSOME should complete as many pending com-
           Advice to implementors.
9
           munications as possible. (End of advice to implementors.)
10
11
     Example 3.16 Client-server code (starvation can occur).
12
13
     CALL MPI_COMM_SIZE(comm, size, ierr)
14
     CALL MPI_COMM_RANK(comm, rank, ierr)
15
     IF (rank .GT. 0) THEN
                                       ! client code
16
        DO WHILE(.TRUE.)
17
            CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
18
            CALL MPI_WAIT(request, status, ierr)
19
        END DO
20
     ELSE
                    ! rank=0 -- server code
21
        DO i=1, size-1
22
            CALL MPI_IRECV(a(1,i), n, MPI_REAL, i, tag, &
23
                             comm, request_list(i), ierr)
24
        END DO
25
        DO WHILE(.TRUE.)
26
            CALL MPI_WAITANY(size-1, request_list, index, status, ierr)
27
            CALL DO_SERVICE(a(1,index)) ! handle one message
28
            CALL MPI_IRECV(a(1, index), n, MPI_REAL, index, tag, &
29
                             comm, request_list(index), ierr)
30
        END DO
31
     END IF
32
33
34
     Example 3.17 Same code, using MPI_WAITSOME.
35
     CALL MPI_COMM_SIZE(comm, size, ierr)
36
     CALL MPI_COMM_RANK(comm, rank, ierr)
37
     IF (rank .GT. 0) THEN
                                       ! client code
38
        DO WHILE(.TRUE.)
39
            CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
40
            CALL MPI_WAIT(request, status, ierr)
41
        END DO
42
     ELSE
                    ! rank=0 -- server code
43
        DO i=1, size-1
44
            CALL MPI_IRECV(a(1,i), n, MPI_REAL, i, tag, &
45
                             comm, request_list(i), ierr)
46
        END DO
47
        DO WHILE(.TRUE.)
```

CHAPTER 3. POINT-TO-POINT COMMUNICATION

80

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)				
		- ,	19	
IN	request	request (handle)	20	
OUT	flag	boolean flag, same as from MPI_TEST (logical)	21	
OUT	status	status object if flag is true (status)	22	
			23	
C binding	r		24	
		equest request, int *flag,	25	
IIIC MFI_N	MPI_Status *status)	quest request, int *riag,	26	
	MFI_Status *Status)		27	
Fortran 2	2008 binding		28	
MPI_Reque	st_get_status(request, f	lag, status, ierror)	29	
TYPE(MPI_Request), INTENT(IN)	:: request	30	
LOGIC	AL, INTENT(OUT) :: flag		31	
TYPE (MPI_Status) :: status		32	
INTEG	ER, OPTIONAL, INTENT(OUT) :: ierror	33	
Foutnow h	in dia a		34	
Fortran b	<u> </u>		35	
	ST_GET_STATUS (REQUEST, F		36	
	ER REQUEST, STATUS(MPI_S	STATUS_SIZE), TERRUR	37	
LUGIC	CAL FLAG		38	
Sets fl	ag = true if the operation is	s complete, and, if so, returns in status the request	39	
status. Ho	owever, unlike test or wait,	it does not deallocate or inactivate the request; a	40	
subsequent	call to test, wait or free sh	ould be executed with that request. It sets flag =	41	
-	e operation is not complete.		42	
One is	One is allowed to call MPL REQUEST GET STATUS with a null or inactive request			

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

 $\mathbf{2}$

1	3.8 Pro	obe and Cancel			
2 3 4 5 6 7 8 9 10 11 12 13	The MPI_PROBE, MPI_IPROBE, MPI_MPROBE, and MPI_IMPROBE operations allow in- coming messages to be checked for, without actually receiving them. The user can then decide how to receive them, based on the information returned by the probe (basically, the information returned by status). In particular, the user may allocate memory for the receive buffer, according to the length of the probed message. The MPI_CANCEL operation allows pending communications to be cancelled. This is required for cleanup. Posting a send or a receive ties up user resources (send or receive buffers), and a cancel may be needed to free these resources gracefully. Cancelling a send request by calling MPI_CANCEL is deprecated. Cancelling a sendrecv request by calling MPI_CANCEL is not allowed.				
14	3.8.1 Pr	obe			
15 16					
17	MPI_IPRC)BE(source, tag, comm, flag, st	atus)		
18 19	IN	source	rank of source or MPI_ANY_SOURCE (integer)		
20	IN	tag	message tag or MPI_ANY_TAG (integer)		
21 22	IN	comm	communicator (handle)		
23 24	OUT	flag	true if there is a matching message that can be received (logical)		
25	OUT	status	status object (status)		
27 28 29 30 31 32 33 34 35 36 37	<pre>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</pre>				
38 39 40 41 42 43 44 45 46 47 48	<pre>Fortran binding MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR) INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR LOGICAL FLAG MPI_IPROBE(source, tag, comm, flag, status) returns flag = true if there is a message that can be received and that matches the pattern specified by the arguments source, tag, and comm. The call matches the same message that would have been received by a call to MPI_RECV(, source, tag, comm, status) executed at the same point in the program, and returns in status the same value that would have been returned by MPI_RECV(). Otherwise, the call returns flag = false, and leaves status undefined.</pre>				

Unofficial Draft for Comment Only

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

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

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

MPI_PROBE(source, tag, comm, status)

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

C binding

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

Fortran 2008 binding

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

Fortran binding

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

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

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

The MPI implementation of MPI_PROBE and MPI_IPROBE needs to guarantee progress: 43 if a call to MPI_PROBE has been issued by a process, and a send that matches the probe 44 has been initiated by some process, then the call to MPI_PROBE will return, unless the 45 message is received by another concurrent receive operation (that is executed by another 46 thread at the probing process). Similarly, if a process busy waits with MPI_IPROBE and a 47 matching message has been issued, then the call to MPI_IPROBE will eventually return flag 48

Unofficial Draft for Comment Only

1

2

3

4

5 6

 $\overline{7}$

8 9

10

11

12

13 14

15

16

17

18

19 20

29

30

31

32

33

34

35

36 37

38

39

40

41

```
1
     = true unless the message is received by another concurrent receive operation or matched
\mathbf{2}
     by a concurrent matched probe.
3
     Example 3.18 Use probe to wait for an incoming message.
4
5
         CALL MPI_COMM_RANK(comm, rank, ierr)
6
         IF (rank .EQ. 0) THEN
7
             CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
8
         ELSE IF (rank .EQ. 1) THEN
9
             CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
10
         ELSE IF (rank .EQ. 2) THEN
11
            DO i=1,2
12
                CALL MPI_PROBE(MPI_ANY_SOURCE, 0, &
13
                                comm, status, ierr)
14
                IF (status(MPI_SOURCE) .EQ. 0) THEN
15
                   CALL MPI_RECV(i, 1, MPI_INTEGER, 0, 0, comm, status, ierr)
     100
16
                ELSE
17
                   CALL MPI_RECV(x, 1, MPI_REAL, 1, 0, comm, status, ierr)
     200
18
                END IF
19
             END DO
20
         END IF
21
22
     Each message is received with the right type.
23
     Example 3.19 A similar program to the previous example, but now it has a problem.
^{24}
25
     /* ------ THIS EXAMPLE IS ERRONEOUS ------ */
26
         CALL MPI_COMM_RANK(comm, rank, ierr)
27
         IF (rank .EQ. 0) THEN
28
             CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
29
         ELSE IF (rank .EQ. 1) THEN
30
             CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
31
         ELSE IF (rank .EQ. 2) THEN
32
             DO i=1,2
33
                CALL MPI_PROBE(MPI_ANY_SOURCE, 0, &
34
                                comm, status, ierr)
35
                IF (status(MPI_SOURCE) .EQ. 0) THEN
36
     100
                   CALL MPI_RECV(i, 1, MPI_INTEGER, MPI_ANY_SOURCE, &
37
                                   0, comm, status, ierr)
38
                ELSE
39
     200
                   CALL MPI_RECV(x, 1, MPI_REAL, MPI_ANY_SOURCE, &
40
                                   0, comm, status, ierr)
41
                END IF
42
             END DO
43
         END IF
44
45
         In Example 3.19, the two receive calls in statements labeled 100 and 200 in Example 3.18
46
```

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.

Unofficial Draft for Comment Only

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

3.8.2 Matching Probe

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

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

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

IN	source	rank of source or MPI_ANY_SOURCE (integer)	37 38
IN	tag	message tag or $MPI_ANY_TAG\xspace$ (integer)	39
IN	comm	communicator (handle)	40
OUT	flag	true if there is a matching message that can be	41
	5	received (logical)	42
OUT	message	returned message (handle)	43
	0	0 ()	44
OUT	status	status object (status)	45
			46

C binding

int MPI_Improbe(int source, int tag, MPI_Comm comm, int *flag,

1

 $\mathbf{2}$

3

4

5

6 7

8

9

10

11

12

13

14

15

16

17

18

19

20

21 22

23 24

25

26

27

28

29

30

 31

32

33

34 35

36

47

MPI_Message *message, MPI_Status *status)	
Fortran 2008 binding	
<pre>MPI_Improbe(source, tag, comm, flag, message, status, ierror) INTEGER, INTENT(IN) :: source, tag TYPE(MPI_Comm), INTENT(IN) :: comm LOGICAL, INTENT(OUT) :: flag</pre>	
TYPE(MPI_Message), INTENT(OUT) :: message TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror	
Fortran binding	
MPI_IMPROBE(SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS, IERROR) INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR LOGICAL FLAG	
MPI_IMPROBE(source, tag, comm, flag, message, status) returns flag = true if th	ere is
a message that can be received and that matches the pattern specified by the argum	
source, tag, and comm. The call matches the same message that would have been rec	
by a call to MPI_RECV(, source, tag, comm, status) executed at the same point i	
program and returns in status the same value that would have been returned by MPI_R	
In addition, it returns in message a handle to the matched message. Otherwise, the	e call
returns flag = false, and leaves status and message undefined. MPI_IMPROBE is a local procedure. According to the definitions in Section 2.4.5) and
in contrast to MPI_IPROBE, it is a nonblocking procedure because it is the initializati	
a matched receive operation.	.011 01
A matched receive (MPI_MRECV or MPI_IMRECV) executed with the message	han-
dle will receive the message that was matched by the probe. Unlike MPI_IPROB other probe or receive operation may match the message returned by MPI_IMPR	E, no
Each message returned by MPI_IMPROBE must be received with either MPI_MREC	CV or
MPI_IMRECV.	
The source argument of MPI_IMPROBE can be MPI_ANY_SOURCE, and the tag ment can be MPI_ANY_TAG, so that one can probe for messages from an arbitrary s	ource
and/or with an arbitrary tag. However, a specific communication context must be pro with the comm argument.	viaea
A synchronous send operation that is matched with MPI_IMPROBE or MPI_MPF	ROBE
will complete successfully only if both a matching receive is posted with MPI_MREC	
MPI_IMRECV, and the receive operation has started to receive the message sent b	
synchronous send.	<i>,</i>
There is a special predefined message: MPI_MESSAGE_NO_PROC, which is a me	ssage
which has MPI_PROC_NULL as its source process. The predefined constant	
MPI_MESSAGE_NULL is the value used for invalid message handles.	
A matching probe with source = MPI_PROC_NULL returns flag = true, message =	
MPI_MESSAGE_NO_PROC, and the status object returns source = MPI_PROC_NULL, t MPI_ANY_TAC_ and count = 0 , see Section 2.10. It is not necessary to call MPI_MPE	-
MPI_ANY_TAG, and count = 0; see Section 3.10. It is not necessary to call MPI_MREC MPI_IMRECV with MPI_MESSAGE_NO_PROC, but it is not erroneous to do so.	∠v or
WE I_INITECV WITH WEI_WESSAGE_NO_FROC, but it is not entoneous 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.)

			4	
MPI_MPROBE(source, tag, comm, message, status)				
IN	source	rank of source or MPI_ANY_SOURCE (integer)	6 7	
IN	tag	message tag or MPI_ANY_TAG (integer)	8	
IN	comm	communicator (handle)	9	
			10 11	
OUT	message	returned message (handle)	11	
OUT	status	status object (status)	13	
C binding	v		14	
	-	g, MPI_Comm comm, MPI_Message *message,	15	
1110 111 1_11	MPI_Status *status)	5, <u>1_</u> 00	16	
Fortron 9	008 binding		17 18	
	e(source, tag, comm, mess	sage, status, ierror)	19	
-	ER, INTENT(IN) :: source	-	20	
TYPE(MPI_Comm), INTENT(IN) ::	comm	21	
	MPI_Message), INTENT(OUT)) :: message	22	
	MPI_Status) :: status		23 24	
INTEG	ER, OPTIONAL, INTENT(OUT)) :: lerror	24 25	
Fortran binding				
	E(SOURCE, TAG, COMM, MESS		27	
		SSAGE, STATUS(MPI_STATUS_SIZE), IERROR	28	
		PROBE except that it is a blocking call that returns	29 30	
only after a matching message has been found. The implementation of MPI_MPROBE and MPI_IMPROBE needs to guarantee progress				
	e way as in the case of MPI_F	· · ·	31 32	
According to the definitions in Section 2.4.2, MPI_MPROBE is incomplete. It is also a				
non-local p		·	34	
Advi	ce to users. This is one of t	he exceptions in which incomplete procedures are	35	
	ocal. (End of advice to users.		36 37	
		,	38	
3.8.3 Ma	tched Receives		39	
The function	ons MPL MRECV and MPL IM	IRECV receive messages that have been previously	40	
	y a matching probe (Section 3		41	
			42 43	
			43	
			45	
			46	
			47	
			48	

1 $\mathbf{2}$

3

```
1
     MPI_MRECV(buf, count, datatype, message, status)
2
       OUT
                 buf
                                             initial address of receive buffer (choice)
3
       IN
                 count
                                             number of elements in receive buffer (non-negative
4
                                             integer)
5
6
       IN
                                             datatype of each receive buffer element (handle)
                 datatype
7
       INOUT
                 message
                                             message (handle)
8
       OUT
                 status
                                             status object (status)
9
10
11
     C binding
     int MPI_Mrecv(void *buf, int count, MPI_Datatype datatype,
12
                    MPI_Message *message, MPI_Status *status)
13
14
     int MPI_Mrecv_c(void *buf, MPI_Count count, MPI_Datatype datatype,
15
                    MPI_Message *message, MPI_Status *status)
16
17
     Fortran 2008 binding
18
     MPI_Mrecv(buf, count, datatype, message, status, ierror)
19
          TYPE(*), DIMENSION(..) :: buf
          INTEGER, INTENT(IN) :: count
20
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
21
          TYPE(MPI_Message), INTENT(INOUT) :: message
22
23
          TYPE(MPI_Status) :: status
^{24}
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
     MPI_Mrecv(buf, count, datatype, message, status, ierror)
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
32
33
     Fortran binding
34
     MPI_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR)
35
          <type> BUF(*)
36
          INTEGER COUNT, DATATYPE, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR
37
          This call receives a message matched by a matching probe operation (Section 3.8.2).
38
          The receive buffer consists of the storage containing count consecutive elements of the
39
     type specified by datatype, starting at address buf. The length of the received message must
40
     be less than or equal to the length of the receive buffer. An overflow error occurs if all
41
     incoming data does not fit, without truncation, into the receive buffer.
42
         If the message is shorter than the receive buffer, then only those locations corresponding
43
     to the (shorter) message are modified.
44
          On return from this function, the message handle is set to MPI_MESSAGE_NULL. All
45
```

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

	J J J J J J J J J J J J J J J J J J J	
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

Fortran 2008 binding	24
MPI_Imrecv(buf, count, datatype, message, request, ierror)	25
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf	26
INTEGER, INTENT(IN) :: count	27
TYPE(MPI_Datatype), INTENT(IN) :: datatype	28
TYPE(MPI_Message), INTENT(INOUT) :: message	29
TYPE(MPI_Request), INTENT(OUT) :: request	30
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	31
MPI_Imrecv(buf, count, datatype, message, request, ierror)	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 INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_IMRECV(BUF, COUNT, DATATYPE, MESSAGE, REQUEST, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, MESSAGE, REQUEST, IERROR

TYPE(MPI_Message), INTENT(INOUT) :: message

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.

 $\mathbf{2}$

1	If MPI_IMRECV is called with MPI_MESSAGE_NO_PROC as the message argument, the		
2	call returns immediately with a request object which, 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		
11	a 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			
15	3.8.4 Cancel		
16			
17			
18	MPI_CANCEL(request)		
19			
20	IN request communication request (handle)		
21			
22	C binding		
23	<pre>int MPI_Cancel(MPI_Request *request)</pre>		
24	Fortran 2008 binding		
25	MPI_Cancel(request, ierror)		
26	TYPE(MPI_Request), INTENT(IN) :: request		
27	INTEGER, OPTIONAL, INTENT(OUT) :: ierror		
28	Fortran binding		
29 30	MPI_CANCEL(REQUEST, IERROR)		
30 31	INTEGER REQUEST, IERROR		
31	TRIEGER REQUERT, TERROR		
33	A call to MPI_CANCEL marks for cancellation a pending, nonblocking communica-		
24	tion operation (send or receive). Cancelling a send request by calling MPI_CANCEL is		

tion operation (send or receive). Cancelling a send request by calling MPI_CANCEL is 34deprecated. 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 operations) 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 request (see ⁴⁴ Section 3.9), in the same way it is used for nonpersistent requests. Cancelling a persistent ⁴⁵ send request by calling MPI_CANCEL is deprecated. A successful cancellation cancels the ⁴⁶ active communication, but not the request itself. After the call to MPI_CANCEL and the ⁴⁷ subsequent call to MPI_WAIT or MPI_TEST, the request becomes inactive and can be ⁴⁸ activated for a new communication. The successful cancellation of a buffered send frees the buffer space occupied by the pending message. Cancelling a buffered send request by calling MPI_CANCEL is deprecated.

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

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

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

			19
MPL TES	T_CANCELLED(status, flag)		20
1011 1_1 120			21
IN	status	status object (status)	22
OUT	flag	true if the operation has been cancelled (logical)	23
	0		24
C bindin	σ		25
	0	[Status watatus int wflam)	26
IIIC MPI_	rest_cancerred(const MP)	I_Status *status, int *flag)	27
Fortran 2008 binding			
MPI_Test_cancelled(status, flag, ierror)			29
TYPE(MPI_Status), INTENT(IN) :: status 3			30
LOGICAL, INTENT(OUT) :: flag			31
INTE	GER, OPTIONAL, INTENT(OU	JT) :: ierror	32
	33		
Fortran binding			34
	MPI_TEST_CANCELLED(STATUS, FLAG, IERROR)		
	GER STATUS(MPI_STATUS_S)	IZE), IERROR	36
LOGI	CAL FLAG		37
D .			

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 $_{47}$ transferred to the receiver before a matching receive is posted), then the cancellation $_{48}$

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6 7

8 9

10

11 12

13

14

15

16

17 18

38

39

40

41

42 43

44

 $45 \\ 46$

of this send may require communication with the intended receiver in order to free allocated buffers. On some systems this may require an interrupt to the intended receiver. Note that, while communication may be needed to implement

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

3.9 Persistent Communication Requests

11 Often a communication with the same argument list (with the exception of the buffer con-12tents) 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-14nication arguments to a **persistent** communication request once and then repeatedly using 15the request to initiate and complete operations. In the case of point-to-point communica-16tion, the persistent request thus created can be thought of as a communication port or a 17"half-channel." It does not provide the full functionality of a conventional channel, since 18 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 20not of the overhead for communication between one communication controller and another. 21It is not necessary that messages sent with a persistent point-to-point request be received 22by 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 and Section 8.8. The remainder of this section covers the point-to-point persistent initialization operations and the start routines, which are used for both persistent point-to-point and persistent collective communication operations.

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

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

			 c . ,
32 33	IN	buf	initial address of send buffer (choice)
34	IN	count	number of elements sent (non-negative integer)
35	IN	datatype	type of each element (handle)
36 37	IN	dest	rank of destination (integer)
38	IN	tag	message tag (integer)
39	IN	comm	communicator (handle)
40	OUT	request	communication request (handle)
41	001	request	communication request (nanale)
42			

```
C binding
```

44	<pre>int MPI_Send_init(const void *buf, int count, MPI_Datatype datatype,</pre>
45	<pre>int dest, int tag, MPI_Comm comm, MPI_Request *request)</pre>
46	<pre>int MPI_Send_init_c(const void *buf, MPI_Count count,</pre>
47	MPI_Datatype datatype, int dest, int tag, MPI_Comm comm,
48	MPI_Request *request)

Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8 9

10

28

29 30 31

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

Unofficial Draft for Comment Only

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

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_Comm), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

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

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

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

C binding

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)
 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

Unofficial Draft for Comment Only

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

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 request is initiated 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 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) 39			
IN	count	list length (non-negative integer)	40
INOUT	array_of_requests	array of requests (array of handles)	41
INCOT	allay_01_lequests	array of requests (array of nandles)	42 43
C binding			
int MPI_Startall(int count, MPI_Request array_of_requests[]) 44			
Fortran 2008 binding			
MPI_Startall(count, array_of_requests, ierror)			47
Mri_Startari(Count, array_or_requests, rerior)			48

2

4

5

6

7

8

9 10 11

12

13 14

15

1617

18

19

20

21

22

23

2425

26

27

28

29

30

 31

32

33

34

35

36

37 38

1	INTEGER, INTENT(IN) :: count
2	<pre>TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)</pre>
3	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4	Fortran binding

6 MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)

INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR

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.

¹¹ A communication started with a call to MPI_START or MPI_STARTALL is completed ¹² by a call to MPI_WAIT, MPI_TEST, or one of the derived functions described in Sec-¹³ tion 3.7.5. The request becomes inactive after successful completion of such call. The re-¹⁴ quest is not deallocated and it can be activated anew by an MPI_START or MPI_STARTALL ¹⁵ call.

16A persistent request is deallocated by a call to MPI_REQUEST_FREE (Section 3.7.3). 17The call to MPI_REQUEST_FREE can occur at any point in the program after the persis-18 tent request was created. However, the request will be deallocated only after it becomes 19inactive. Active receive requests should not be freed. Otherwise, it will not be possible to 20check that the receive has completed. Collective operation requests (defined in Section 6.1221and Section 8.7 for nonblocking collective operations, and Section 6.13 and Section 8.8 for 22persistent collective operations) must not be freed while active. It is preferable, in general, 23to free requests when they are inactive. If this rule is followed, then the functions described 24 in this section will be invoked in a sequence of the form, 25

Create (Start Complete)* Free

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

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

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 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 non-circular shift done with calls to send-receive.

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

Unofficial Draft for Comment Only

7 8

9

10

26 27

28

29

30

 31

32

33 34

35

36

37 38

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.



Chapter 4

Partitioned Point-to-Point Communication

4.1 Introduction

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

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

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

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

 24

 $45 \\ 46$

CHAPTER 4. PARTITIONED POINT-TO-POINT COMMUNICATION

1 2

4.2 Semantics of Partitioned Point-to-Point Communication

MPI guarantees certain general properties of partitioned point-to-point communication
 ⁴ 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

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

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

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

 $\frac{24}{25}$

26

27

 31

32

33 34

35

36

37

38 39

40

41 42 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 implementa tion, making performance concerns of a large number of sender-side partitions poten tially less impactful than receiver-side granularity. (*End of advice to users.*)

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

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

Example 4.1 Simple partitioned communication example.

- $43 \\ 44$
- 45

46

ſ

```
1
#include "mpi.h"
                                                                                       \mathbf{2}
#define PARTITIONS 8
                                                                                       3
#define COUNT 5
int main(int argc, char *argv[])
                                                                                       4
                                                                                       5
  double message[PARTITIONS*COUNT];
                                                                                       6
  MPI_Count partitions = PARTITIONS;
  int source = 0, dest = 1, tag = 1, flag = 0;
                                                                                       9
  int myrank, i;
                                                                                       10
  int provided;
                                                                                       11
  MPI_Request request;
  MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZED, &provided);
                                                                                       12
  if (provided < MPI_THREAD_SERIALIZED)
                                                                                       13
                                                                                       14
     MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
                                                                                       15
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
                                                                                       16
  if (myrank == 0)
                                                                                       17
  {
                                                                                       18
     MPI_Psend_init(message, partitions, COUNT, MPI_DOUBLE, dest, tag,
                MPI_COMM_WORLD, MPI_INFO_NULL, &request);
                                                                                       19
     MPI_Start(&request);
                                                                                      20
                                                                                      21
     for(i = 0; i < partitions; ++i)</pre>
                                                                                      22
     {
        /* compute and fill partition #i, then mark ready: */
                                                                                      23
                                                                                       ^{24}
        MPI_Pready(i, &request);
                                                                                       25
     }
                                                                                       26
     while(!flag)
                                                                                      27
     {
        /* do useful work #1 */
                                                                                      28
                                                                                      29
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
                                                                                      30
        /* do useful work #2 */
                                                                                       31
     }
                                                                                       32
     MPI_Request_free(&request);
                                                                                       33
  }
                                                                                      34
  else if (myrank == 1)
                                                                                      35
  {
     MPI_Precv_init(message, partitions, COUNT, MPI_DOUBLE, source, tag,
                                                                                      36
                                                                                      37
                MPI_COMM_WORLD, MPI_INFO_NULL, &request);
                                                                                       38
     MPI_Start(&request);
                                                                                       39
     while(!flag)
                                                                                       40
     {
                                                                                       41
        /* do useful work #1 */
                                                                                      42
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
        /* do useful work #2 */
                                                                                       43
                                                                                       44
     }
     MPI_Request_free(&request);
                                                                                       45
  }
                                                                                       46
                                                                                       47
  MPI_Finalize();
                                                                                       48
  return 0;
```

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 called on a request before transferring data. However, this modality of communication gives MPI implementations far more flexibility in data movement than non-partitioned communications. (*End of rationale.*)

4.2.1 Communication Initialization and Starting with Partitioning

Initialization of partitioned communication operations use the initialization calls described below. Subsequent to initialization, MPI_START/MPI_STARTALL are used as the first indication to MPI that a message transfer will occur. For send-side operations, neither initializing nor starting the operation enables transfer of any part of the user buffer. Freeing or canceling a partitioned communication request that is active (i.e., initialized and started) and not completed is erroneous. After the partitioned communication operation is started, individual partitions of a message are indicated as ready to be sent by MPI via the MPI_PREADY function, described below. 24

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

	_	= $($ $,$ $)$ $($ $,$ $)$ $($ $,$ $)$		
27 28	IN	buf	initial address of send buffer (choice)	
28	IN	partitions	number of partitions (non-negative integer)	
30	IN	count	number of elements sent per partition (non-negative	
31			integer)	
32 33	IN	datatype	type of each element (handle)	
33 34	IN	dest	rank of destination (integer)	
35	IN	tag	message tag (integer)	
36 37	IN	comm	communicator (handle)	
37	IN	info	info argument (handle)	
39	OUT	request	communication request (handle)	
40				
41	C binding			
42	² int MPI_Psend_init(const void *buf, int partitions, MPI_Count count,			
43	•			
44	MPI_Info info, MPI_Request *request)			
45				
46		2008 binding		
47				
48		request, ierror)		

}

1

 $\mathbf{2}$

4

5

6

10

11

12

13

14

1516

17

18

19 20

21

22

23

24

25

26

27

28

48

TYPE(*), DIMENSION(..), INTENT(IN) :: buf INTEGER, INTENT(IN) :: partitions, dest, tag INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count 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

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, dest, 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 sent per partition (non-negative integer)		
IN	datatype	type of each element (handle)		
IN	dest	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,				

Ν

106

1		MPI_Datatype datatype	e, int dest, int tag, MPI_Comm comm,	
2		MPI_Info info, MPI_R	equest *request)	
3	Fortran 2008 binding			
4		0	ount, datatype, dest, tag, comm, info,	
5 6		request, ierror)		
7	TYPE((*), DIMENSION(), INTENT	(IN) :: buf	
8	INTEGER, INTENT(IN) :: partitions, dest, tag			
9	INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count			
10	TYPE(MPI_Datatype), INTENT(IN) :: datatype			
11	TYPE(MPI_Comm), INTENT(IN) :: comm			
12		<pre>MPI_Info), INTENT(IN) ::</pre>		
13		MPI_Request), INTENT(OUT)	-	
14	INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	
15	Fortran b	binding		
16	MPI_PRECV	_INIT(BUF, PARTITIONS, CO	DUNT, DATATYPE, DEST, TAG, COMM, INFO,	
17		REQUEST, IERROR)		
18	• 1	> BUF(*)		
19			DEST, TAG, COMM, INFO, REQUEST, IERROR	
20 21	INTEG	ER(KIND=MPI_COUNT_KIND) (COUNT	
21	י ת			
23		<i>onale.</i> The info argument is ation-defined info keys. (<i>End</i>	provided in order to support per-operation imple-	
24	ment	ation-defined into keys. (End	of fationale.)	
25		MPI_PRECV_INIT creates a partitioned communication receive request and binds to it		
26	all the arguments of a partitioned receive operation. This operation can only match with			
27	partitioned communication initialization operations, therefore the MPI library is required to			
28	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)			
29	with communicator, tag, and source dictating message matching. In the event that the			
30	communicator, tag, and source do not uniquely identify a message, the order in which			
31	partitioned communication initialization calls are made is the order in which they will			
32 33	eventually match. Partitioned communication initialization calls are local. That is,			
33 34	MPI_PRECV_INIT may return before the operation completes. It is erroneous to provide a			
35	partitions value ≤ 0 . Wildcards for source and tag are not allowed.			
36	Advice to implementors. Unlike MPI_RECV_INIT, MPI_PRECV_INIT may communi-			
37	cate. Also unlike MPI_RECV_INIT, MPI_PRECV_INIT takes an info argument. (<i>End</i>			
38	of advice to implementors.)			
39	57 444			
40				
41	MPI_PREA	ADY(partition, request)		
42	IN	partition	partition to mark ready for transfer (non-negative	
43	IIN	partition	integer)	
44		roquest		
45 46	INOUT	request	partitioned communication request (handle)	
46 47	Chindin	71		
48	C binding	<pre>c binding int MPI_Pready(int partition, MPI_Request *request)</pre>		
	Int in I_IIGady (Int partition, in I_nequest *Iequest)			

Unofficial Draft for Comment Only

```
1
Fortran 2008 binding
                                                                                           \mathbf{2}
MPI_Pready(partition, request, ierror)
                                                                                           3
    INTEGER, INTENT(IN) :: partition
    TYPE(MPI_Request), INTENT(INOUT) :: request
                                                                                           4
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                           5
                                                                                           6
Fortran binding
MPI_PREADY(PARTITION, REQUEST, IERROR)
    INTEGER PARTITION, REQUEST, IERROR
                                                                                           9
                                                                                          10
    MPI_PREADY is a send-side call that indicates that a given partition is ready to be
                                                                                          11
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
                                                                                          12
                                                                                          13
MPI_PSEND_INIT call. Partition numbering starts at zero and ranges to one less than
                                                                                          14
the number of partitions declared in the MPI_PSEND_INIT call. Specifying a partition
                                                                                          15
number that is equal to or larger than the number of partitions is erroneous. After a call
                                                                                          16
to MPI_START/MPI_STARTALL, all partitions associated with that operation are inactive.
                                                                                          17
A call to MPI_PREADY marks the indicated partition as active. Calling MPI_PREADY on
                                                                                          18
an active partition is erroneous.
                                                                                          19
                                                                                          20
MPI_PREADY_RANGE(partition_low, partition_high, request)
                                                                                          21
                                                                                          22
  IN
            partition_low
                                       lowest partition ready for transfer (non-negative
                                                                                          23
                                        integer)
                                                                                          ^{24}
  IN
            partition_high
                                        highest partition ready for transfer (non-negative
                                                                                          25
                                        integer)
                                                                                          26
  INOUT
            request
                                        partitioned communication request (handle)
                                                                                          27
                                                                                          28
C binding
                                                                                          29
int MPI_Pready_range(int partition_low, int partition_high,
                                                                                          30
                                                                                          31
               MPI_Request *request)
                                                                                          32
Fortran 2008 binding
                                                                                          33
MPI_Pready_range(partition_low, partition_high, request, ierror)
                                                                                          34
    INTEGER, INTENT(IN) :: partition_low, partition_high
                                                                                          35
    TYPE(MPI_Request), INTENT(INOUT) :: request
                                                                                          36
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                          37
                                                                                          38
Fortran binding
                                                                                          39
MPI_PREADY_RANGE(PARTITION_LOW, PARTITION_HIGH, REQUEST, IERROR)
                                                                                          40
    INTEGER PARTITION_LOW, PARTITION_HIGH, REQUEST, IERROR
                                                                                          41
    A call to MPI_PREADY_RANGE has the same effect as calls to
                                                                                          42
MPI_PREADY, executed for i=partition_low, ..., partition_high, in some arbitrary order.
                                                                                          43
Calls to MPI_PREADY_RANGE follow the same rules as those for MPI_PREADY calls.
                                                                                          44
                                                                                          45
                                                                                          46
                                                                                          47
```

	108	CHAPTER 4.	PARTITI	ONED POINT-TO-POINT COMMUNICATION
1	MPI_PREA	DY_LIST(length, ar	ray_of_part	itions, request)
2	IN	length		list length (integer)
3 4	IN	array_of_partitions		array of partitions (array of non-negative integers)
5	INOUT	request		partitioned communication request (handle)
6				
7 8 9	C binding int MPI_P		0	<pre>nst int array_of_partitions[],</pre>
10 11 12 13 14 15	MPI_Pread INTEG TYPE(• •	: length, TENT(INOUT	-
16 17 18 19		Y_LIST(LENGTH, A		ARTITIONS, REQUEST, IERROR) FIONS(*), REQUEST, IERROR
20 21 22 23 24	MPI_PREA	DY, executed for the _of_partitions[courd	he partition $nt - 1$ of the second	the same effect as calls to ns specified in the range <i>array_of_partitions</i> [0] ne array_of_partitions, executed in some arbitrary w the same rules as those for MPI_PREADY calls.
25	4.2.2 Coi	mmunication Comp	letion unde	r Partitioning
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40	communica the sender subsequent the user ca partitioned does not in The co indicates to partial reco function ca received in	ation operation. The is now free to call ly MPI_PREADY, M operation. For the adjoint that the part operation of a part: that the receive buffle eption of the receive an be used to determ to the receive buffle	e completion MPI_STAR IPI_PREAD rtitioned consending pro- citions of the itioned receptor er contains buffer is pro- nine if the re- pring the re-	(and variants) are used to complete a partitioned on of a partitioned send operation indicates that RT/MPI_STARTALL to restart the operation and DY_RANGE or MPI_PREADY_LIST. Alternatively, ommunication request after the completion of the press, completion of the partitioned send operation are message have all been received. eive operation through MPI_WAIT or MPI_TEST all of the partitions. A function for probing the rovided by MPI_PARRIVED. The MPI_PARRIVED message data for the indicated partition has been success, the receiver becomes free to access the that previously completed for that operation).
40 41 42 43 44 45 46 47 48				

MPI_PARRIVED(request, partition, flag)				
INOUT	request	partitioned communication request (handle)		
IN	partition	partition to be tested (non-negative integer)		
OUT	flag	true if operation completed on the specified partition,		
		false if not (logical)		
C binding	r			

C binding

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

Fortran 2008 binding

```
MPI_Parrived(request, partition, flag, ierror)
   TYPE(MPI_Request), INTENT(INOUT) :: request
   INTEGER, INTENT(IN) :: partition
   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 operation. A subsequent MPI_TEST/MPI_WAIT operation is required to complete the message, 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.

Semantics of Communications in Partitioned Mode 4.2.3

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.

4647

48

109

1 $\mathbf{2}$ 3

5 6

9

10

11

12

13

14

15

1617

18

19

20

21

22

23

 24

25

26

27

28

29

30 31

32 33

34

3536

37

38

39

40

41

42

43

44

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 like in the
 previous example, but this example uses multiple threads on the send-side. Note that the
 MPI_PSEND_INIT and MPI_PRECV_INIT functions match each other like in the previous
 example.

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

```
22
     #include "mpi.h"
23
     #define NUM_THREADS 8
^{24}
     #define PARTITIONS 8
25
     #define PARTLENGTH 16
26
     int main(int argc, char *argv[]) /* same send/recv partitioning */
27
     Ł
28
       double message[PARTITIONS*PARTLENGTH];
29
       int partitions = PARTITIONS;
30
       int partlength = PARTLENGTH;
31
       int count = 1, source = 0, dest = 1, tag = 1, flag = 0;
32
       int myrank;
33
       int provided;
34
       MPI_Request request;
35
       MPI_Info info = MPI_INFO_NULL;
36
       MPI_Datatype xfer_type;
37
       MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
38
       if (provided < MPI_THREAD_MULTIPLE)</pre>
39
          MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
40
       MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
41
       MPI_Type_contiguous(partlength, MPI_DOUBLE, &xfer_type);
42
       MPI_Type_commit(&xfer_type);
43
       if (myrank == 0)
                            /* code for process zero */
44
       {
45
          MPI_Psend_init(message, partitions, count, xfer_type, dest, tag,
46
                info, MPI_COMM_WORLD, &request);
47
          MPI_Start(&request);
48
```

6 7

8 9

10

11 12

13

19

```
#pragma omp parallel for shared(request) num_threads(NUM_THREADS)
     for (int i=0; i<partitions; i++)</pre>
     {
         /* compute and fill partition #i, then mark ready: */
        MPI_Pready(i, &request);
     }
     while(!flag)
     {
                                                                                         10
         /* Do useful work */
                                                                                         11
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
         /* Do useful work */
                                                                                         12
     }
                                                                                         13
                                                                                         14
     MPI_Request_free(&request);
                                                                                         15
  }
                                                                                         16
  else if (myrank == 1) /* code for process one */
                                                                                         17
  {
     MPI_Precv_init(message, partitions, count, xfer_type, source, tag,
                                                                                         18
                                                                                         19
            info, MPI_COMM_WORLD, &request);
     MPI_Start(&request);
                                                                                         20
                                                                                         21
     while(!flag)
                                                                                         22
     {
                                                                                         23
         /* Do useful work */
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
                                                                                         ^{24}
                                                                                         25
         /* Do useful work */
                                                                                         26
     }
     MPI_Request_free(&request);
                                                                                         27
  }
                                                                                         28
                                                                                         29
  MPI_Finalize();
                                                                                         30
  return 0;
                                                                                         31
}
                                                                                         32
                                                                                         33
4.3.2 Send-only Partitioning Example with Tasks and OpenMP version 4.0 or later
                                                                                         34
The previous example is tailored specifically for send-side partitioning using threads. This
                                                                                         35
is an example where parallel task producers produce input to part of an overall buffer; they
                                                                                         36
complete in any order and contribute to the overall buffer.
                                                                                         37
                                                                                         38
Example 4.3 Parallel task producers for partitioned communication using threads.
                                                                                         39
                                                                                         40
                                                                                         41
#include "mpi.h"
                                                                                         42
#define NUM_THREADS 8
#define NUM_TASKS 64
                                                                                         43
                                                                                         44
#define PARTITIONS NUM_TASKS
#define PARTLENGTH 16
                                                                                         45
                                                                                         46
#define MESSAGE_LENGTH PARTITIONS*PARTLENGTH
```

int main(int argc, char *argv[]) /* send-side partitioning */

{

1 $\mathbf{2}$

5

6

9

47

112

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

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

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

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

Example 4.4 Partitioned communication receive-side partial completion.

```
#include "mpi.h"
#define NUM_THREADS 64
#define PARTITIONS NUM THREADS
#define PARTLENGTH 16
#define MESSAGE_LENGTH PARTITIONS*PARTLENGTH
int main(int argc, char *argv[]) /* send-side partitioning */
{
  double message [MESSAGE_LENGTH];
  int send_partitions = PARTITIONS,
      send_partlength = PARTLENGTH,
      recv_partitions = PARTITIONS*2,
      recv_partlength = PARTLENGTH/2;
  int source = 0, dest = 1, tag = 1, flag = 0;
  int myrank;
  int provided;
  MPI_Request request;
```

1 2

5 6

9 10

11

12

13

14 15

16 17

18

19

20

21

22

23

24

25

26

27

28

29 30

31 32 33

34

35 36

37

38 39

40

41

42

43 44

45

46

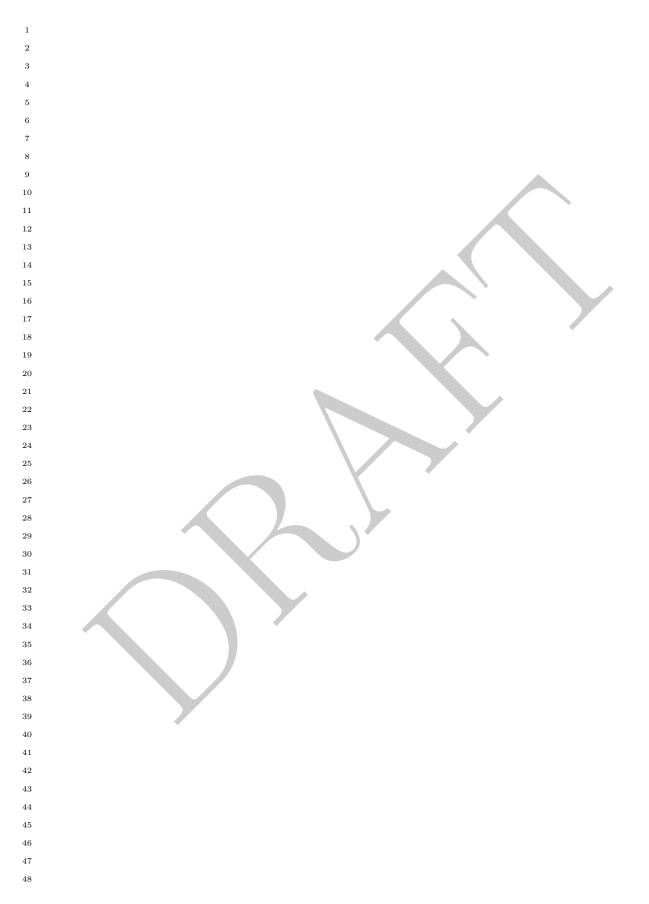
47

114

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

```
MPI_Parrived(&request, j+1, &flag);
              if(flag && part2_complete == 0)
              {
                 part2_complete++;
                 /* Do work using partition j+1 */
              }
           }
           else {
               part2_complete++;
           }
        }
     }
     while(!flag)
     {
        /* Do useful work */
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
        /* Do useful work */
     }
     MPI_Request_free(&request);
  }
 MPI_Finalize();
 return 0;
}
```

 $15 \\ 16$



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

 24

 31

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

6 7

1

 $\mathbf{2}$

3

4

5

8 9

10

11 12

13

14

15

16

17

18

 $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

```
Typesiq = \{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 non-negative integers. If the value is zero, no elements are generated in the type map and there is no effect on datatype bounds or extent.

¹⁹ We can use a handle to a general datatype as an argument in a send or receive operation, ²⁰ instead of a basic datatype argument. The operation MPI_SEND(buf, 1, datatype,...) will ²² use the send buffer defined by the base address buf and the general datatype associated ²³ with datatype; it will generate a message with the type signature determined by the datatype ²⁴ argument. MPI_RECV(buf, 1, datatype,...) will use the receive buffer defined by the base ²⁵ 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

11 / m

38 39

32

33

34 35 36

37

$$lb(Typemap) = \min_{j} aisp_{j},$$

$$ub(Typemap) = \max_{j} (disp_{j} + \text{sizeof}(type_{j})) + \epsilon, \text{ and}$$

$$extent(Typemap) = ub(Typemap) - lb(Typemap).$$
(5.1)

40 41 42

⁴³ If $type_j$ requires alignment to a byte address that is a multiple of k_j , then ϵ is the least ⁴⁴ non-negative increment needed to round extent(Typemap) to the next multiple of $\max_j k_j$. ⁴⁵ In Fortran, it is implementation dependent whether the MPI implementation computes ⁴⁶ the alignments k_j according to the alignments used by the compiler in common blocks, ⁴⁷ SEQUENCE derived types, BIND(C) derived types, or derived types that are neither SEQUENCE ⁴⁸ nor BIND(C). The complete definition of **extent** is given by Equation 5.1 Section 5.1.

Unofficial Draft for Comment Only

Let

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

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

5.1.1 Type Constructors with Explicit Addresses

In Fortran, the functions MPI_TYPE_CREATE_HVECTOR,
MPI_TYPE_CREATE_HINDEXED, MPI_TYPE_CREATE_HINDEXED_BLOCK,
MPI_TYPE_CREATE_STRUCT, and MPI_GET_ADDRESS accept arguments of type
INTEGER(KIND=MPI_ADDRESS_KIND), wherever arguments of type MPI_Aint are used in C.
For Fortran compilers that do not support the Fortran 90 KIND notation, and where ad-
dresses 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 lower count consists of these determines constructions with coulisity addresses

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.

			37
MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)			38
IN	count	replication count (non-negative integer)	39
IN	aldtypa	ald datatuma (handla)	40
IIN	oldtype	old datatype (handle)	41
OUT	newtype	new datatype (handle)	42
			43
C bindir	lo.		44
	int MPI_Type_contiguous(int count, MPI_Datatype oldtype,		
_	MPI_Datatype *newty	VI VI	46
	_ 51		47

1 2

3

4

5

6

 $\overline{7}$

8

9

10

11

12

13

14

15

16 17 18

28

29

30

31 32

33 34

35

36

```
1
            int MPI_Type_contiguous_c(MPI_Count count, MPI_Datatype oldtype,
\mathbf{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)
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
19
            newtype is the datatype obtained by concatenating count copies of oldtype. Concatenation
20
            is defined using extent as the size of the concatenated copies.
21
            Example 5.2 Let oldtype have type map \{(double, 0), (char, 8)\}, with extent 16, and let
22
            count = 3. The type map of the datatype returned by newtype is
23
24
                        \{(double, 0), (char, 8), (double, 16), (char, 24), (double, 32), (char, 40)\};
25
26
            i.e., alternating double and char elements, with displacements 0, 8, 16, 24, 32, 40.
27
                      In general, assume that the type map of oldtype is
28
29
                        \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\},\
30
^{31}
            with extent ex. Then newtype has a type map with count \cdot n entries defined by:
32
                   \{(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
33
34
                   ..., (type_0, disp_0 + ex \cdot (count - 1)), \dots, (type_{n-1}, disp_{n-1} + ex \cdot (count - 1))\}.
35
36
            Vector The function MPI_TYPE_VECTOR is a more general constructor that allows repli-
37
            cation of a datatype into locations that consist of equally spaced blocks. Each block is
38
            obtained by concatenating the same number of copies of the old datatype. The spacing
39
            between blocks is a multiple of the extent of the old datatype.
40
41
42
43
44
45
46
47
48
```

MPI_TYPE	_VECTOR(count, blocklength,	stride, oldtype, newtype)	1
IN	count	number of blocks (non-negative integer)	2 3
IN	blocklength	number of elements in each block (non-negative integer)	3 4 5
IN	stride	number of elements between start of each block (integer)	6 7
IN	oldtype	old datatype (handle)	8
OUT	newtype	new datatype (handle)	9 10
			11
C binding			12
int MPI_T	-	blocklength, int stride, MPI_Datatype *newtype)	13 14
int MPI_T	-	unt, MPI_Count blocklength, Datatype oldtype, MPI_Datatype *newtype)	15 16
Dantara 9			17 18
	008 binding	, stride, oldtype, newtype, ierror)	18
01	ER, INTENT(IN) :: count,		20
	MPI_Datatype), INTENT(IN)	•	21
	MPI_Datatype), INTENT(OUT		22
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	23
MPT Type	vector(count, blocklength	, stride, oldtype, newtype, ierror)	24
		INTENT(IN) :: count, blocklength, stride	25 26
	MPI_Datatype), INTENT(IN)	-	20 27
TYPE(1	MPI_Datatype), INTENT(OUT) :: newtype	28
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	29
Fortran b	inding		30
MPI_TYPE_	VECTOR (COUNT, BLOCKLENGTH	, STRIDE, OLDTYPE, NEWTYPE, IERROR)	31
INTEG	ER COUNT, BLOCKLENGTH, ST	RIDE, OLDTYPE, NEWTYPE, IERROR	32
			33
Example	5.3 Assume, again, that old	type has type map $\{(\texttt{double}, 0), (\texttt{char}, 8)\}$, with	34
extent 16.	A call to MPI_TYPE_VECTOR	R(2, 3, 4, oldtype, newtype) will create the datatype	35 36
with type r	nap,		37
{(dou	ble (0) (char 8) (double 16)	(char, 24), (double, 32), (char, 40),	38
((,(,, (,), (,),	39
(doub	ble, 64), (char, 72), (double, 80)	$0), (\texttt{char}, 88), (\texttt{double}, 96), (\texttt{char}, 104) \}.$	40
			41
	b blocks with three copies each veen the the start of each bloc	h of the old type, with a stride of 4 elements $(4 \cdot 16)$ k.	42 43
Fromale		CTOP(2, 1, 2) oldtyng nowtyng) will aposts the	44
datatype,	5.4 A call to WP1_TYPE_VE	ECTOR(3, 1, -2, oldtype, newtype) will create the	45 46
{(dou	$\mathtt{ble},0),(\mathtt{char},8),(\mathtt{double},-3$	$2), (char, -24), (double, -64), (char, -56)\}.$	47
			48

1	In ger	neral, assume that oldtype has	s type map,	
2 3	$\{(ty)$	$pe_0, disp_0), \ldots, (type_{n-1}, disp_n)$	$_{n-1})\},$	
4 5 6	with extence $count \cdot bl \cdot$		a. The newly created datatype has a type map with	
7	$\{(ty_{I})\}$	$pe_0, disp_0), \ldots, (type_{n-1}, disp_n)$	(n-1),	
9	(type	$e_0, disp_0 + ex), \dots, (type_{n-1}, disp_0)$	$lisp_{n-1} + ex), \ldots,$	
10 11	(type	$e_0, disp_0 + (bl - 1) \cdot ex), \dots, (u$	$type_{n-1}, disp_{n-1} + (bl - 1) \cdot ex),$	
12 13	(type	$e_0, disp_0 + stride \cdot ex), \dots, (typ_0)$	$pe_{n-1}, disp_{n-1} + stride \cdot ex), \dots,$	
14 15	(type	$e_0, disp_0 + (stride + bl - 1) \cdot e_2$	$(x), \ldots, (type_{n-1}, disp_{n-1} + (stride + bl - 1) \cdot ex), \ldots, the set of th$	
16 17	(type	$e_0, disp_0 + stride \cdot (count - 1)$	$(ex),\ldots,$	
18 19	(type	$e_{n-1}, disp_{n-1} + stride \cdot (count)$	$(-1) \cdot ex), \ldots,$	
20	$(type_0, disp_0 + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots,$			
21 22	$(type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex)\}.$			
23 24	A call to MPI_TYPE_CONTIGUOUS(count, oldtype, newtype) is equivalent to a call to			
25 26	MPI_TYPE_VECTOR(count, 1, 1, oldtype, newtype), or to a call to MPI_TYPE_VECTOR(1, count, n, oldtype, newtype), where n is an arbitrary integer value.			
27 28 29 30 31 32	Hvector The function MPI_TYPE_CREATE_HVECTOR is identical to MPI_TYPE_VECTOR, except that stride is given in bytes, rather than in elements. The use for both types of vector constructors is illustrated in Section 5.1.14. (H stands for "heterogeneous").			
$33 \\ 34$	MPI_TYP	E_CREATE_HVECTOR(count,	blocklength, stride, oldtype, newtype)	
35	IN	count	number of blocks (non-negative integer)	
36 37	IN	blocklength	number of elements in each block (non-negative integer)	
38 39	IN	stride	number of bytes between start of each block (integer)	
40	IN	oldtype	old datatype (handle)	
41 42	OUT	newtype	new datatype (handle)	
43 44 45 46	C bindin int MPI_7	 Type_create_hvector(int c	ount, int blocklength, MPI_Aint stride, e, MPI_Datatype *newtype)	
47 48	int MPI_1	• 1	_Count count, MPI_Count blocklength, YI_Datatype oldtype, MPI_Datatype *newtype)	

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

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

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

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

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

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

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

In general, assume that oldtype has type map,

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

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

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

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

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

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

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

and

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

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

1 2	MPI_TYPE	_CREATE_HINDEXED(coun oldtype, newtype)	t, array_of_blocklengths, array_of_displacements,
3 4 5 6	IN	count	<pre>number of blocks—also number of entries in array_of_displacements and array_of_blocklengths (non-negative integer)</pre>
7 8	IN	array_of_blocklengths	number of elements in each block (array of non-negative integers)
9	IN	array_of_displacements	byte displacement of each block (array of integers)
10	IN	oldtype	old datatype (handle)
11 12	OUT	newtype	new datatype (handle)
13 14 15 16 17 18		ype_create_hindexed(int	
19 20 21 22		const MPI_Count arra const MPI_Count arra	ay_of_blocklengths[], ay_of_displacements[], e, MPI_Datatype *newtype)
23		008 binding	
24	MPI_Type_	create_hindexed(count, a	
25 26 27 28 29 30 31	INTEG TYPE(I TYPE(I		ents(count) N) :: oldtype JT) :: newtype
32 33 34 35 36 37 38 39	INTEGI TYPE (I TYPE (I	ER(KIND=MPI_COUNT_KIND),	<pre>nts, oldtype, newtype, ierror) , INTENT(IN) :: count, ths(count), array_of_displacements(count) N) :: oldtype JT) :: newtype</pre>
40 41 42 43 44 45	INTEG	CREATE_HINDEXED(COUNT, A ARRAY_OF_DISPLACEME ER COUNT, ARRAY_OF_BLOCK	ARRAY_OF_BLOCKLENGTHS, NTS, OLDTYPE, NEWTYPE, IERROR) KLENGTHS(*), OLDTYPE, NEWTYPE, IERROR)) ARRAY_OF_DISPLACEMENTS(*)
45	Assum	e that oldtype has type map	9,
47 48	$\{(typ)$	$e_0, disp_0), \ldots, (type_{n-1}, disp_n)$	$_{n-1})\},$

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 $n \cdot \sum_{i=0}^{count-1} B[i]$ entries:

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

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.

.....

IN count number of blocks—also number of entries in array_of_displacements (non-negative integer) 2 IN blocklength number of elements in each block (non-negative integer) 2 IN array_of_displacements array of displacements, in multiples of oldtype (array of integers) 3 IN oldtype old datatype (handle) 3 OUT newtype new datatype (handle) 3	22			
IN count number of blocks—also number of entries in array_of_displacements (non-negative integer) 2 IN blocklength number of elements in each block (non-negative integer) 2 IN array_of_displacements array of displacements, in multiples of oldtype (array of integers) 2 IN old type old datatype (handle) 3 OUT new type new datatype (handle) 3 C binding Image: Section Sectio	23			
IN blocklength number of elements in each block (non-negative integer) 2 IN array_of_displacements array of displacements, in multiples of oldtype (array of integers) 2 IN oldtype old datatype (handle) 3 OUT newtype new datatype (handle) 3 C binding Image: State	24			
IN blocklength number of elements in each block (non-negative integer) 2 IN array_of_displacements array of displacements, in multiples of oldtype (array of integers) 2 IN oldtype old datatype (handle) 3 OUT newtype new datatype (handle) 3 C binding Image: State of State	25			
IN array_of_displacements array of displacements, in multiples of oldtype (array of integers) IN oldtype old datatype (handle) OUT newtype new datatype (handle) C binding Image: State Stat	26			
IN old datatype (handle) OUT new datatype (handle) C binding Image: State of the state o	27 28			
IN old datatype (handle) OUT new datatype (handle) C binding Image: State of the state o	29			
IN old type old datatype (handle) a OUT newtype new datatype (handle) a C binding a a	30			
OUT new datatype (handle) C binding	31			
C binding	32			
C binding	33			
5	34			
int MPI_Type_create_indexed_block(int count, int blocklength,	35			
	37			
MPI_Datatype *newtype) 3	38			
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)	11			
	12			
	13			
	14			
	15			
11120210, 111211(11), 11 County, 51001101601,	16			
	17			
TYPE(MPI_Datatype), INTENT(IN) :: oldtype 4	18			

Unofficial Draft for Comment Only

```
1
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
\mathbf{2}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_Type_create_indexed_block(count, blocklength, array_of_displacements,
4
                    oldtype, newtype, ierror)
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
                    OLDTYPE, NEWTYPE, IERROR)
14
         INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDTYPE,
15
                     NEWTYPE, IERROR
16
17
     Hindexed_block The function MPI_TYPE_CREATE_HINDEXED_BLOCK is identical to
18
     MPI_TYPE_CREATE_INDEXED_BLOCK, except that block displacements in
19
     array_of_displacements are specified in bytes, rather than in multiples of the oldtype extent.
20
21
22
     MPI_TYPE_CREATE_HINDEXED_BLOCK(count, blocklength, array_of_displacements,
23
                    oldtype, newtype)
24
       IN
                count
                                            number of blocks—also number of entries in
25
                                            array_of_displacements (non-negative integer)
26
27
       IN
                 blocklength
                                            number of elements in each block (non-negative
28
                                            integer)
29
       IN
                 array_of_displacements
                                            byte displacement of each block (array of integers)
30
       IN
                oldtype
                                            old datatype (handle)
^{31}
32
       OUT
                 newtype
                                            new datatype (handle)
33
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
     int MPI_Type_create_hindexed_block_c(MPI_Count count,
39
                    MPI_Count blocklength,
40
                    const MPI_Count array_of_displacements[],
41
                    MPI_Datatype oldtype, MPI_Datatype *newtype)
42
43
     Fortran 2008 binding
44
     MPI_Type_create_hindexed_block(count, blocklength, array_of_displacements,
45
                    oldtype, newtype, ierror)
46
         INTEGER, INTENT(IN) :: count, blocklength
47
48
```

<pre>INTEGER(KIND=MPI_ADDRESS_KIND)</pre>	, INTENT(IN) ::	1
array_of_displacemer	nts(count)	2
TYPE(MPI_Datatype), INTENT(IN)	:: oldtype	3
TYPE(MPI_Datatype), INTENT(OUT	C) :: newtype	4
INTEGER, OPTIONAL, INTENT(OUT)	:: ierror	5
• =		6 7
		8
		9
		10
• 1	• •	11
		12
INTEGER, OFIIONAL, INTENI(001,		13
ran binding		14
		15 16
INTEGER COUNT, BLOCKLENGTH, OI	LDTYPE, NEWTYPE, IERROR	17
INTEGER(KIND=MPI_ADDRESS_KIND)	ARRAY_OF_DISPLACEMENTS(*)	18
		19
		20
		21
	XED in that it allows each block to consist of repli-	22
ons of different datatypes.		23
		24
		25 26
		27
count		28
· · · · · ·		29
	array_of_blocklengths (non-negative integer)	30
array_of_blocklengths	number of elements in each block (array of	31
	non-negative integers)	32
array_of_displacements	byte displacement of each block (array of integers)	33 34
array_of_types	type of elements in each block (array of handles)	35
JT newtype	new datatype (handle)	36
		37
inding		38
	<pre>unt, const int array_of_blocklengths[],</pre>	39
		40
-	-	41
		42
• -		43
	v ot blocklongthall	
	y_of_blocklengths[],	44
const MPI_Count arra	y_of_displacements[],	45
const MPI_Count arra	-	
	array_of_displacement TYPE(MPI_Datatype), INTENT(IN) TYPE(MPI_Datatype), INTENT(OUT) INTEGER, OPTIONAL, INTENT(OUT) Type_create_hindexed_block(cou	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(*) ct MPI_TYPE_CREATE_STRUCT is the most general type constructor. It further tralizes MPI_TYPE_CREATE_HINDEXED in that it allows each block to consist of repli- ons of different datatypes. _TYPE_CREATE_STRUCT(count, array_of_blocklengths, array_of_displacements, array_of_types, newtype) count number of blocks—also number of entries in arrays array_of_blocklengths (non-negative integer) array_of_blocklengths number of elements in each block (array of non-negative integers) array_of_displacements array_of_types type of elements in each block (array of integers) type of elements in each block (array of integers) JT newtype new datatype (handle)

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

$$(type_{0}^{\mathsf{C}-1}, disp_{0}^{\mathsf{C}-1} + \mathsf{D}[\mathsf{c}-1] + (\mathsf{B}[\mathsf{c}-1] - 1) \cdot ex_{\mathsf{C}-1}), \dots,$$

$$(type_{n\mathsf{C}-1}^{\mathsf{C}-1}, disp_{n\mathsf{C}-1}^{\mathsf{C}-1} + \mathsf{D}[\mathsf{c}-1] + (\mathsf{B}[\mathsf{c}-1] - 1) \cdot ex_{\mathsf{C}-1})\}.$$
¹
²
³

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)	

			14
IN	ndims	number of array dimensions (positive integer)	15
IN	array_of_sizes	number of elements of type oldtype in each dimension	16
		of the full array (array of positive integers)	17
IN	array_of_subsizes	number of elements of type oldtype in each dimension	18
	-	of the subarray (array of positive integers)	19
IN	array_of_starts	starting coordinates of the subarray in each	20
		dimension (array of non-negative integers)	21
		dimension (array of non negative moegers)	22
IN	order	array storage order flag (state)	23
IN	oldtype	old datatype (handle)	24
<u></u>			25
OUT	newtype	new datatype (handle)	26

C binding

```
int MPI_Type_create_subarray(int ndims, const int array_of_sizes[],
                                                                                  29
              const int array_of_subsizes[], const int array_of_starts[],
                                                                                  30
             int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
                                                                                  31
                                                                                  32
int MPI_Type_create_subarray_c(int ndims, const MPI_Count array_of_sizes[],
                                                                                  33
              const MPI_Count array_of_subsizes[],
                                                                                  34
              const MPI_Count array_of_starts[], int order,
                                                                                  35
             MPI_Datatype oldtype, MPI_Datatype *newtype)
                                                                                  36
Fortran 2008 binding
                                                                                  37
MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,
                                                                                  38
              array_of_starts, order, oldtype, newtype, ierror)
                                                                                  39
    INTEGER, INTENT(IN) :: ndims, array_of_sizes(ndims),
                                                                                  40
              array_of_subsizes(ndims), array_of_starts(ndims), order
                                                                                  41
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
                                                                                  42
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                  43
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  44
                                                                                  45
MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,
                                                                                  46
              array_of_starts, order, oldtype, newtype, ierror)
                                                                                  47
    INTEGER, INTENT(IN) :: ndims, order
                                                                                  48
```

4

5

6

7 8

27

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

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

		7
$Subarray(1, \{size_0\}, \{subsize_0\}, \{start_0\},\$	(5.2)	8
$\{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\})$		9
$= \{(lb_marker, 0),$		10
$(type_0, disp_0 + start_0 \times ex), \dots, (type_{n-1}, disp_{n-1} + start_0 \times ex),$		11
$(type_0, disp_0 + (start_0 + 1) \times ex), \dots, (type_{n-1}, type_{n-1})$		12 13
$(sgpe_0, usp_0 + (start_0 + 1) \times cu), \dots, (sgpe_{n-1}, disp_{n-1} + (start_0 + 1) \times ex), \dots$		14
$(type_0, disp_0 + (start_0 + subsize_0 - 1) \times ex), \dots,$		15
		16
$(type_{n-1}, disp_{n-1} + (start_0 + subsize_0 - 1) \times ex),$		17
$(ub_marker, size_0 \times ex)$ }		18
		19
Subarray $(ndims, \{size_0, size_1, \dots, size_{ndims-1}\},\$	(5.3)	20
$\{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\},\$		21
$\{start_0, start_1, \dots, start_{ndims-1}\}, oldtype\}$		22 23
= Subarray($ndims - 1, \{size_1, size_2, \dots, size_{ndims-1}\},\$		23 24
$\{subsize_1, subsize_2, \dots, subsize_{ndims-1}\},\$		25
		26
$\{start_1, start_2, \dots, start_{ndims-1}\},$		27
$Subarray(1, \{size_0\}, \{subsize_0\}, \{start_0\}, oldtype))$		28
		29
Subarray $(ndims, \{size_0, size_1, \dots, size_{ndims-1}\},\$	(5.4)	30
$\{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\},\$		31
$\{start_0, start_1, \dots, start_{ndims-1}\}, oldtype)$		32
= Subarray($ndims - 1$, { $size_0, size_1, \ldots, size_{ndims-2}$ },		33 34
$\{subsize_0, subsize_1, \dots, subsize_{ndims-2}\},\$		34 35
$\{start_0, start_1, \dots, start_{ndims-2}\},\$		36
Subarray $(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, oldty$	(n o))	37
Subarray $(1, \{si \ge e_{ndims-1}\}, \{subsi \ge e_{ndims-1}\}, \{si = array, \{si = array, 1\}, outy$	(he)	38

For an example use of MPI_TYPE_CREATE_SUBARRAY in the context of I/O see Section 14.9.2.

5.1.4Distributed Array Datatype Constructor

4

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 47follows. Complementary filetypes are created by having every process of a group call 48

Unofficial Draft for Comment Only

1 $\mathbf{2}$

3

4

56

39 40

 41 42

43

44

	134		CHAPTER 5. DATATYPES
1 2 3 4 5 6 7	set a to de and	ppropriately). These filetypes efine the view (via MPI_FILE_S Section 14.3. Using this view	iments (with the exception of rank which should be (along with identical disp and etype) are then used SET_VIEW), see MPI I/O, especially Section 14.1.1 , a collective data access operation (with identical ribution pattern. (<i>End of advice to users.</i>)
8 9 10	MPI_TYP	•	k, ndims, array_of_gsizes, array_of_distribs, _psizes, order, oldtype, newtype)
11	IN	size	size of process group (positive integer)
12	IN	rank	rank in process group (non-negative integer)
13 14 15	IN	ndims	number of array dimensions as well as process grid dimensions (positive integer)
16 17	IN	array_of_gsizes	number of elements of type oldtype in each dimension of global array (array of positive integers)
18 19	IN	array_of_distribs	distribution of array in each dimension (array of states)
20 21 22	IN	array_of_dargs	distribution argument in each dimension (array of positive integers)
23 24	IN	array_of_psizes	size of process grid in each dimension (array of positive integers)
25	IN	order	array storage order flag (state)
26 27	IN	oldtype	old datatype (handle)
28	OUT	newtype	new datatype (handle)
29 30 31 32 33 34	C bindin int MPI_7	Type_create_darray(int siz const int array_of_g const int array_of_d	ze, int rank, int ndims, sizes[], const int array_of_distribs[], args[], const int array_of_psizes[], ype oldtype, MPI_Datatype *newtype)
35	int MPT '		size, int rank, int ndims,
36 37	Inc in I	const MPI_Count arra	
38			<pre>istribs[], const int array_of_dargs[],</pre>
39			<pre>sizes[], int order, MPI_Datatype oldtype,</pre>
40		MPI_Datatype *newtyp	e)
41		2008 binding	
42 43	MPI_Type	_create_darray(size, rank	
44		oldtype, newtype, ie	<pre>rray_of_dargs, array_of_psizes, order, rror)</pre>
45	INTE		rank, ndims, array_of_gsizes(ndims),
46			dims), array_of_dargs(ndims),
47		array_of_psizes(ndir	
48	TYPE	(MPI_Datatype), INTENT(IN)) :: oldtype

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

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

⁴⁸ i. The values of r[i] are given by the following code fragment:

```
t_rank = rank;
t_size = 1;
for (i = 0; i < ndims; i++)
    t_size *= array_of_psizes[i];
for (i = 0; i < ndims; i++) {
    t_size = t_size / array_of_psizes[i];
    r[i] = t_rank / t_size;
    t_rank = t_rank % t_size;
}
```

Let the typemap of **oldtype** have the form:

```
\{(type_0, disp_0), (type_1, disp_1), \ldots, (type_{n-1}, disp_{n-1})\}
```

where $type_i$ is a predefined MPI datatype, and let ex be the extent of oldtype. The following function uses the conceptual datatypes lb_marker and ub_marker, see Section 5.1.6 for details.

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

```
18
cyclic(darg, gsize, r, psize, oldtype)
                                                                                                                             19
  = {(lb_marker, 0),
                                                                                                                             20
        (type_0, disp_0 + r \times darg \times ex), \ldots,
                                                                                                                             21
                  (type_{n-1}, disp_{n-1} + r \times darg \times ex),
                                                                                                                             22
                                                                                                                             23
        (type_0, disp_0 + (r \times darg + 1) \times ex), \ldots,
                                                                                                                             ^{24}
                  (type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex)
                                                                                                                             25
                                                                                                                             26
        (type_0, disp_0 + ((r+1) \times darg - 1) \times ex), \ldots,
                                                                                                                             27
                   (type_{n-1}, disp_{n-1} + ((r+1) \times darg - 1) \times ex),
                                                                                                                             28
                                                                                                                             29
                                                                                                                             30
        (type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex), \ldots,
                                                                                                                             31
                   (type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex),
                                                                                                                             32
        (type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex), \dots,
                                                                                                                             33
                                                                                                                             34
                   (type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex + psize \times darg \times ex),
                                                                                                                             35
                                                                                                                             36
        (type_0, disp_0 + ((r+1) \times darg - 1) \times ex + psize \times darg \times ex), \dots,
                                                                                                                             37
                   (type_{n-1}, disp_{n-1} + ((r+1) \times darg - 1) \times ex + psize \times darg \times ex),
                                                                                                                             38
                                                                                                                             39
                                                                                                                             40
        (type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex \times (count - 1)), \dots,
                                                                                                                             41
                   (type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex \times (count - 1)),
                                                                                                                             42
                                                                                                                             43
        (type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex \times (count - 1)), \dots,
                                                                                                                             44
                   (type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex
                                                                                                                             45
                             +psize \times darg \times ex \times (count - 1)),
                                                                                                                             46
                                                                                                                             47
                                                                                                                             48
        (type_0, disp_0 + (r \times darg + darg_{last} - 1) \times ex
```

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

10

11 12

13

14

15

16

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

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

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 20 and 154.

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) location IN 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.

Unofficial Draft for Comment Only

1

2

6

8

9

10

11

12

13

14

15

16

17

18

19

20

21 22

23

 24

25

26

27 28 29

30

31 32

33 34

35

36

37

38

39

40

41 42

43

44

45

46

1 2 3 4 5	<i>Rationale.</i> 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. (<i>End of rationale.</i>)
6	Example 5.8 Using MPI_GET_ADDRESS for an array.
7 8 9 10 11 12 13 14	<pre>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.</pre>
15 16 17 18 19 20 21 22 23	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.)
24 25 26 27	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.)
28 29 30 31	To ensure portability, arithmetic on MPI addresses must be performed using the MPI_AINT_ADD and MPI_AINT_DIFF functions.
32	MPI_AINT_ADD(base, disp)
33 34	IN base base address (integer)
35	IN disp displacement (integer)
36 37 38	C binding MPI_Aint MPI_Aint_add(MPI_Aint base, MPI_Aint disp)
39 40 41 42	Fortran 2008 binding INTEGER(KIND=MPI_ADDRESS_KIND) MPI_Aint_add(base, disp) INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: base, disp
43 44 45	Fortran binding INTEGER(KIND=MPI_ADDRESS_KIND) MPI_AINT_ADD(BASE, DISP) INTEGER(KIND=MPI_ADDRESS_KIND) BASE, DISP
46 47 48	MPI_AINT_ADD produces a new MPI_Aint value that is equivalent to the sum of the base and disp arguments, where base represents a base address returned by a call to

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

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

MPI_AINT_DIFF(addr1, addr2)

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

C binding

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

Fortran 2008 binding

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

Fortran binding

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

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

The following auxiliary functions provide useful information on derived datatypes.

MPI_TYPE_SIZE(datatype, size)

IN	datatura	deteture to get information on (handle)	36
IIN	datatype	datatype to get information on (handle)	37
OUT	size	datatype size (integer)	38
			39
C bindir	C binding		
<pre>int MPI_Type_size(MPI_Datatype datatype, int *size)</pre>			41
			42
<pre>int MPI_Type_size_c(MPI_Datatype datatype, MPI_Count *size)</pre>			43
Fortran 2008 binding			44
MPI_Type_size(datatype, size, ierror)			45
TYPE	(MPI_Datatype), INTENT(IN	I) :: datatype	46
INTE	GER, INTENT(OUT) :: size		47
INTE	GER, OPTIONAL, INTENT(OUT	I) :: ierror	48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5 6

7 8 9

15

16 17

18

19

20

21

22

23 24

25

26

27

28

29

30

31

32

33 34

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

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

35 36

45

5.1.6 Lower-Bound and Upper-Bound Markers

It is often convenient to define explicitly the lower bound and upper bound of a type map, 37 and override the definition given on page 143. This allows one to define a datatype that has 38 "holes" at its beginning or its end, or a datatype with entries that extend above the upper 39 bound or below the lower bound. Examples of such usage are provided in Section 5.1.14. 4041 Also, the user may want to overide the alignment rules that are used to compute upper 42bounds and extents. E.g., a C compiler may allow the user to overide default alignment rules for some of the structures within a program. The user has to specify explicitly the 43 bounds of the datatypes that match these structures. 44

To achieve this, we add two additional conceptual datatypes, **lb_marker** and

⁴⁶ **ub_marker**, that represent the lower bound and upper bound of a datatype. These con-⁴⁷ ceptual datatypes occupy no space $(extent(lb_marker) = extent(ub_marker) = 0)$. They do ⁴⁸ 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 {(lb_marker, -3), (int, 0), (ub_marker, 6)}. If this type is replicated twice by a call to MPI_TYPE_CONTIGUOUS(2, type1, type2) then the newly created type can be described by the typemap {(lb_marker, -3), (int, 0), (int, 0), (int, 9), (ub_marker, 15)}. (An entry of type ub_marker can be deleted if there is another entry of type ub_marker with a higher displacement; an entry of type lb_marker can be deleted if there is another entry of type lb_marker with a lower displacement.)

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

	min dien.	if no entry has type
$lb(Typemap) = \langle$	$\min_j disp_j$	lb_marker
	$\min_{j} \{ disp_j \text{ such that } type_j = lb_marker \}$	otherwise

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

$$ub(Typemap) = \begin{cases} \max_{j}(disp_{j} + sizeof(type_{j})) + \epsilon & \text{if no entry has type} \\ \max_{j}\{disp_{j} \text{ such that } type_{j} = ub_marker \} & \text{otherwise} \end{cases}$$

Then

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

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

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

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

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

 $\overline{7}$

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

As an example, instead of

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

one should define

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

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

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

5.1.7 Extent and Bounds of Datatypes 1 2 MPI_TYPE_GET_EXTENT(datatype, lb, extent) IN datatype datatype to get information on (handle) 6 OUT lb lower bound of datatype (integer) OUT extent extent of datatype (integer) 10 C binding 11 int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *1b, 12MPI_Aint *extent) 13 int MPI_Type_get_extent_c(MPI_Datatype datatype, MPI_Count *1b, 14MPI_Count *extent) 1516Fortran 2008 binding 17 MPI_Type_get_extent(datatype, lb, extent, ierror) 18 TYPE(MPI_Datatype), INTENT(IN) :: datatype 19 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: lb, extent 20INTEGER, OPTIONAL, INTENT(OUT) :: ierror 21MPI_Type_get_extent(datatype, lb, extent, ierror) 22 TYPE(MPI_Datatype), INTENT(IN) :: datatype 2324INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: 1b, extent INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2526Fortran binding 27MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR) 28 INTEGER DATATYPE, IERROR 29 INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT 30 31 32 MPI_TYPE_GET_EXTENT_X(datatype, lb, extent) 33 34 IN datatype datatype to get information on (handle) 35 OUT lower bound of datatype (integer) lb 36 OUT extent extent of datatype (integer) 37 38 39 C binding int MPI_Type_get_extent_x(MPI_Datatype datatype, MPI_Count *lb, 40 41 MPI_Count *extent) 42Fortran 2008 binding 43 MPI_Type_get_extent_x(datatype, lb, extent, ierror) 44 TYPE(MPI_Datatype), INTENT(IN) :: datatype 45INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: lb, extent 46INTEGER, OPTIONAL, INTENT(OUT) :: ierror 47

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

```
48
```

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

5.1.8 True Extent of Datatypes

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

MPI_TYPE_GET_TRUE_EXTENT(datatype, true_lb, true_extent)

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

C binding

int	MPI_Type_get_true_extent(MPI_Datatype	datatype, 1	MPI_Aint *t	rue_lb,
	MPI_Aint *true_extent)			
int	MPI_Type_get_true_extent_c(MPI_Datatyp	e datatype	, MPI_Count	<pre>*true_lb,</pre>

MPI_Count *true_extent)

Fortran 2008 binding

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

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

Fortran binding

```
MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
INTEGER DATATYPE, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT
```

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

Unofficial Draft for Comment Only

MPI_TYPE_COMMIT(datatype) 1 $\mathbf{2}$ INOUT datatype datatype that is committed (handle) 3 4 C binding 5 int MPI_Type_commit(MPI_Datatype *datatype) 6 Fortran 2008 binding MPI_Type_commit(datatype, ierror) 9 TYPE(MPI_Datatype), INTENT(INOUT) :: datatype 10 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 11 Fortran binding 12MPI_TYPE_COMMIT(DATATYPE, IERROR) 13 INTEGER DATATYPE, IERROR 1415The commit operation commits the datatype, that is, the formal description of a com-16munication buffer, not the content of that buffer. Thus, after a datatype has been commit-17ted, it can be repeatedly reused to communicate the changing content of a buffer or, indeed, 18 the content of different buffers, with different starting addresses. 19 The system may "compile" at commit time an internal Advice to implementors. 20representation for the datatype that facilitates communication, e.g., change from a 21compacted representation to a flat representation of the datatype, and select the most 22 convenient transfer mechanism. (End of advice to implementors.) 2324MPI_TYPE_COMMIT will accept a committed datatype; in this case, it is equivalent 25to a no-op. 2627**Example 5.10** The following code fragment gives examples of using MPI_TYPE_COMMIT. 28 29 INTEGER type1, type2 30 CALL MPI_TYPE_CONTIGUOUS(5, MPI_REAL, type1, ierr) 31 ! new type object created 32 CALL MPI_TYPE_COMMIT(type1, ierr) 33 ! now type1 can be used for communication 34 type2 = type135! type2 can be used for communication 36 ! (it is a handle to same object as type1) 37 CALL MPI_TYPE_VECTOR(3, 5, 4, MPI_REAL, type1, ierr) 38 ! new uncommitted type object created 39 CALL MPI_TYPE_COMMIT(type1, ierr) 40 ! now type1 can be used anew for communication 41 4243 MPI_TYPE_FREE(datatype) 44 INOUT datatype datatype that is freed (handle) 45

C binding int MPI_Type_free(MPI_Datatype *datatype)

Unofficial Draft for Comment Only

46 47

1	Fortran 2008 binding		
2	MPI_Type_free(datatype, ierror)		
3	TYPE(MPI_Datatype), INTENT(INOUT) :: datatype		
4	INTEGER, OPTIONAL, INTENT(OUT) :: ierror		
5	INTEGER, OFTIONAL, INTENT(001) TETIOI		
6	Fortran binding		
7	MPI_TYPE_FREE(DATATYPE, IERROR)		
8	INTEGER DATATYPE, IERROR		
9	Marks the datatype object associated with datatype for deallocation and sets datatype		
10	to MPI_DATATYPE_NULL. Any communication that is currently using this datatype will		
11	complete normally. Freeing a datatype does not affect any other datatype that was built		
12	from the freed datatype. The system behaves as if input datatype arguments to derived		
13	datatype constructors are passed by value.		
14			
15	Advice to implementors. The implementation may keep a reference count of active		
16	communications that use the datatype, in order to decide when to free it. Also, one		
17	may implement constructors of derived datatypes so that they keep pointers to their		
18	datatype arguments, rather than copying them. In this case, one needs to keep track		
19	of active datatype definition references in order to know when a datatype object can		
20	be freed. (End of advice to implementors.)		
21	be need. (Enta of dablee to implementors.)		
21			
22	5.1.10 Duplicating a Datatype		
23 24			
25	MPI_TYPE_DUP(oldtype, newtype)		
26	IN oldtype datatype (handle)		
27			
28	OUT newtype copy of oldtype (handle)		
29			
30	C binding		
31	int MPI_Type_dup(MPI_Datatype oldtype, MPI_Datatype *newtype)		
32			
33	Fortran 2008 binding		
34	MPI_Type_dup(oldtype, newtype, ierror)		
35	TYPE(MPI_Datatype), INTENT(IN) :: oldtype		
36	TYPE(MPI_Datatype), INTENT(OUT) :: newtype		
37	INTEGER, OPTIONAL, INTENT(OUT) :: ierror		
38	Fortney hinding		
39	Fortran binding		
40	MPI_TYPE_DUP(OLDTYPE, NEWTYPE, IERROR)		
41	INTEGER OLDTYPE, NEWTYPE, IERROR		
42	MPI_TYPE_DUP is a type constructor which duplicates the existing oldtype with as-		
43	sociated key values. For each key value, the respective copy callback function determines		
43	the attribute value associated with this key in the new communicator; one particular action		
	that a copy callback may take is to delete the attribute from the new datatype. Returns		
45	in newtype a new datatype with exactly the same properties as oldtype and any copied		

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 8 lower bound and yields the same net result when fully decoded with the functions in Section 5.1.13. The newtype has the same committed state as the old oldtype.

5.1.11 Use of General Datatypes in Communication

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

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

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

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

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

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

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

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

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

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

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

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

 $45 \\ 46$

 $\mathbf{2}$

 24

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

5.1. DERIVED DATATYPES

TYPE(TYPE) INTE(INTE(Fortran h MPI_GET_H	ELEMENTS(STATUS, DATATYPE	:: status) :: datatype INTENT(OUT) :: count) :: ierror	1 2 3 4 5 6 7 8 9 10 11
MPI_GET	_ELEMENTS_X(status, dataty	pe, count)	12
IN	status	return status of receive operation (status)	13
IN	datatype	datatype used by receive operation (handle)	14 15
OUT	count	number of received basic elements (integer)	16
			17
C bindin			18
int MPI_(<pre>Set_elements_x(const MPI_S MPI_Count *count)</pre>	Status *status, MPI_Datatype datatype,	19 20
			21
	2008 binding		22
	elements_x(status, dataty) (MPI_Status), INTENT(IN)		23
	(MPI_Datatype), INTENT(IN)		24
	GER(KIND=MPI_COUNT_KIND),		25 26
INTEC	GER, OPTIONAL, INTENT(OUT) :: ierror	20
Fortran h	binding		28
	ELEMENTS_X(STATUS, DATATY	PE, COUNT, IERROR)	29
INTEC	GER STATUS (MPI_STATUS_SIZ	E), DATATYPE, IERROR	30
INTEC	GER(KIND=MPI_COUNT_KIND)	COUNT	31
The d	atatype argument should mat	ch the argument provided by the receive call that	32 33
		ns, if the OUT parameter cannot express the value	34
		s too small to hold the output value), it is set to	35
MPI_UNDE			36
-	°	PI_GET_COUNT (Section 3.2.5), has a different be- vel entries" received, i.e. the number of "copies" of	37
	-	, MPI_GET_COUNT may return any integer value	38
		DUNT returns k , then the number of basic elements	39
		_GET_ELEMENTS or MPI_GET_ELEMENTS_X) is	40 41
$n \cdot k$. If th	e number of basic elements re	ceived is not a multiple of n , that is, if the receive	42
-	_	mber of datatype "copies," then MPI_GET_COUNT	43
sets the va	lue of count to MPI_UNDEFIN	ED.	44
Example	5.12 Usage of MPI GET CC	OUNT and MPI_GET_ELEMENTS.	45
T	0		46
			47

```
1
      . . .
2
     CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, Type2, ierr)
3
     CALL MPI_TYPE_COMMIT(Type2, ierr)
4
      . . .
5
     CALL MPI_COMM_RANK(comm, rank, ierr)
6
     IF (rank.EQ.0) THEN
7
         CALL MPI_SEND(a, 2, MPI_REAL, 1, 0, comm, ierr)
8
         CALL MPI_SEND(a, 3, MPI_REAL, 1, 0, comm, ierr)
9
     ELSE IF (rank.EQ.1) THEN
10
         CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
11
         CALL MPI_GET_COUNT(stat, Type2, i, ierr)
                                                             ! returns i=1
12
         CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr)
                                                            ! returns i=2
13
         CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
14
         CALL MPI_GET_COUNT(stat, Type2, i, ierr)
                                                             ! returns i=MPI_UNDEFINED
15
         CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr)
                                                            ! returns i=3
16
     END IF
17
         The functions MPI_GET_ELEMENTS and MPI_GET_ELEMENTS_X can also be used
18
     after a probe to find the number of elements in the probed message. Note that the
19
     MPI_GET_COUNT, MPI_GET_ELEMENTS, and MPI_GET_ELEMENTS_X return the same
20
     values when they are used with basic datatypes as long as the limits of their respective
21
     count arguments are not exceeded.
22
23
           Rationale. The extension given to the definition of MPI_GET_COUNT seems natural:
24
           one would expect this function to return the value of the count argument, when the
25
           receive buffer is filled. Sometimes datatype represents a basic unit of data one wants
26
           to transfer, for example, a record in an array of records (structures). One should be
27
           able to find out how many components were received without bothering to divide by
28
           the number of elements in each component. However, on other occasions, datatype
29
           is used to define a complex layout of data in the receiver memory, and does not
30
           represent a basic unit of data for transfers. In such cases, one needs to use the
31
           function MPI_GET_ELEMENTS or MPI_GET_ELEMENTS_X. (End of rationale.)
32
33
                                     The definition implies that a receive cannot change the
           Advice to implementors.
34
           value of storage outside the entries defined to compose the communication buffer. In
35
           particular, the definition implies that padding space in a structure should not be mod-
36
           ified when such a structure is copied from one process to another. This would prevent
37
           the obvious optimization of copying the structure, together with the padding, as one
38
           contiguous block. The implementation is free to do this optimization when it does not
39
           impact the outcome of the computation. The user can "force" this optimization by
40
           explicitly including padding as part of the message. (End of advice to implementors.)
41
42
     5.1.12
            Correct Use of Addresses
43
     Successively declared variables in C or Fortran are not necessarily stored at contiguous
44
45
     locations. Thus, care must be exercised that displacements do not cross from one variable
46
     to another. Also, in machines with a segmented address space, addresses are not unique
47
     and address arithmetic has some peculiar properties. Thus, the use of addresses, that is,
```

⁴⁸ displacements relative to the start address MPI_BOTTOM, has to be restricted.

Unofficial Draft for Comment Only

Variables belong to the same **sequential storage** if they belong to the same array, to the same COMMON block in Fortran, or to the same structure in C. Valid addresses are defined recursively as follows:

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

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

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

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

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

5.1.13 Decoding a Datatype

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

 24

 31

156		CHAPTER 5. DATATYPES
MPI_TYF	PE_GET_ENVELOPE(dataty num_datatypes, com	/pe, num_integers, num_addresses, num_large_counts, biner)
IN	datatype	datatype to decode (handle)
OUT	num_integers	number of input integers used in call constructing combiner (non-negative integer)
OUT	num_addresses	number of input addresses used in call constructing combiner (non-negative integer)
OUT	num_large_counts	number of input large counts used in call constructing combiner (non-negative integer, only present for large count variants)
OUT	num_datatypes	number of input datatypes used in call constructing combiner (non-negative integer)

OUT

combiner

1

 $\mathbf{2}$

3

4

56 7

8 9

10 11 12

13 1415

16

19

20

21

22

23

28

29

30

31

43

44

17C binding 18

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

MPI_Count *num_addresses, MPI_Count *num_large_counts, MPI_Count *num_datatypes, int *combiner)

combiner (state)

 24 Fortran 2008 binding 25

MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes, 26combiner, ierror) 27TYPE(MPI_Datatype), INTENT(IN) :: datatype

INTEGER, INTENT(OUT) :: num_integers, num_addresses, num_datatypes, combiner

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_get_envelope(datatype, num_integers, num_addresses, 32 num_large_counts, num_datatypes, combiner, ierror) 33 TYPE(MPI_Datatype), INTENT(IN) :: datatype 34 INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: num_integers, 35num_addresses, num_large_counts, num_datatypes 36 INTEGER, INTENT(OUT) :: combiner 37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 38 39 Fortran binding 40MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, 41

COMBINER, IERROR) 42

INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER, IERROR

For the given datatype, MPI_TYPE_GET_ENVELOPE returns information on the num-45ber and type of input arguments used in the call that created the datatype. The number-of-4647arguments 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 48

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.

MPI_COMBINER_NAMED	a named predefined datatype
MPI_COMBINER_DUP	MPI_TYPE_DUP
MPI_COMBINER_CONTIGUOUS	MPI_TYPE_CONTIGUOUS
MPI_COMBINER_VECTOR	MPI_TYPE_VECTOR
MPI_COMBINER_HVECTOR	MPI_TYPE_CREATE_HVECTOR
MPI_COMBINER_INDEXED	MPI_TYPE_INDEXED
MPI_COMBINER_HINDEXED	MPI_TYPE_CREATE_HINDEXED
MPI_COMBINER_INDEXED_BLOCK	MPI_TYPE_CREATE_INDEXED_BLOCK
MPI_COMBINER_HINDEXED_BLOCK	MPI_TYPE_CREATE_HINDEXED_BLOCK
MPI_COMBINER_STRUCT	MPI_TYPE_CREATE_STRUCT
MPI_COMBINER_SUBARRAY	MPI_TYPE_CREATE_SUBARRAY
MPI_COMBINER_DARRAY	MPI_TYPE_CREATE_DARRAY
MPI_COMBINER_F90_REAL	MPI_TYPE_CREATE_F90_REAL
MPI_COMBINER_F90_COMPLEX	MPI_TYPE_CREATE_F90_COMPLEX
MPI_COMBINER_F90_INTEGER	MPI_TYPE_CREATE_F90_INTEGER
MPI_COMBINER_RESIZED	MPI_TYPE_CREATE_RESIZED

Table 5.1: combiner values returned from MPI_TYPE_GET_ENVELOPE

If combiner is MPI_COMBINER_NAMED then datatype is a named predefined datatype. If the MPI_TYPE_GET_ENVELOPE variant without num_large_counts is invoked with a datatype that requires an output value of num_large_counts > 0, then an error of class MPI_ERR_TYPE is raised.

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

The actual arguments used in the creation call for a datatype can be obtained using MPI_TYPE_GET_CONTENTS.

MPI_TYPE_GET_ENVELOPE and MPI_TYPE_GET_CONTENTS also support large 46 count types in separate additional MPI procedures in C (suffixed with the "_c") and interface 47 polymorphism in Fortran when using USE mpi_f08. 48

Unofficial Draft for Comment Only

1	MPI_TYP	· - ·	pe, max_integers, max_addresses, max_large_counts,
2 3		max_datatypes, array_ array_of_large_counts,	of_integers, array_of_addresses, array_of_datatypes)
4 5	IN	datatype	datatype to decode (handle)
6 7	IN	max_integers	number of elements in array_of_integers (non-negative integer)
8 9	IN	max_addresses	number of elements in array_of_addresses (non-negative integer)
10 11 12 13	IN	max_large_counts	number of elements in array_of_large_counts (non-negative integer, only present for large count variants)
13 14 15	IN	max_datatypes	number of elements in array_of_datatypes (non-negative integer)
16 17	OUT	array_of_integers	contains integer arguments used in constructing datatype (array of integers)
18 19 20	OUT	array_of_addresses	contains address arguments used in constructing datatype (array of integers)
20 21 22 23	OUT	array_of_large_counts	contains large count arguments used in constructing datatype (array of integers, only present for large count variants)
24 25 26	OUT	array_of_datatypes	contains datatype arguments used in constructing datatype (array of handles)
27 28 29 30 31	C bindin int MPI_7	Type_get_contents(MPI_Dates)	
32 33 34 35 36 37 38	int MPI_7	MPI_Count max_addr	_large_counts[],
 39 40 41 42 43 44 45 46 47 48 	MPI_Type TYPE INTEC INTEC	array_of_integers, ierror) (MPI_Datatype), INTENT(1 GER, INTENT(IN) :: max_:	integers, max_addresses, max_datatypes ay_of_integers(max_integers) ND), INTENT(OUT) ::

TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes)	1 2
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	3
<pre>MPI_Type_get_contents(datatype, max_integers, max_addresses,</pre>	4
<pre>max_large_counts, max_datatypes, array_of_integers,</pre>	5
array_of_addresses, array_of_large_counts, array_of_datatypes,	6
ierror)	7
TYPE(MPI_Datatype), INTENT(IN) :: datatype	8
<pre>INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: max_integers,</pre>	9
<pre>max_addresses, max_large_counts, max_datatypes</pre>	10
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
Fortran binding	18
MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,	19
ARRAY_OF_INTEGERS, ARRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES,	20
IERROR)	21
INTEGER DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,	22
ARRAY_OF_INTEGERS(*), ARRAY_OF_DATATYPES(*), IERROR	23
<pre>INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_ADDRESSES(*)</pre>	24
datatype must be a predefined unnamed or a derived datatype; the call is erroneous if	25
datatype is a predefined named datatype.	26
The values given for max_integers, max_addresses, max_large_counts, and	27
max_datatypes must be at least as large as the value returned in num_integers,	28 29
num_addresses, num_large_counts, and num_datatypes, respectively, in the call	30
MPI_TYPE_GET_ENVELOPE for the same datatype argument.	31
	32
<i>Rationale.</i> The arguments max_integers, max_addresses, max_large_counts, and	33
max_datatypes allow for error checking in the call. (End of rationale.)	34
	35
If the MPI_TYPE_GET_CONTENTS variant without max_large_counts is invoked with	36
a datatype that requires > 0 values in array_of_large_counts, then an error of class	37
MPI_ERR_TYPE is raised.	38
Rationale. The large count variant of this MPI procedure was added in MPI-4.	39
It contains new max_large_counts and array_of_large_counts parameters. The other	40
variant—the variant that existed before MPI-4—was not changed in order to preserve	41
backwards compatibility. (End of rationale.)	42
Sachwards comparishing. (Drea of racconduct.)	43
The datatypes returned in array_of_datatypes are handles to datatype objects that	44
are equivalent to the datatypes used in the original construction call. If these were derived	45
datatypes, then the returned datatypes are new datatype objects, and the user is responsible	46
	47

Unofficial Draft for Comment Only

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.

1 The committed state of returned derived datatypes is undefined, i.e., the datatypes may $\mathbf{2}$ or may not be committed. Furthermore, the content of attributes of returned datatypes is 3 undefined. 4 Note that MPI_TYPE_GET_CONTENTS can be invoked with a 5datatype argument that was constructed using MPI_TYPE_CREATE_F90_REAL, 6 MPI_TYPE_CREATE_F90_INTEGER, or MPI_TYPE_CREATE_F90_COMPLEX (an unnamed $\overline{7}$ predefined datatype). In such a case, an empty array_of_datatypes is returned. 8 *Rationale.* The definition of datatype equivalence implies that equivalent predefined 9 datatypes are equal. By requiring the same handle for named predefined datatypes, 10 it is possible to use the == or .EQ. comparison operator to determine the datatype 11 involved. (End of rationale.) 1213 Advice to implementors. The datatypes returned in array_of_datatypes must appear 14to the user as if each is an equivalent copy of the datatype used in the type constructor 15call. Whether this is done by creating a new datatype or via another mechanism such 16as a reference count mechanism is up to the implementation as long as the semantics 17 are preserved. (End of advice to implementors.) 18 19 The committed state and attributes of the returned datatype is delib-Rationale. 20erately left vague. The datatype used in the original construction may have been 21modified since its use in the constructor call. Attributes can be added, removed, or 22 modified as well as having the datatype committed. The semantics given allow for 23a reference count implementation without having to track these changes. (End of 24rationale.) 2526In the deprecated datatype constructor calls, the address arguments in Fortran are 27of type INTEGER. In the preferred calls, the address arguments are of type 28 INTEGER(KIND=MPI_ADDRESS_KIND). The call MPI_TYPE_GET_CONTENTS returns all ad-29 dresses in an argument of type INTEGER(KIND=MPI_ADDRESS_KIND). This is true even if the 30 deprecated calls were used. Thus, the location of values returned can be thought of as being 31 returned by the C bindings. It can also be determined by examining the preferred calls for 32 datatype constructors for the deprecated calls that involve addresses. 33 34 Rationale. By having all address arguments returned in the 35array_of_addresses argument, the result from a C and Fortran decoding of a datatype 36 gives the result in the same argument. It is assumed that an integer of type 37 INTEGER(KIND=MP1_ADDRESS_KIND) will be at least as large as the INTEGER argument 38 used in datatype construction with the old MPI-1 calls so no loss of information will 39 occur. (End of rationale.) 40 41 The following defines what values are placed in each entry of the returned arrays 42depending on the datatype constructor used for datatype. It also specifies the size of the 43 arrays needed which is the values returned by MPI_TYPE_GET_ENVELOPE. In Fortran, 44the following calls were made: 4546PARAMETER (LARGE = 1000) 47INTEGER TYPE, NI, NA, ND, COMBINER, I(LARGE), D(LARGE), IERROR 48 INTEGER (KIND=MPI_ADDRESS_KIND) A(LARGE)

```
! CONSTRUCT DATATYPE TYPE (NOT SHOWN)
                                                                                        2
CALL MPI_TYPE_GET_ENVELOPE(TYPE, NI, NA, ND, COMBINER, IERROR)
IF ((NI .GT. LARGE) .OR. (NA .GT. LARGE) .OR. (ND .GT. LARGE)) THEN
   WRITE (*, *) "NI, NA, OR ND = ", NI, NA, ND, &
   " RETURNED BY MPI_TYPE_GET_ENVELOPE IS LARGER THAN LARGE = ", LARGE
   CALL MPI_ABORT(MPI_COMM_WORLD, 99, IERROR)
ENDIF
CALL MPI_TYPE_GET_CONTENTS(TYPE, NI, NA, ND, I, A, D, IERROR)
or in C the analogous calls of:
                                                                                       10
                                                                                       11
#define LARGE 1000
                                                                                       12
int ni, na, nd, combiner, i[LARGE];
                                                                                       13
MPI_Aint a[LARGE];
                                                                                       14
MPI_Datatype type, d[LARGE];
                                                                                       15
/* construct datatype type (not shown) */
                                                                                       16
MPI_Type_get_envelope(type, &ni, &na, &nd, &combiner);
                                                                                       17
if ((ni > LARGE) || (na > LARGE) || (nd > LARGE)) {
                                                                                       18
    fprintf(stderr, "ni, na, or nd = %d %d %d returned by ", ni, na, nd);
                                                                                       19
    fprintf(stderr, "MPI_Type_get_envelope is larger than LARGE = %d\n",
                                                                                       20
             LARGE);
                                                                                       21
    MPI_Abort(MPI_COMM_WORLD, 99);
                                                                                       22
};
                                                                                       23
MPI_Type_get_contents(type, ni, na, nd, i, a, d);
                                                                                       ^{24}
                                                                                       25
The following describes the values of the arguments for each combiner. The lower case name
                                                                                       26
of arguments is used.
                                                                                       27
MPI_COMBINER_NAMED the datatype represent a predefined type and therefore it is er-
                                                                                       28
roneous to call MPI_TYPE_GET_CONTENTS.
                                                                                       29
                                                                                       30
MPI_COMBINER_DUP ni = 0, na = 0, nd = 1, and
                                                                                       31
                                                                                       32
                                           С
                                                Fortran location
                    Constructor argument
                                                                                       33
                    oldtype
                                          d[0]
                                                     D(1)
                                                                                       34
                                                                                       35
MPI_COMBINER_CONTIGUOUS ni = 1, na = 0, nd = 1, and
                                                                                       36
                                                                                       37
                                           С
                                                Fortran location
                    Constructor argument
                                                                                       38
                                           i[0]
                    count
                                                      I(1)
                                                                                       39
                    oldtype
                                          d[0]
                                                     D(1)
                                                                                       40
                                                                                       41
MPI_COMBINER_VECTOR ni = 3, na = 0, nd = 1, and
                                                                                       42
                                                                                       43
                                                                                       44
                    Constructor argument
                                           С
                                                Fortran location
                                                                                       45
                                           i[0]
                    count
                                                      I(1)
                                                                                       46
                    blocklength
                                           i[1]
                                                      I(2)
                                                                                       47
                    stride
                                           i[2]
                                                      I(3)
                                                                                       48
                    oldtype
                                          d[0]
                                                     D(1)
```

Unofficial Draft for Comment Only

	Constructor argu	ument C Fe	ortran location	
	count	i[0]	I(1)	
	blocklength	i[1]	I(2)	
	stride	a[0]	A(1)	
	oldtype	d[0]	D(1)	
/IPI_C	OMBINER_INDEXED ni = 2*0	count+1, na = (, nd = 1, and	
	Constructor argument	С	Fortran locatio	on
	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 $i[2*i[0]]$		1)-
	oldtype	d[0]	D(1)	
	Constructor argument count	C i[0]	Fortran location I(1)	_
	$array_of_blocklengths$	i[1] to i[i[0]]	I(2) to $I(I(1)+1)$	
	array_of_displacement			
	array_or_uspracement	a[0] to $a[i[0]$ -	1] $A(1)$ to $A(I(1))$	
/IPI_C	OMBINER_INDEXED_BLOCK	d[0]	D(1)	
MPI_C	OMBINER_INDEXED_BLOCK Constructor argument count blocklength array_of_displacement	d[0] ni = count+2, r C i[0] i[1] s i[2] to i[i[0]+	$D(1)$ $a = 0, nd = 1, and$ $\overline{Fortran \ location}$ $I(1)$ $I(2)$ $I(3) \ to \ I(I(1)+2)$	
ИРІ_С	OMBINER_INDEXED_BLOCK Constructor argument count blocklength	d[0] ni = count+2, r C i[0] i[1]	D(1) $a = 0, nd = 1, and$ $Fortran location$ $I(1)$ $I(2)$	
	OMBINER_INDEXED_BLOCK Constructor argument count blocklength array_of_displacement oldtype OMBINER_HINDEXED_BLOCK	d[0] ni = count+2, r C i[0] i[1] s i[2] to i[i[0]+ d[0] ni = 2, na =	D(1) $na = 0, nd = 1, and$ Fortran location $I(1)$ $I(2)$ $I(3) to I(I(1)+2)$ $D(1)$ count, nd = 1, and	
	OMBINER_INDEXED_BLOCK Constructor argument count blocklength array_of_displacement oldtype	d[0] ni = count+2, r C $i[0]$ $i[1]$ s $i[2] to i[i[0]+$ $d[0]$ ni = 2, na = C	D(1) $Ia = 0, nd = 1, and$ $Fortran location$ $I(1)$ $I(2)$ $I(3) to I(I(1)+2)$ $D(1)$ $Count, nd = 1, and$ $Fortran location$	
	OMBINER_INDEXED_BLOCK Constructor argument count blocklength array_of_displacement oldtype OMBINER_HINDEXED_BLOCK Constructor argument count	d[0] $ni = count+2, n$ C $i[0]$ $i[1]$ s $i[2] to i[i[0]+$ $d[0]$ $ni = 2, na =$ C $i[0]$	D(1) $Ia = 0, nd = 1, and$ $Fortran location$ $I(1)$ $I(2)$ $I] I(3) to I(I(1)+2)$ $D(1)$ $Count, nd = 1, and$ $Fortran location$ $I(1)$	
	OMBINER_INDEXED_BLOCK Constructor argument count blocklength array_of_displacement oldtype OMBINER_HINDEXED_BLOCK Constructor argument count blocklength	d[0] ni = count+2, n C i[0] i[1] s i[2] to i[i[0]+ d[0] ni = 2, na = C i[0] i[1]	D(1) $a = 0, nd = 1, and$ Fortran location $I(1)$ $I(2)$ $I = I(3) to I(I(1)+2)$ $D(1)$ count, nd = 1, and Fortran location $I(1)$ $I(2)$	
	oldtype OMBINER_INDEXED_BLOCK Constructor argument count blocklength array_of_displacement oldtype OMBINER_HINDEXED_BLOCK Constructor argument count blocklength array_of_displacement oldtype	d[0] ni = count+2, n C $i[0]$ $i[1]$ s i[2] to i[i[0]+ d[0] ni = 2, na = C $i[0]$ $i[1]$ s a[0] to a[i[0]-	D(1) $a = 0, nd = 1, and$ $Fortran location$ $I(1)$ $I(2)$ $I] I(3) to I(I(1)+2)$ $D(1)$ $count, nd = 1, and$ $Fortran location$ $I(1)$ $I(2)$ $I] A(1) to A(I(1))$	
	OMBINER_INDEXED_BLOCK Constructor argument count blocklength array_of_displacement oldtype OMBINER_HINDEXED_BLOCK Constructor argument count blocklength	d[0] ni = count+2, n C i[0] i[1] s i[2] to i[i[0]+ d[0] ni = 2, na = C i[0] i[1]	D(1) $a = 0, nd = 1, and$ Fortran location $I(1)$ $I(2)$ $I = I(3) to I(I(1)+2)$ $D(1)$ count, nd = 1, and Fortran location $I(1)$ $I(2)$	
MPI_C	oldtype OMBINER_INDEXED_BLOCK Constructor argument count blocklength array_of_displacement oldtype OMBINER_HINDEXED_BLOCK Constructor argument count blocklength array_of_displacement oldtype	d[0] ni = count+2, n C i[0] i[1] s i[2] to i[i[0]+ d[0] ni = 2, na = C i[0] i[1] s a[0] to a[i[0]- d[0]	D(1) $D(1)$	
MPI_C	oldtype OMBINER_INDEXED_BLOCK Constructor argument count blocklength array_of_displacement oldtype OMBINER_HINDEXED_BLOCK Constructor argument count blocklength array_of_displacement oldtype	d[0] ni = count+2, n C i[0] i[1] s i[2] to i[i[0]+ d[0] ni = 2, na = C i[0] i[1] s a[0] to a[i[0]- d[0]	D(1) $D(1)$	
MPI_C	oldtype OMBINER_INDEXED_BLOCK Constructor argument count blocklength array_of_displacement oldtype OMBINER_HINDEXED_BLOCK Constructor argument count blocklength array_of_displacement oldtype OMBINER_HINDEXED_BLOCK Constructor argument count blocklength array_of_displacement oldtype OMBINER_STRUCT ni = court	d[0] ni = count+2, n C i[0] i[1] s i[2] to i[i[0]+ d[0] ni = 2, na = C i[0] i[1] s a[0] to a[i[0]- d[0] nt+1, na = coun C	D(1) $D(1)$	
MPI_C	oldtype OMBINER_INDEXED_BLOCK Constructor argument count blocklength array_of_displacement oldtype OMBINER_HINDEXED_BLOCK Constructor argument count blocklength array_of_displacement oldtype OMBINER_HINDEXED_BLOCK Constructor argument count blocklength array_of_displacement oldtype OMBINER_STRUCT ni = cours Constructor argument	d[0] ni = count+2, n C $i[0]$ i[1] s i[2] to i[i[0]+ d[0] ni = 2, na = C $i[0]$ i[1] s a[0] to a[i[0]- d[0] nt+1, na = coun C $i[0]$	D(1) $D(1)$ $D(1)$ $D(1)$ $D(1)$ $I(1)$ $I(2)$ $I = I, and$ $D(1)$ $D(1)$ $I(1)$ $I(2)$ $I = I, and$ $I(1)$	
MPI_C	oldtype OMBINER_INDEXED_BLOCK Constructor argument count blocklength array_of_displacement oldtype OMBINER_HINDEXED_BLOCK Constructor argument count blocklength array_of_displacement oldtype OMBINER_HINDEXED_BLOCK Constructor argument count blocklength array_of_displacement oldtype OMBINER_STRUCT ni = court Constructor argument count	d[0] ni = count+2, n C i[0] i[1] s i[2] to i[i[0]+ d[0] ni = 2, na = C i[0] i[1] s a[0] to a[i[0]- d[0] nt+1, na = coun C i[0] i[1] to i[i[0]]	D(1) $D(1)$ $D(1)$ $D(1)$ $D(1)$ $I(1)$ $I(2)$ $I = I, and$ $I(1)$ $I(1)$ $I(1)$ $I(1)$ $I(2)$ $I = I, and$ $I(1)$	

Unofficial Draft for Comment Only

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

MPI_COMBINER_SUBARRAY ni = 3*ndims+2, na = 0, nd = 1, and

MPI_COMBINER_DARRAY ni = 4*ndims+4, na = 0, nd = 1, and

Constructor argument	\mathbf{C}	Fortran location
size	i[0]	I(1)
rank	i[1]	I(2)
ndims	i[2]	I(3)
$\operatorname{array_of}_{\operatorname{gsizes}}$	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	\mathbf{C}	Fortran location
р	i[0]	I(1)
r	i[1]	I(2)

MPI_COMBINER_F90_COMPLEX ni = 2, na = 0, nd = 0, and

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

MPI_COMBINER_F90_INTEGER ni = 1, na = 0, nd = 0, and

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

MPI_COMBINER_RESIZED ni = 0, na = 2, nd = 1, and

Constructor argument	С	Fortran location
lb	a[0]	A(1)
extent	a[1]	A(2)
oldtype	d[0]	D(1)

Unofficial Draft for Comment Only

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

END DO ! create datatype for lower triangular part CALL MPI_TYPE_INDEXED(100, blocklen, disp, MPI_REAL, ltype, ierr) CALL MPI_TYPE_COMMIT(ltype, ierr) CALL MPI_SENDRECV(a, 1, ltype, myrank, 0, b, 1, & ltype, myrank, 0, MPI_COMM_WORLD, status, ierr) **Example 5.15** Transpose a matrix. REAL a(100,100), b(100,100) INTEGER row, xpose, myrank, ierr INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal INTEGER status(MPI_STATUS_SIZE) ! transpose matrix a onto b CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr) CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lb, sizeofreal, ierr) ! create datatype for one row CALL MPI_TYPE_VECTOR(100, 1, 100, MPI_REAL, row, ierr) ! create datatype for matrix in row-major order CALL MPI_TYPE_CREATE_HVECTOR(100, 1, sizeofreal, row, xpose, ierr) CALL MPI_TYPE_COMMIT(xpose, ierr) ! send matrix in row-major order and receive in column major order CALL MPI_SENDRECV(a, 1, xpose, myrank, 0, b, 100*100, & MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr) **Example 5.16** Another approach to the transpose problem: REAL a(100,100), b(100,100) INTEGER row, row1 INTEGER (KIND=MPI_ADDRESS_KIND) disp(2), lb, sizeofreal INTEGER myrank, ierr INTEGER status(MPI_STATUS_SIZE) CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr) ! transpose matrix a onto b

CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lb, sizeofreal, ierr)

1

2 3

5

6

9 10

11 12

13

14

15

16 17

18 19

20 21

22 23

24

25 26

27

28 29

30 31

32

33

34 35 36

37

38

39

40

41

42 43

44 45

46 47

```
1
\mathbf{2}
     ! create datatype for one row
3
     CALL MPI_TYPE_VECTOR(100, 1, 100, MPI_REAL, row, ierr)
4
5
     ! create datatype for one row, with the extent of one real number
6
     1b = 0
7
     CALL MPI_TYPE_CREATE_RESIZED(row, lb, sizeofreal, row1, ierr)
8
9
     CALL MPI_TYPE_COMMIT(row1, ierr)
10
11
     ! send 100 rows and receive in column major order
12
     CALL MPI_SENDRECV(a, 100, row1, myrank, 0, b, 100*100, &
13
                        MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
14
15
     Example 5.17 Use of MPI datatypes to manipulate an array of structures.
16
17
     struct Partstruct
^{18}
     ſ
19
                type; /* particle type */
        int
20
        double d[6]; /* particle coordinates */
21
               b[7]; /* some additional information */
        char
22
     };
23
^{24}
                           particle[1000];
     struct Partstruct
25
26
                   i, dest, tag;
     int
27
     MPI_Comm
                   comm;
28
29
30
     /* build datatype describing structure */
^{31}
32
     MPI_Datatype Particlestruct, Particletype;
33
     MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
34
                   blocklen[3] = {1, 6, 7};
     int
35
     MPI_Aint
                   disp[3];
36
     MPI_Aint
                   base, lb, sizeofentry;
37
38
39
     /* compute displacements of structure components */
40
41
     MPI_Get_address(particle, disp);
42
     MPI_Get_address(particle[0].d, disp+1);
43
     MPI_Get_address(particle[0].b, disp+2);
44
     base = disp[0];
45
     for (i=0; i < 3; i++) disp[i] = MPI_Aint_diff(disp[i], base);</pre>
46
47
     MPI_Type_create_struct(3, blocklen, disp, type, &Particlestruct);
48
```

```
1
/* Since the compiler may pad the structure, it is best to explicitly
                                                                                       \mathbf{2}
   set the extent of the MPI datatype for a structure element using
                                                                                       3
   MPI_Type_create_resized */
                                                                                       \mathbf{4}
/* compute extent of the structure */
                                                                                       5
                                                                                       6
MPI_Get_address(particle+1, &sizeofentry);
sizeofentry = MPI_Aint_diff(sizeofentry, base);
                                                                                       7
/* build datatype describing structure */
                                                                                       9
                                                                                      10
MPI_Type_create_resized(Particlestruct, 0, sizeofentry, &Particletype);
                                                                                      11
                                                                                      12
/* 4.1: send the entire array */
                                                                                      13
                                                                                      14
                                                                                      15
MPI_Type_commit(&Particletype);
                                                                                      16
MPI_Send(particle, 1000, Particletype, dest, tag, comm);
                                                                                      17
                                                                                      18
                                                                                      19
/* 4.2: send only the entries of type zero particles,
        preceded by the number of such entries */
                                                                                      20
                                                                                      21
                             /* datatype describing all particles
                                                                                      22
MPI_Datatype Zparticles;
                                                                                      23
                                 with type zero (needs to be recomputed
                                                                                      24
                                 if types change) */
                                                                                      25
MPI_Datatype Ztype;
                                                                                      26
              zdisp[1000];
                                                                                      27
int
              zblock[1000], j, k;
                                                                                      28
int
                                                                                      29
int
              zzblock[2] = \{1,1\};
                                                                                      30
              zzdisp[2];
MPI_Aint
                                                                                      31
MPI_Datatype zztype[2];
                                                                                      32
                                                                                      33
/* compute displacements of type zero particles */
                                                                                      34
j = 0;
for (i=0; i < 1000; i++)</pre>
                                                                                      35
   if (particle[i].type == 0)
                                                                                      36
                                                                                      37
      {
        zdisp[j] = i;
                                                                                      38
                                                                                      39
        zblock[j] = 1;
        j++;
                                                                                      40
                                                                                      41
      }
                                                                                      42
/* create datatype for type zero particles */
                                                                                      43
                                                                                      44
MPI_Type_indexed(j, zblock, zdisp, Particletype, &Zparticles);
                                                                                      45
                                                                                      46
/* prepend particle count */
                                                                                      47
MPI_Get_address(&j, zzdisp);
                                                                                      48
MPI_Get_address(particle, zzdisp+1);
```

```
1
     zztype[0] = MPI_INT;
\mathbf{2}
     zztype[1] = Zparticles;
3
     MPI_Type_create_struct(2, zzblock, zzdisp, zztype, &Ztype);
4
\mathbf{5}
     MPI_Type_commit(&Ztype);
6
     MPI_Send(MPI_BOTTOM, 1, Ztype, dest, tag, comm);
\overline{7}
8
9
     /* A probably more efficient way of defining Zparticles */
10
11
     /* consecutive particles with index zero are handled as one block */
12
     i=0;
13
     for (i=0; i < 1000; i++)
14
        if (particle[i].type == 0)
15
           {
16
               for (k=i+1; (k < 1000)&&(particle[k].type == 0); k++);</pre>
17
               zdisp[j] = i;
18
               zblock[j] = k-i;
19
               j++;
20
               i = k:
21
           }
     MPI_Type_indexed(j, zblock, zdisp, Particletype, &Zparticles);
22
23
^{24}
25
     /* 4.3: send the first two coordinates of all entries */
26
27
     MPI_Datatype Allpairs;
                                    /* datatype for all pairs of coordinates */
28
29
     MPI_Type_get_extent(Particletype, &lb, &sizeofentry);
30
^{31}
     /* sizeofentry can also be computed by subtracting the address
32
        of particle[0] from the address of particle[1] */
33
34
     MPI_Type_create_hvector(1000, 2, sizeofentry, MPI_DOUBLE, &Allpairs);
     MPI_Type_commit(&Allpairs);
35
36
     MPI_Send(particle[0].d, 1, Allpairs, dest, tag, comm);
37
     /* an alternative solution to 4.3 */
38
39
40
     MPI_Datatype Twodouble;
41
42
     MPI_Type_contiguous(2, MPI_DOUBLE, &Twodouble);
43
44
     MPI_Datatype Onepair;
                               /* datatype for one pair of coordinates, with
45
                                 the extent of one particle entry */
46
47
     MPI_Type_create_resized(Twodouble, 0, sizeofentry, &Onepair );
48
     MPI_Type_commit(&Onepair);
```

MPI_Send(particle[0].d, 1000, Onepair, dest, tag, comm);

Example 5.18 The same manipulations as in the previous example, but use absolute addresses in datatypes.

```
struct Partstruct
                                                                                       8
{
                                                                                       9
    int
            type;
                                                                                       10
    double d[6];
                                                                                       11
    char
           b[7];
                                                                                       12
};
                                                                                       13
                                                                                       14
struct Partstruct particle[1000];
                                                                                       15
                                                                                       16
/* build datatype describing first array entry */
                                                                                       17
                                                                                       18
MPI_Datatype Particletype;
                                                                                       19
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
                                                                                       20
int
              block[3] = \{1, 6, 7\};
                                                                                      21
MPI_Aint
              disp[3];
                                                                                      22
                                                                                      23
MPI_Get_address(particle, disp);
                                                                                       ^{24}
MPI_Get_address(particle[0].d, disp+1);
                                                                                       25
MPI_Get_address(particle[0].b, disp+2);
                                                                                       26
MPI_Type_create_struct(3, block, disp, type, &Particletype);
                                                                                       27
                                                                                       28
/* Particletype describes first array entry -- using absolute
                                                                                      29
   addresses */
                                                                                       30
                                                                                       31
/* 5.1: send the entire array */
                                                                                       32
                                                                                       33
MPI_Type_commit(&Particletype);
                                                                                      34
MPI_Send(MPI_BOTTOM, 1000, Particletype, dest, tag, comm);
                                                                                      35
                                                                                      36
                                                                                      37
/* 5.2: send the entries of type zero,
                                                                                       38
        preceded by the number of such entries */
                                                                                       39
                                                                                       40
MPI_Datatype Zparticles, Ztype;
                                                                                       41
                                                                                       42
int
              zdisp[1000];
                                                                                       43
              zblock[1000], i, j, k;
int
                                                                                       44
              zzblock[2] = \{1,1\};
int
                                                                                       45
MPI_Datatype zztype[2];
                                                                                       46
MPI_Aint
              zzdisp[2];
                                                                                       47
                                                                                       48
```

1

2 3 4

5

```
1
     j=0;
\mathbf{2}
     for (i=0; i < 1000; i++)
3
         if (particle[i].type == 0)
4
              {
                  for (k=i+1; (k < 1000)&&(particle[k].type == 0); k++);</pre>
5
6
                  zdisp[j] = i;
7
                  zblock[j] = k-i;
8
                  j++;
9
                  i = k;
10
              }
11
     MPI_Type_indexed(j, zblock, zdisp, Particletype, &Zparticles);
12
     /* Zparticles describe particles with type zero, using
13
        their absolute addresses*/
14
15
     /* prepend particle count */
16
     MPI_Get_address(&j, zzdisp);
17
     zzdisp[1] = (MPI_Aint)0;
18
     zztype[0] = MPI_INT;
19
     zztype[1] = Zparticles;
20
     MPI_Type_create_struct(2, zzblock, zzdisp, zztype, &Ztype);
21
22
     MPI_Type_commit(&Ztype);
23
     MPI_Send(MPI_BOTTOM, 1, Ztype, dest, tag, comm);
^{24}
25
26
     Example 5.19 This example shows how datatypes can be used to handle unions.
27
28
     union {
29
        int
                 ival;
30
        float
                 fval;
^{31}
            } u[1000];
32
33
     int
              i, utype;
34
35
     /* All entries of u have identical type; variable
36
        utype keeps track of their current type */
37
38
     MPI_Datatype
                     mpi_utype[2];
39
     MPI_Aint
                     ubase, extent;
40
41
     /* compute an MPI datatype for each possible union type;
42
        assume values are left-aligned in union storage. */
43
44
     MPI_Get_address(u, &ubase);
45
     MPI_Get_address(u+1, &extent);
46
     extent = MPI_Aint_diff(extent, ubase);
47
48
```

```
1
MPI_Type_create_resized(MPI_INT, 0, extent, &mpi_utype[0]);
                                                                                      2
MPI_Type_create_resized(MPI_FLOAT, 0, extent, &mpi_utype[1]);
for(i=0; i<2; i++) MPI_Type_commit(&mpi_utype[i]);</pre>
                                                                                      5
                                                                                      6
/* actual communication */
MPI_Send(u, 1000, mpi_utype[utype], dest, tag, comm);
                                                                                      9
                                                                                     10
Example 5.20 This example shows how a datatype can be decoded. The routine
                                                                                     11
printdatatype prints out the elements of the datatype. Note the use of MPI_Type_free for
                                                                                     12
datatypes that are not predefined.
                                                                                     13
                                                                                     14
/*
                                                                                     15
  Example of decoding a datatype.
                                                                                     16
                                                                                     17
  Returns 0 if the datatype is predefined, 1 otherwise
                                                                                     18
 */
                                                                                     19
#include <stdio.h>
#include <stdlib.h>
                                                                                     20
                                                                                     21
#include "mpi.h"
                                                                                     22
int printdatatype(MPI_Datatype datatype)
                                                                                     23
{
                                                                                     24
    int *array_of_ints;
                                                                                     25
    MPI_Aint *array_of_adds;
                                                                                     26
    MPI_Datatype *array_of_dtypes;
    int num_ints, num_adds, num_dtypes, combiner;
                                                                                     27
                                                                                     28
    int i;
                                                                                     29
    MPI_Type_get_envelope(datatype,
                                                                                     30
                                                                                     31
                            &num_ints, &num_adds, &num_dtypes, &combiner);
                                                                                     32
    switch (combiner) {
                                                                                     33
    case MPI_COMBINER_NAMED:
                                                                                     34
        printf("Datatype is named:");
        /* To print the specific type, we can match against the
                                                                                     35
                                                                                     36
            predefined forms. We can NOT use a switch statement here
                                                                                     37
           We could also use MPI_TYPE_GET_NAME if we prefered to use
                                                                                     38
           names that the user may have changed.
                                                                                     39
         */
                                                                                     40
        if
                 (datatype == MPI_INT)
                                            printf("MPI_INT\n");
                                                                                     41
        else if (datatype == MPI_DOUBLE) printf("MPI_DOUBLE\n");
                                                                                     42
         ... else test for other types ...
                                                                                     43
        return 0;
                                                                                     44
        break;
    case MPI_COMBINER_STRUCT:
                                                                                     45
                                                                                     46
    case MPI_COMBINER_STRUCT_INTEGER:
                                                                                     47
        printf("Datatype is struct containing");
                                                                                     48
        array_of_ints = (int *)malloc(num_ints * sizeof(int));
```

```
1
             array_of_adds
2
                          (MPI_Aint *) malloc(num_adds * sizeof(MPI_Aint));
3
             array_of_dtypes = (MPI_Datatype *)
4
                  malloc(num_dtypes * sizeof(MPI_Datatype));
5
             MPI_Type_get_contents(datatype, num_ints, num_adds, num_dtypes,
6
                                  array_of_ints, array_of_adds, array_of_dtypes);
7
             printf(" %d datatypes:\n", array_of_ints[0]);
8
             for (i=0; i<array_of_ints[0]; i++) {</pre>
9
                  printf("blocklength %d, displacement %ld, type:\n",
10
                          array_of_ints[i+1], (long)array_of_adds[i]);
11
                  if (printdatatype(array_of_dtypes[i])) {
12
                      /* Note that we free the type ONLY if it
13
                         is not predefined */
14
                      MPI_Type_free(&array_of_dtypes[i]);
15
                  }
16
             }
17
             free(array_of_ints);
18
             free(array_of_adds);
19
             free(array_of_dtypes);
20
             break:
21
              ... other combiner values ...
22
         default:
23
             printf("Unrecognized combiner type\n");
24
         }
25
         return 1;
26
     }
27
```

```
5.2 Pack and Unpack
```

30 Some existing communication libraries provide pack/unpack functions for sending noncon- 31 tiguous data. In these, the user explicitly packs data into a contiguous buffer before sending 32 it, and unpacks it from a contiguous buffer after receiving it. Derived datatypes, which are 33 described in Section 5.1, allow one, in most cases, to avoid explicit packing and unpacking. 34 The user specifies the layout of the data to be sent or received, and the communication 35 library directly accesses a noncontiguous buffer. The pack/unpack routines are provided 36 for compatibility with previous libraries. Also, they provide some functionality that is not 37 otherwise available in MPI. For instance, a message can be received in several parts, where 38 the receive operation done on a later part may depend on the content of a former part. 39 Another use is that outgoing messages may be explicitly buffered in user supplied space, 40 thus overriding the system buffering policy. Finally, the availability of pack and unpack 41 operations facilitates the development of additional communication libraries layered on top 42of MPI. 43

44

28

29

- 45
- 46
- 47

MPL PACK (inclust datature output outsize position comm)				
m 1_1 ACR(mbul, mcount, datatype, outbul, outsize, position, comm)				
IN	inbuf	input buffer start (choice)	2 3	
IN	incount	number of input data items (non-negative integer)	4	
IN	datatype	datatype of each input data item (handle)	5	
OUT	outbuf	output buffer start (choice)	6	
IN	outsize	output buffer size, in bytes (non-negative integer)	7	
INOUT			8 9	
	position	current position in buffer, in bytes (integer)	9 10	
IN	comm	communicator for packed message (handle)	11	
C hindin	a.		12	
C bindin	•	nt incount MPI Datatype datatype	13	
<pre>int MPI_Pack(const void *inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outsize, int *position, MPI_Comm comm)</pre>			14	
int MDT I			15	
int MPI_P		MPI_Count incount, MPI_Datatype datatype,	16 17	
<pre>void *outbuf, MPI_Count outsize, MPI_Count *position, MPI_Comm comm)</pre>		Juit outbize, in i_oount .pobleton,	18	
Fortran 2008 binding MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror)			20	
	(*), DIMENSION(), INTEN		21	
	GER, INTENT(IN) :: incoun		22	
TYPE(MPI Datatype), INTENT(IN) :: datatype			23 24	
TYPE(*), DIMENSION() :: outbuf			25	
INTEGER, INTENT(INOUT) :: position			26	
	(MPI_Comm), INTENT(IN) ::		27	
	GER, OPTIONAL, INTENT(OUT		28	
		, outbuf, outsize, position, comm, ierror)	29	
	(*), DIMENSION(), INTEN		30	
		INTENT(IN) :: incount, outsize	31 32	
TYPE(MPI_Datatype), INTENT(IN) :: datatype32TYPE(*), DIMENSION() :: outbuf33				
INTEGER(KIND=MPI_COUNT_KIND), INTENT(INOUT) :: position 34				
			35	
INTEC	INTEGER, OPTIONAL, INTENT(OUT) :: ierror ³			
Fortran binding			37	
MPT PACK(INBUE, INCOUNT, DATATYPE, OUTBUE, OUTSIZE, POSITION, COMM, IERROR)			38	
<tvpe> INBUF(*). OUTBUF(*)</tvpe>			$\frac{39}{40}$	
INTEC	INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR			
Packs the message in the send buffer specified by inbuf, incount, datatype into the buffer			41 42	
space specified by outbuf and outsize. The input buffer can be any communication buffer				

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). 42 43 44 45 46

The input value of **position** is the first location in the output buffer to be used for packing. **position** is incremented by the size of the packed message, and the output value 48

Unofficial Draft for Comment Only

of position is the first location in the output buffer following the locations occupied by the $\mathbf{2}$ packed message. The comm argument is the communicator that will be subsequently used 3 for sending the packed message. 4 5MPI_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm) 6 7 IN inbuf input buffer start (choice) 8 IN insize size of input buffer, in bytes (non-negative integer) 9 INOUT position current position in bytes (integer) 10 11 OUT outbuf output buffer start (choice) 12IN outcount number of items to be unpacked (integer) 13 datatype of each output data item (handle) IN datatype 1415communicator for packed message (handle) IN comm 1617C binding 18 int MPI_Unpack(const void *inbuf, int insize, int *position, void *outbuf, 19 int outcount, MPI_Datatype datatype, MPI_Comm comm) 20int MPI_Unpack_c(const void *inbuf, MPI_Count insize, MPI_Count *position, 21void *outbuf, MPI_Count outcount, MPI_Datatype datatype, 22 MPI Comm comm) 23 24 Fortran 2008 binding 25MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm, 26ierror) 27TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf 28INTEGER, INTENT(IN) :: insize, outcount 29 INTEGER, INTENT(INOUT) :: position 30 TYPE(*), DIMENSION(..) :: outbuf 31TYPE(MPI_Datatype), INTENT(IN) :: datatype 32 TYPE(MPI_Comm), INTENT(IN) :: comm 33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 34 MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm, 35 ierror) 36 TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf 37 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: insize, outcount 38 INTEGER(KIND=MPI_COUNT_KIND), INTENT(INOUT) :: position 39 TYPE(*), DIMENSION(..) :: outbuf 40 TYPE(MPI_Datatype), INTENT(IN) :: datatype 41 TYPE(MPI_Comm), INTENT(IN) :: comm 42INTEGER, OPTIONAL, INTENT(OUT) :: ierror 43 44Fortran binding 45MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM, 46 IERROR) 47 <type> INBUF(*), OUTBUF(*)

1

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 ⁴⁶ substring of a packing unit necessarily a packing unit. Thus, one cannot concatenate two ⁴⁷ packing units and then unpack the result as one packing unit; nor can one unpack a substring ⁴⁸

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6

 $\overline{7}$

8

9

10 11

12

13

14

15

16

17

18

19

20

21

22

23 24

25

26

27

28 29

30

 31

32 33

34

35 36

37

38

39

40 41

42

43 44

count argument to packing call (non-negative integer)

of a packing unit as a separate packing unit. Each packing unit, that was created by a related
 sequence of pack calls, or by a regular send, must be unpacked as a unit, by a sequence of
 related unpack calls.

Rationale. The restriction on "atomic" packing and unpacking of packing units allows the implementation to add at the head of packing units additional information, such as a description of the sender architecture (to be used for type conversion, in a heterogeneous environment) (*End of rationale.*)

The following call allows the user to find out how much space is needed to pack a message and, thus, manage space allocation for buffers.

```
12
13
```

IN

5

6

7

8 9

10

11

14

25 26

31

32

33

34

43

44

MPI_PACK_SIZE(incount, datatype, comm, size)

incount

15			
16	IN	datatype	datatype argument to packing call (handle)
17	IN	comm	communicator argument to packing call (handle)
18	OUT	size	upper bound on size of packed message, in bytes
19			(non-negative integer)
20			(
21	Chindin	~	
22	C binding	0	
23	int MPL_F		<pre>MPI_Datatype datatype, MPI_Comm comm,</pre>
24		int *size)	

Fortran 2008 binding

```
MPI_Pack_size(incount, datatype, comm, size, ierror)
INTEGER, INTENT(IN) :: incount
```

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

TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, INTENT(OUT) :: size

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

```
    MPI_Pack_size(incount, datatype, comm, size, ierror)
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: incount
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

41 42 Fortran binding

```
MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)
INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR
```

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];
                   /* larger than sizeof(int) + 200 * sizeof(float) */
char buff[1000];
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0)
{
    /* SENDER CODE */
   int len[2];
    MPI_Aint disp[2];
    MPI_Datatype type[2], newtype;
    /* build datatype for i followed by a[0]...a[i-1] */
    len[0] = 1;
    len[1] = i;
    MPI_Get_address(&i, disp);
    MPI_Get_address(a, disp+1);
    type[0] = MPI_INT;
    type[1] = MPI_FLOAT;
    MPI_Type_create_struct(2, len, disp, type, &newtype);
    MPI_Type_commit(&newtype);
```

1

 $\mathbf{2}$

3

 $\frac{4}{5}$

6 7

8

9 10

11

12

13

14 15

16

17

18

19

20

21

22 23

24 25

26

27

28

29 30

31

32

33 34

35

36

37

38 39 40

41

42

43 44

45

46

47

```
1
         /* Pack i followed by a[0]...a[i-1]*/
\mathbf{2}
3
         position = 0;
4
         MPI_Pack(MPI_BOTTOM, 1, newtype, buff, 1000, &position, MPI_COMM_WORLD);
5
6
         /* Send */
7
8
         MPI_Send(buff, position, MPI_PACKED, 1, 0,
9
                   MPI_COMM_WORLD);
10
11
     /* ****
12
        One can replace the last three lines with
13
        MPI_Send(MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
14
        **** */
15
     }
16
     else if (myrank == 1)
17
     {
18
         /* RECEIVER CODE */
19
20
         MPI_Status status;
21
22
         /* Receive */
23
^{24}
         MPI_Recv(buff, 1000, MPI_PACKED, 0, 0, MPI_COMM_WORLD, &status);
25
26
         /* Unpack i */
27
28
         position = 0;
         MPI_Unpack(buff, 1000, &position, &i, 1, MPI_INT, MPI_COMM_WORLD);
29
30
^{31}
         /* Unpack a[0]...a[i-1] */
32
         MPI_Unpack(buff, 1000, &position, a, i, MPI_FLOAT, MPI_COMM_WORLD);
33
     }
34
35
     Example 5.23 Each process sends a count, followed by count characters to the root; the
36
     root concatenates all characters into one string.
37
38
          count, gsize, counts[64], totalcount, k1, k2, k,
     int
39
           displs[64], position, concat_pos;
40
     char chr[100], *lbuf, *rbuf, *cbuf;
41
42
     MPI_Comm_size(comm, &gsize);
43
     MPI_Comm_rank(comm, &myrank);
44
45
            /* allocate local pack buffer */
46
     MPI_Pack_size(1, MPI_INT, comm, &k1);
47
     MPI_Pack_size(count, MPI_CHAR, comm, &k2);
48
     k = k1 + k2;
```

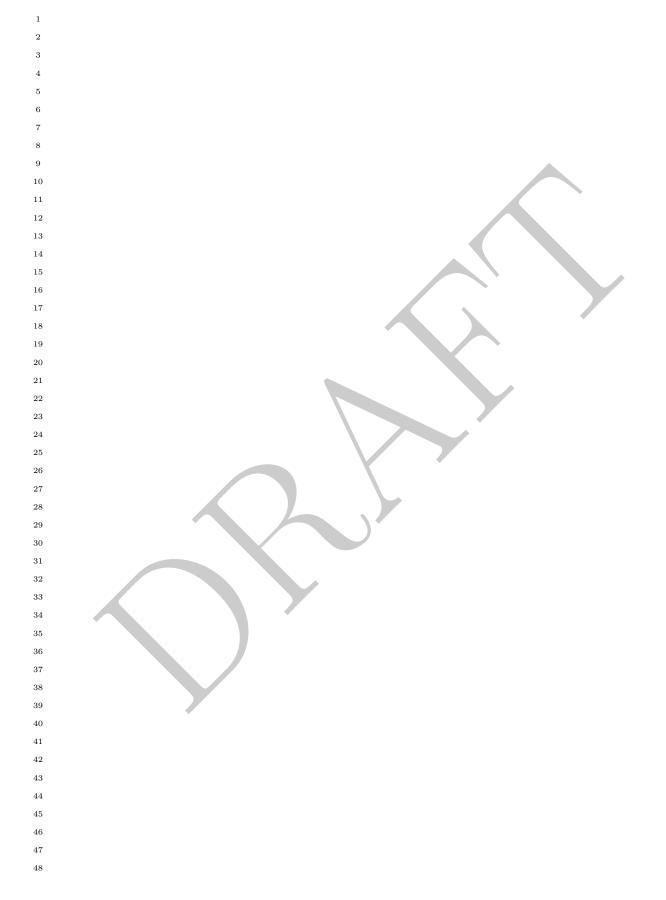
```
1
lbuf = (char *)malloc(k);
                                                                                      \mathbf{2}
                                                                                      3
      /* pack count, followed by count characters */
position = 0;
                                                                                      4
MPI_Pack(&count, 1, MPI_INT, lbuf, k, &position, comm);
                                                                                      5
                                                                                      6
MPI_Pack(chr, count, MPI_CHAR, lbuf, k, &position, comm);
                                                                                      7
if (myrank != root) {
                                                                                      9
    /* gather at root sizes of all packed messages */
                                                                                     10
    MPI_Gather(&position, 1, MPI_INT, NULL, 0,
                                                                                     11
                MPI_DATATYPE_NULL, root, comm);
                                                                                     12
    /* gather at root packed messages */
                                                                                     13
    MPI_Gatherv(lbuf, position, MPI_PACKED, NULL,
                                                                                     14
                                                                                     15
                 NULL, NULL, MPI_DATATYPE_NULL, root, comm);
                                                                                     16
                                                                                     17
} else {
          /* root code */
                                                                                     18
    /* gather sizes of all packed messages */
    MPI_Gather(&position, 1, MPI_INT, counts, 1,
                                                                                     19
                MPI_INT, root, comm);
                                                                                     20
                                                                                     21
    /* gather all packed messages *
                                                                                     22
    displs[0] = 0;
                                                                                     23
                                                                                     ^{24}
    for (i=1; i < gsize; i++)</pre>
        displs[i] = displs[i-1] + counts[i-1];
                                                                                     25
                                                                                     26
    totalcount = displs[gsize-1] + counts[gsize-1];
    rbuf = (char *)malloc(totalcount);
                                                                                     27
    cbuf = (char *)malloc(totalcount);
                                                                                     28
                                                                                     29
    MPI_Gatherv(lbuf, position, MPI_PACKED, rbuf,
                                                                                     30
                 counts, displs, MPI_PACKED, root, comm);
                                                                                     31
    /* unpack all messages and concatenate strings */
                                                                                     32
                                                                                     33
    concat_pos = 0;
                                                                                     34
   for (i=0; i < gsize; i++) {</pre>
        position = 0;
                                                                                     35
        MPI_Unpack(rbuf+displs[i], totalcount-displs[i],
                                                                                     36
                                                                                     37
                    &position, &count, 1, MPI_INT, comm);
        MPI_Unpack(rbuf+displs[i], totalcount-displs[i],
                                                                                     38
                    &position, cbuf+concat_pos, count, MPI_CHAR, comm);
                                                                                     39
                                                                                     40
        concat_pos += count;
                                                                                     41
    }
                                                                                     42
    cbuf[concat_pos] = '\0';
}
                                                                                     43
                                                                                     44
                                                                                     45
```

5.3 Ca	nonical MPI_PAC	K and MPI_UNPACK					
in Section the data f	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 datarep argument is "external32".						
		Inctions could be used, for example, to send typed data in a MPI implementation to another. (<i>End of advice to users.</i>)					
		actly the packed data, without headers. MPI_BYTE should a that is packed using MPI_PACK_EXTERNAL.					
and allow	further specifies the wed to) use a header,	EXTERNAL specifies that there is no header on the message exact format of the data. Since MPI_PACK may (and is the datatype MPI_PACKED cannot be used for data packed NAL. (<i>End of rationale.</i>)					
MPI_PAC	K_EXTERNAL(datarep	o, inbuf, incount, datatype, outbuf, outsize, position)					
IN	datarep	data representation (string)					
IN	inbuf	input buffer start (choice)					
IN	incount	number of input data items (integer)					
IN	datatype	datatype of each input data item (handle)					
OUT	outbuf	output buffer start (choice)					
IN	outsize	output buffer size, in bytes (integer)					
INOUT	position	current position in buffer, in bytes (integer)					
	Pack_external(cons MPI_Datatype MPI_Aint *pos Pack_external_c(con	t char datarep[], const void *inbuf, int incount, datatype, void *outbuf, MPI_Aint outsize, ition) nst char datarep[], const void *inbuf, count, MPI_Datatype datatype, void *outbuf,					
		size, MPI_Count *position)					
	2008 binding						
MPI_Pack	_external(datarep, position, ier	inbuf, incount, datatype, outbuf, outsize,					
CHAR	ACTER(LEN=*), INTE						
	-	, INTENT(IN) :: inbuf					
	GER, INTENT(IN) ::						
	• -	TENT(IN) :: datatype					
	(*), DIMENSION()	:: outbuf SS_KIND), INTENT(IN) :: outsize					
		SS_KIND), INTENT(IN) :: OutSize SS_KIND), INTENT(INOUT) :: position					

INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1 2 MPI_Pack_external(datarep, inbuf, incount, datatype, outbuf, outsize, position, ierror) CHARACTER(LEN=*), INTENT(IN) :: datarep 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 10 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 11 Fortran binding 12MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, 13 14POSITION, IERROR) 15CHARACTER*(*) DATAREP <type> INBUF(*), OUTBUF(*) 1617 INTEGER INCOUNT, DATATYPE, IERROR 18 INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, POSITION 19 2021MPI_UNPACK_EXTERNAL(datarep, inbuf, insize, position, outbuf, outcount, datatype) 22 IN data representation (string) datarep 23IN inbuf input buffer start (choice) 2425IN insize input buffer size, in bytes (integer) 26INOUT position current position in buffer, in bytes (integer) 27OUT outbuf output buffer start (choice) 2829 IN number of output data items (integer) outcount 30 IN datatype datatype of output data item (handle) 3132 C binding 33 int MPI_Unpack_external(const char datarep[], const void *inbuf, 34 MPI_Aint insize, MPI_Aint *position, void *outbuf, 35 int outcount, MPI_Datatype datatype) 36 37 int MPI_Unpack_external_c(const char datarep[], const void *inbuf, 38 MPI_Count insize, MPI_Count *position, void *outbuf, 39 MPI_Count outcount, MPI_Datatype datatype) 40 Fortran 2008 binding 41 MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outcount, 42datatype, ierror) 43 CHARACTER(LEN=*), INTENT(IN) :: datarep 44 TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf 45INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: insize 46INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position 47TYPE(*), DIMENSION(..) :: outbuf 48

```
1
         INTEGER, INTENT(IN) :: outcount
\mathbf{2}
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outcount,
5
                   datatype, ierror)
6
         CHARACTER(LEN=*), INTENT(IN) :: datarep
7
         TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
8
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: insize, outcount
9
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(INOUT) :: position
10
         TYPE(*), DIMENSION(..) :: outbuf
11
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     Fortran binding
15
     MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT,
16
                   DATATYPE, IERROR)
17
         CHARACTER*(*) DATAREP
18
         <type> INBUF(*), OUTBUF(*)
19
         INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION
20
         INTEGER OUTCOUNT, DATATYPE, IERROR
21
22
23
     MPI_PACK_EXTERNAL_SIZE(datarep, incount, datatype, size)
24
                                           data representation (string)
       IN
                datarep
25
26
       IN
                incount
                                           number of input data items (integer)
27
       IN
                                           datatype of each input data item (handle)
                datatype
28
       OUT
                size
                                           output buffer size, in bytes (integer)
29
30
     C binding
^{31}
     int MPI_Pack_external_size(const char datarep[], int incount,
32
                   MPI_Datatype datatype, MPI_Aint *size)
33
34
     int MPI_Pack_external_size_c(const char datarep[], MPI_Count incount,
35
                   MPI_Datatype datatype, MPI_Count *size)
36
     Fortran 2008 binding
37
     MPI_Pack_external_size(datarep, incount, datatype, size, ierror)
38
         CHARACTER(LEN=*), INTENT(IN) :: datarep
39
         INTEGER, INTENT(IN) :: incount
40
41
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
42
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44
     MPI_Pack_external_size(datarep, incount, datatype, size, ierror)
45
         CHARACTER(LEN=*), INTENT(IN) :: datarep
46
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: incount
47
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
48
```

INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size $\mathbf{2}$ INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR) CHARACTER*(*) DATAREP INTEGER INCOUNT, DATATYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SIZE 24 31 44



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: Barrier synchronization across all members of a group (Section 6.3 and Section 6.12.1).
- MPI_BCAST, MPI_IBCAST: Broadcast from one member to all members of a group (Section 6.4 and Section 6.12.2). This is shown as "broadcast" in Figure 6.1.
- MPI_GATHER, MPI_IGATHER, MPI_GATHERV, MPI_IGATHERV: Gather data from all members of a group to one member (Section 6.5 and Section 6.12.3). This is shown as "gather" in Figure 6.1.
- MPI_SCATTER, MPI_ISCATTER, MPI_SCATTERV, MPI_ISCATTERV: Scatter data from one member to all members of a group (Section 6.6 and Section 6.12.4). This is shown as "scatter" in Figure 6.1.
- MPI_ALLGATHER, MPI_IALLGATHER, MPI_ALLGATHERV, MPI_IALLGATHERV: A variation on Gather where all members of a group receive the result (Section 6.7 and Section 6.12.5). This is shown as "allgather" in Figure 6.1.
- MPI_ALLTOALL, MPI_IALLTOALL, MPI_ALLTOALLV, MPI_IALLTOALLV, MPI_ALLTOALLW, MPI_IALLTOALLW, MPI_IALLTOALLW: Scatter/Gather data from all members to all members of a group (also called complete exchange) (Section 6.8 and Section 6.12.6). This is shown as "complete exchange" in Figure 6.1.
- MPI_ALLREDUCE, MPI_IALLREDUCE, MPI_REDUCE, MPI_IREDUCE: Global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to all members of a group (Section 6.9.6 and Section 6.12.8) and a variation where the result is returned to only one member (Section 6.9 and Section 6.12.7).
- MPI_REDUCE_SCATTER_BLOCK, MPI_IREDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, MPI_IREDUCE_SCATTER: A combined reduction and scatter operation (Section 6.10, Section 6.12.9, and Section 6.12.10).

3

4

• MPI_SCAN, MPI_ISCAN, MPI_EXSCAN, MPI_IEXSCAN: Scan across all members of a group (also called prefix) (Section 6.11, Section 6.11.2, Section 6.12.11, and Section 6.12.12).

One of the key arguments in a call to a collective routine is a communicator that 5defines the group or groups of participating processes and provides a context for the oper-6 ation. This is discussed further in Section 6.2. The syntax and semantics of the collective 7 operations are defined to be consistent with the syntax and semantics of the point-to-point 8 operations. Thus, general datatypes are allowed and must match between sending and re-9 ceiving processes as specified in Chapter 5. Several collective routines such as broadcast 10 and gather have a single originating or receiving process. Such a process is called the *root*. 11 Some arguments in the collective functions are specified as "significant only at root," and 12are ignored for all participants except the root. The reader is referred to Chapter 5 for 13 information concerning communication buffers, general datatypes and type matching rules, 14and to Chapter 7 for information on how to define groups and create communicators. 15

The type-matching conditions for the collective operations are more strict than the corresponding conditions between sender and receiver in point-to-point. Namely, for collective operations, the amount of data sent must exactly match the amount of data specified by the receiver. Different type maps (the layout in memory, see Section 5.1) between sender and receiver are still allowed.

Collective operations can (but are not required to) complete as soon as the caller's 21participation in the collective communication is finished. A blocking operation is complete 22 as soon as the call returns. A nonblocking (immediate) call requires a separate completion 23call (cf. Section 3.7). The completion of a collective operation indicates that the caller is free 24to modify locations in the communication buffer. It does not indicate that other processes 25in the group have completed or even started the operation (unless otherwise implied by the 26description of the operation). Thus, a collective communication operation may, or may not, 27have the effect of synchronizing all participating MPI processes. 28

Collective communication calls may use the same communicators as point-to-point communication; MPI guarantees that messages generated on behalf of collective communication calls will not be confused with messages generated by point-to-point communication. The collective operations do not have a message tag argument. A more detailed discussion of correct use of collective routines is found in Section 6.14.

 Rationale. The equal-data restriction (on type matching) was made so as to avoid the complexity of providing a facility analogous to the status argument of MPI_RECV for discovering the amount of data sent. Some of the collective routines would require an array of status values.

The statements about synchronization are made so as to allow a variety of implementations of the collective functions.

(End of rationale.)

Advice to users. It is dangerous to rely on synchronization side-effects of the col lective operations for program correctness. For example, even though a particular
 implementation may provide a broadcast routine with a side-effect of synchroniza tion, the standard does not require this, and a program that relies on this will not be
 portable.

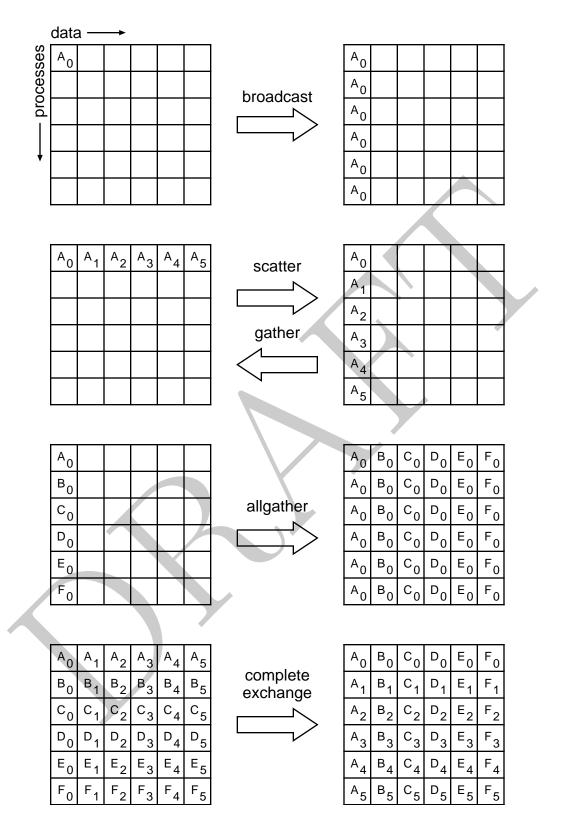


Figure 6.1: Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially just the first process contains the data A_0 , but after the broadcast all processes contain it.

Unofficial Draft for Comment Only

 31

 $44 \\ 45$

On the other hand, a correct, portable program must allow for the fact that a collective call *may* be synchronizing. Though one cannot rely on any synchronization side-effect, one must program so as to allow it. These issues are discussed further in Section 6.14. (*End of advice to users.*)

Advice to implementors. While vendors may write optimized collective routines matched to their architectures, a complete library of the collective communication routines can be written entirely using the MPI point-to-point communication functions and a few auxiliary functions. If implementing on top of point-to-point, a hidden, special communicator might be created for the collective operation so as to avoid interference with any on-going point-to-point communication at the time of the collective call. This is discussed further in Section 6.14. (End of advice to implementors.)

¹³ Many of the descriptions of the collective routines provide illustrations in terms of ¹⁴ blocking MPI point-to-point routines. These are intended solely to indicate what data is ¹⁵ sent or received by what process. Many of these examples are *not* correct MPI programs; ¹⁶ for purposes of simplicity, they often assume infinite buffering.

6.2 Communicator Argument

The key concept of the collective functions is to have a group or groups of participating processes. The routines do not have group identifiers as explicit arguments. Instead, there is a communicator argument. Groups and communicators are discussed in full detail in Chapter 7. For the purposes of this chapter, it is sufficient to know that there are two types of communicators: *intra-communicators* and *inter-communicators*. An intra-communicator can be thought of as an identifier for a single group of processes linked with a context. An inter-communicator identifies two distinct groups of processes linked with a context.

²⁸ 6.2.1 Specifics for Intra-Communicator Collective Operations

All processes in the group identified by the intra-communicator must call the collective routine.

In many cases, collective communication can occur "in place" for intra-communicators, with the output buffer being identical to the input buffer. This is specified by providing a special argument value, MPI_IN_PLACE, instead of the send buffer or the receive buffer argument, depending on the operation performed.

36 The "in place" operations are provided to reduce unnecessary memory Rationale. 37 motion by both the MPI implementation and by the user. Note that while the simple 38 check of testing whether the send and receive buffers have the same address will 39 work for some cases (e.g., MPI_ALLREDUCE), they are inadequate in others (e.g., 40 MPI_GATHER, with root not equal to zero). Further, Fortran explicitly prohibits 41 aliasing of arguments; the approach of using a special value to denote "in place" 42operation eliminates that difficulty. (End of rationale.) 43

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.

⁴⁷ Note that MPI_IN_PLACE is a special kind of value; it has the same restrictions on its ⁴⁸ use that MPI_BOTTOM has. (*End of advice to users.*)

Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8

9

10

11

12

17 18

6.2.2 Applying Collective Operations to Inter-Communicators	1
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]):	2 3 4 5
All-To-All All processes contribute to the result. All processes receive the result.	6
 MPI_ALLGATHER, MPI_IALLGATHER, MPI_ALLGATHERV, MPI_IALLGATHERV 	7 8 9
 MPI_ALLTOALL, MPI_IALLTOALL, MPI_ALLTOALLV, MPI_IALLTOALLV, MPI_ALLTOALLW, MPI_IALLTOALLW 	10 11
 MPI_ALLREDUCE, MPI_IALLREDUCE, MPI_REDUCE_SCATTER_BLOCK, MPI_IREDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, MPI_IREDUCE_SCATTER 	12 13 14 15
MPI_BARRIER, MPI_IBARRIER	16
All-To-One All processes contribute to the result. One process receives the result.	17
 MPI_GATHER, MPI_IGATHER, MPI_GATHERV, MPI_IGATHERV MPI_REDUCE, MPI_IREDUCE 	18 19 20
One-To-All One process contributes to the result. All processes receive the result.	21 22
MPI_BCAST, MPI_IBCAST	23
 MPI_SCATTER, MPI_ISCATTER, MPI_SCATTERV, MPI_ISCATTERV 	24 25
Other Collective operations that do not fit into one of the above categories.	26
 MPI_SCAN, MPI_ISCAN, MPI_EXSCAN, MPI_IEXSCAN 	27 28
The data movement patterns of MPI_SCAN, MPI_ISCAN, MPI_EXSCAN, and MPI_IEXSCAN do not fit this taxonomy.	29 30
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	31 32
described as collecting data from all members of one group with the result appearing in all	33 34
members of the other group (see Figure 6.2). As another example, a one-to-all	35
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	36
similar interpretation (see Figure 6.3). For intra-communicators, these two groups are the	37 38
same. For inter-communicators, these two groups are distinct. For the all-to-all operations,	39
each such operation is described in two phases, so that it has a symmetric, full-duplex behavior.	40
The following collective operations also apply to inter-communicators:	41 42
MPI_BARRIER, MPI_IBARRIER	43
 MPI_BCAST, MPI_IBCAST 	44
	45 46
• MPI_GATHER, MPI_IGATHER, MPI_GATHERV, MPI_IGATHERV,	47
 MPI_SCATTER, MPI_ISCATTER, MPI_SCATTERV, MPI_ISCATTERV, 	48

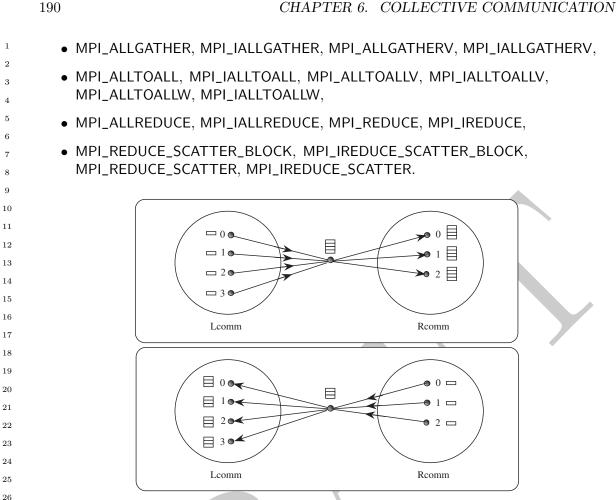


Figure 6.2: Inter-communicator allgather. The focus of data to one process is represented, not mandated by the semantics. The two phases do allgathers in both directions.

2930 31

32

33

27

28

6.2.3 Specifics for Inter-Communicator Collective Operations

All processes in both groups identified by the inter-communicator must call the collective routine.

34 Note that the "in place" option for intra-communicators does not apply to inter-35 communicators since in the inter-communicator case there is no communication from a 36 process to itself. 37

For inter-communicator collective communication, if the operation is in the All-To-One 38 or One-To-All categories, then the transfer is unidirectional. The direction of the transfer is 39 indicated by a special value of the root argument. In this case, for the group containing the 40 root process, all processes in the group must call the routine using a special argument for 41 the root. For this, the root process uses the special root value MPI_ROOT; all other processes 42in the same group as the root use MPI_PROC_NULL. All processes in the other group (the 43 group that is the remote group relative to the root process) must call the collective routine 44 and provide the rank of the root. If the operation is in the All-To-All category, then the 45 transfer is bidirectional. 46

47Rationale. Operations in the All-To-One and One-To-All categories are unidirectional 48 by nature, and there is a clear way of specifying direction. Operations in the All-To-All

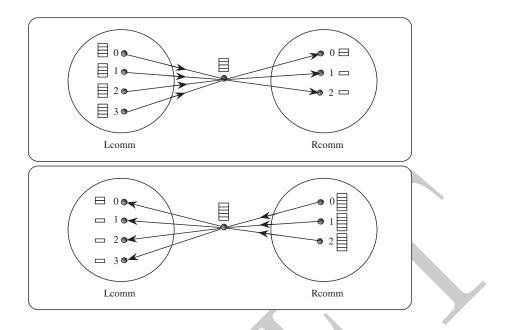


Figure 6.3: Inter-communicator reduce-scatter. The focus of data to one process is represented, not mandated by the semantics. The two phases do reduce-scatters in both directions.

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_BARRI	ER(comm)			
IN	comm	co	ommunicator (handle)	
C binding				
<pre>int MPI_Barrier(MPI_Comm comm)</pre>				
Fortran 20	08 binding			
MPI_Barrier(comm, ierror)				
TYPE(M	PI_Comm), INTEN	T(IN) :: co	mm	
INTEGE	R, OPTIONAL, IN	TENT(OUT) :	: ierror	

Fortran binding

MPI_BARRIER(COMM, IERROR) INTEGER COMM, IERROR

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.

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

Unofficial Draft for Comment Only

 41

```
1
     the other group (group B) have entered the call (and vice versa). A process may return
\mathbf{2}
     from the call before all processes in its own group have entered the call.
3
4
           Broadcast
     6.4
5
6
7
8
     MPI_BCAST(buffer, count, datatype, root, comm)
9
       INOUT
                 buffer
                                             starting address of buffer (choice)
10
       IN
                                             number of entries in buffer (non-negative integer)
                 count
11
12
       IN
                 datatype
                                             datatype of buffer (handle)
13
       IN
                 root
                                             rank of broadcast root (integer)
14
                                             communicator (handle)
       IN
                 comm
15
16
17
     C binding
     int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root,
18
19
                    MPI_Comm comm)
20
     int MPI_Bcast_c(void *buffer, MPI_Count count, MPI_Datatype datatype,
21
                    int root, MPI_Comm comm)
22
23
     Fortran 2008 binding
24
     MPI_Bcast(buffer, count, datatype, root, comm, ierror)
25
          TYPE(*), DIMENSION(..) :: buffer
26
          INTEGER, INTENT(IN) :: count, root
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
          TYPE(MPI_Comm), INTENT(IN) :: comm
28
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
     MPI_Bcast(buffer, count, datatype, root, comm, ierror)
^{31}
          TYPE(*), DIMENSION(..) :: buffer
32
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
33
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
34
          INTEGER, INTENT(IN) :: root
35
          TYPE(MPI_Comm), INTENT(IN) :: comm
36
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
     Fortran binding
39
     MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
40
          <type> BUFFER(*)
41
          INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
42
         If comm is an intra-communicator, MPI_BCAST broadcasts a message from the process
43
     with rank root to all processes of the group, itself included. It is called by all members of
44
     the group using the same arguments for comm and root. On return, the content of root's
45
     buffer is copied to all other processes.
46
```

General, derived datatypes are allowed for datatype. The type signature of count, datatype on any process must be equal to the type signature of count, datatype at the root. This implies that the amount of data sent must be equal to the amount received, pairwise between each process and the root. MPI_BCAST and all other data-movement collective routines make this restriction. Distinct type maps between sender and receiver are still allowed.

The "in place" option is not meaningful here.

If comm is an inter-communicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is broadcast from the root to all processes in group B. The buffer arguments of the processes in group B must be consistent with the buffer argument of the root.

6.4.1 Example using MPI_BCAST

The examples in this section use intra-communicators.

Example 6.1 Broadcast 100 ints from process 0 to every process in the group.

```
MPI_Comm comm;
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.

 $\mathbf{2}$

 $\overline{7}$

CHAPTER 6. COLLECTIVE COMMUNICATION

6.5 Gather

3 4 MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm) 5IN sendbuf starting address of send buffer (choice) 6 7 IN sendcount number of elements in send buffer (non-negative 8 integer) 9 IN sendtype datatype of send buffer elements (handle) 10 OUT recvbuf address of receive buffer (choice, significant only at 11 root) 1213 IN number of elements for any single receive recvcount 14(non-negative integer, significant only at root) 15recvtype datatype of recy buffer elements (handle, significant IN 16 only at root) 17IN root rank of receiving process (integer) 18 19IN comm communicator (handle) 2021C binding 22 int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, 23 void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, 24MPI_Comm comm) 2526int MPI_Gather_c(const void *sendbuf, MPI_Count sendcount, MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount, 27MPI_Datatype recvtype, int root, MPI_Comm comm) 2829 Fortran 2008 binding 30 MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 31 root, comm, ierror) 32 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf 33 INTEGER, INTENT(IN) :: sendcount, recvcount, root 34 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 35TYPE(*), DIMENSION(..) :: recvbuf 36 TYPE(MPI_Comm), INTENT(IN) :: comm 37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 38 MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 39 40root, comm, ierror) 41 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf 42INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 43 TYPE(*), DIMENSION(...) :: recvbuf 44 INTEGER, INTENT(IN) :: root 45TYPE(MPI_Comm), INTENT(IN) :: comm 46 47 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 48

Fortran binding	1
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,	2
ROOT, COMM, IERROR)	3
<type> SENDBUF(*), RECVBUF(*)</type>	4
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR	5
	6
If comm is an intra-communicator, each process (root process included) sends the con-	7
tents of its send buffer to the root process. The root process receives the messages and stores	8
them in rank order. The outcome is <i>as if</i> each of the n processes in the group (including	9
the root process) had executed a call to	10
	11
MPI_Send(sendbuf, sendcount, sendtype, root ,),	12
and the root had executed n calls to	13
and the root had executed it cans to	14
MPI_Recv(recvbuf+i· recvcount· extent(recvtype), recvcount, recvtype, i,),	15
	16
where extent(recvtype) is the type extent obtained from a call to MPI_Type_get_extent.	17
An alternative description is that the n messages sent by the processes in the group	18
are concatenated in rank order, and the resulting message is received by the root as if by a	19
call to MPI_RECV(recvbuf, recvcount·n, recvtype,).	20
The receive buffer is ignored for all non-root processes.	20
	21
General, derived datatypes are allowed for both sendtype and recvtype. The type signa-	
ture of sendcount, sendtype on each process must be equal to the type signature of recvcount,	23
recvtype at the root. This implies that the amount of data sent must be equal to the amount	24
of data received, pairwise between each process and the root. Distinct type maps between	25
sender and receiver are still allowed.	26
All arguments to the function are significant on process root , while on other processes,	27
only arguments sendbuf, sendcount, sendtype, root, and comm are significant. The arguments	28
root and comm must have identical values on all processes.	29
The specification of counts and types should not cause any location on the root to be	30
written more than once. Such a call is erroneous.	31
Note that the recvcount argument at the root indicates the number of items it receives	32
from <i>each</i> process, not the total number of items it receives.	33
The "in place" option for intra-communicators is specified by passing MPI_IN_PLACE	34
as the value of sendbuf at the root. In such a case, sendcount and sendtype are ignored, and	35
the contribution of the root to the gathered vector is assumed to be already in the correct	36
place in the receive buffer.	37
If comm is an inter-communicator, then the call involves all processes in the inter-	38
communicator, but with one group (group A) defining the root process. All processes in	39
the other group (group B) pass the same value in argument root, which is the rank of the	40
root in group A. The root passes the value MPI_ROOT in root. All other processes in group	41
A pass the value MPI_PROC_NULL in root. Data is gathered from all processes in group B to the root. The good buffer arguments of the processes in group R must be consistent with	42
to the root. The send buffer arguments of the processes in group B must be consistent with	43
the receive buffer argument of the root.	44
	45
	46

12	MPI_GATH	IERV(sendbuf, sendcount, send comm)	dtype, recvbuf, recvcounts, displs, recvtype, root,		
3 4	IN	sendbuf	starting address of send buffer (choice)		
4 5 6	IN	sendcount	number of elements in send buffer (non-negative integer)		
7	IN	sendtype	datatype of send buffer elements (handle)		
8 9 10	OUT	recvbuf	address of receive buffer (choice, significant only at root)		
11 12 13	IN	recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process (significant only at root)		
14 15 16 17 18	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)		
19 20	IN	recvtype	datatype of recv buffer elements (handle, significant only at root)		
21	IN	root	rank of receiving process (integer)		
22	IN	comm	communicator (handle)		
23 24 25 26 27 28 29 30 31 32 33	int MPI_G	atherv(const void *sendb void *recvbuf, const MPI_Datatype recvtyp atherv_c(const void *send MPI_Datatype sendtyp const MPI_Count recv MPI_Datatype recvtyp	uf, int sendcount, MPI_Datatype sendtype, int recvcounts[], const int displs[], ee, int root, MPI_Comm comm) dbuf, MPI_Count sendcount, ee, void *recvbuf, counts[], const MPI_Aint displs[], ee, int root, MPI_Comm comm)		
34 35 36 37 38 39 40 41 42	<pre>MP1_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm, ierror) TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*), root TYPE(MP1_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(*), DIMENSION() :: recvbuf TYPE(MP1_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>				
43 44 45 46 47 48	TYPE(INTEG TYPE(<pre>recvtype, root, comm *), DIMENSION(), INTEN</pre>	T(IN) :: sendbuf INTENT(IN) :: sendcount, recvcounts(*)) :: sendtype, recvtype		

```
1
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
                                                                                               \mathbf{2}
    INTEGER, INTENT(IN) :: root
                                                                                               3
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                               4
                                                                                               5
Fortran binding
                                                                                               6
MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
                RECVTYPE, ROOT, COMM, IERROR)
     <type> SENDBUF(*), RECVBUF(*)
                                                                                               9
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
                                                                                               10
                 COMM, IERROR
                                                                                               11
    MPI_GATHERV extends the functionality of MPI_GATHER by allowing a varying count
                                                                                               12
                                                                                               13
of data from each process, since recvcounts is now an array. It also allows more flexibility
                                                                                               14
as to where the data is placed on the root, by providing the new argument, displs.
                                                                                               15
    If comm is an intra-communicator, the outcome is as if each process, including the
                                                                                               16
root process, sends a message to the root,
                                                                                               17
   MPI_Send(sendbuf, sendcount, sendtype, root, ...),
                                                                                               18
                                                                                               19
and the root executes n receives,
                                                                                               20
                                                                                               21
   MPI_Recv(recvbuf+displs[j] · extent(recvtype), recvcounts[j], recvtype, i, ...).
                                                                                               22
                                                                                               23
    The data received from process j is placed into recvbuf of the root process beginning at
                                                                                               ^{24}
offset displs[j] elements (in terms of the recvtype).
                                                                                               25
    The receive buffer is ignored for all non-root processes.
                                                                                               26
    The type signature implied by sendcount, sendtype on process i must be equal to the
                                                                                               27
type signature implied by recvcounts[i], recvtype at the root. This implies that the amount
                                                                                               28
of data sent must be equal to the amount of data received, pairwise between each process
                                                                                               29
and the root. Distinct type maps between sender and receiver are still allowed, as illustrated
                                                                                               30
in Example 6.6.
                                                                                               31
    All arguments to the function are significant on process root, while on other processes,
                                                                                               32
only arguments sendbuf, sendcount, sendtype, root, and comm are significant. The arguments
                                                                                               33
root and comm must have identical values on all processes.
                                                                                               34
    The specification of counts, types, and displacements should not cause any location on
                                                                                               35
the root to be written more than once. Such a call is erroneous.
                                                                                               36
    The "in place" option for intra-communicators is specified by passing MPI_IN_PLACE
                                                                                               37
as the value of sendbuf at the root. In such a case, sendcount and sendtype are ignored, and
                                                                                               38
the contribution of the root to the gathered vector is assumed to be already in the correct
                                                                                               39
place in the receive buffer.
                                                                                               40
    If comm is an inter-communicator, then the call involves all processes in the inter-
                                                                                               41
communicator, but with one group (group A) defining the root process. All processes in
                                                                                               42
the other group (group B) pass the same value in argument root, which is the rank of the
                                                                                               43
root in group A. The root passes the value MPI_ROOT in root. All other processes in group
                                                                                               44
```

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.

45

```
CHAPTER 6. COLLECTIVE COMMUNICATION
```

1 2 3 4 5 6 7 8 9	Figure 6.4: The root process gathers 100 ints from each process in the group.
10 11 12	6.5.1 Examples using MPI_GATHER, MPI_GATHERV
13	The examples in this section use intra-communicators.
14	
15 16	Example 6.2 Gather 100 ints from every process in group to root. See Figure 6.4.
17	MPI_Comm comm;
18	<pre>int gsize,sendarray[100];</pre>
19 20	<pre>int root, *rbuf;</pre>
20 21	 MPI_Comm_size(comm, &gsize);
22	rbuf = (int *)malloc(gsize*100*sizeof(int));
23	MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
24	
25	
26	Example 6.3 Previous example modified—only the root allocates memory for the receive buffer.
27 28	buildt.
28 29	MPI_Comm comm;
30	<pre>int gsize,sendarray[100];</pre>
31	<pre>int root, myrank, *rbuf;</pre>
32	
33	<pre>MPI_Comm_rank(comm, &myrank);</pre>
34	if (myrank == root) { MBL Comm size(comm & maize);
35	<pre>MPI_Comm_size(comm, &gsize); rbuf = (int *)malloc(gsize*100*sizeof(int));</pre>
36	}
37	MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
38 39	
40	
41	Example 6.4 Do the same as the previous example, but use a derived datatype. Note
42	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.
43	pairwise between the root and each process in the gather.
44	
45	
46	
47	
48	

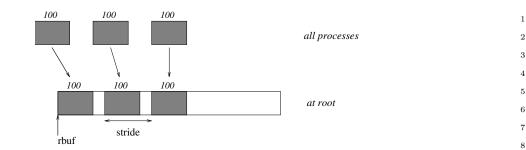


Figure 6.5: The root process gathers 100 ints from each process in the group, each set is placed stride ints apart.

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, *rbuf;
MPI_Datatype rtype;
...
MPI_Comm_size(comm, &gsize);
MPI_Type_contiguous(100, MPI_INT, &rtype);
MPI_Type_commit(&rtype);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather(sendarray, 100, MPI_INT, rbuf, 1, rtype, root, comm);
```

Example 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 $stride \ge 100$. See Figure 6.5.

```
27
MPI_Comm comm;
                                                                                    28
int gsize, sendarray [100];
                                                                                    29
int root, *rbuf, stride;
                                                                                    30
int *displs,i,*rcounts;
                                                                                    31
                                                                                    32
                                                                                    33
                                                                                    34
MPI_Comm_size(comm, &gsize);
                                                                                    35
rbuf = (int *)malloc(gsize*stride*sizeof(int));
                                                                                    36
displs = (int *)malloc(gsize*sizeof(int));
                                                                                    37
rcounts = (int *)malloc(gsize*sizeof(int));
                                                                                    38
for (i=0; i<gsize; ++i) {</pre>
                                                                                    39
    displs[i] = i*stride;
                                                                                    40
    rcounts[i] = 100;
                                                                                    41
}
                                                                                    42
MPI_Gatherv(sendarray, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT,
                                                                                    43
             root, comm);
                                                                                    44
                                                                                    45
Note that the program is erroneous if stride < 100.
                                                                                    46
```

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.

Unofficial Draft for Comment Only

9

10

11 12

13

14

15

16

17

18

19

20

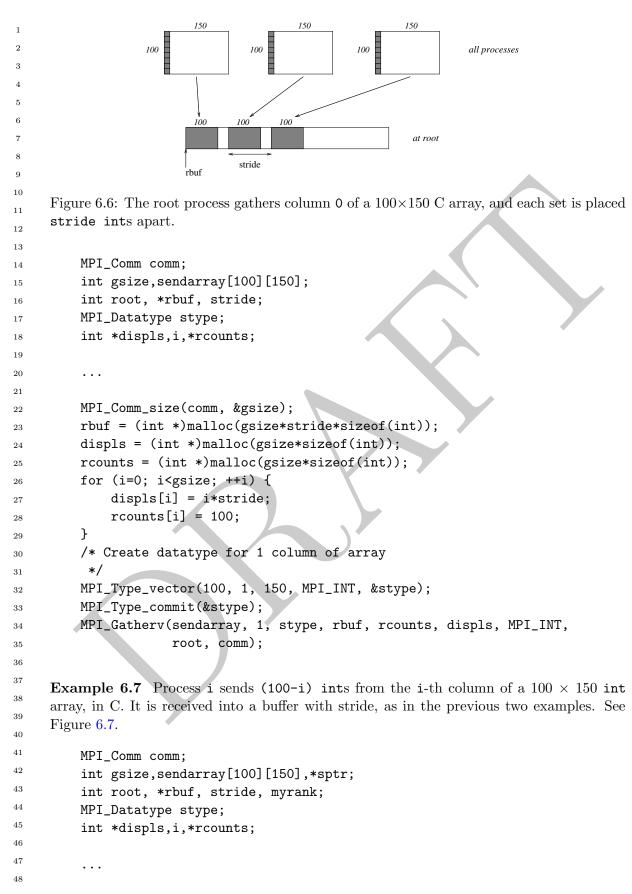
21

22 23

 $\frac{24}{25}$

26

47



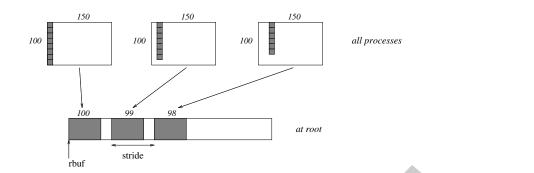


Figure 6.7: The root process gathers 100-i ints from column i of a 100×150 C array, and each set is placed stride ints apart.

```
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {</pre>
    displs[i] = i*stride;
                            /* note change from previous example */
    rcounts[i] = 100-i;
}
/* 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.

```
38
MPI_Comm comm;
                                                                                  39
int gsize, sendarray[100][150], *sptr;
int root, *rbuf, stride, myrank;
                                                                                   40
                                                                                   41
MPI_Datatype stype;
                                                                                   42
int *displs, i, *rcounts;
                                                                                   43
                                                                                   44
. . .
                                                                                   45
                                                                                   46
MPI_Comm_size(comm, &gsize);
                                                                                   47
MPI_Comm_rank(comm, &myrank);
                                                                                   48
rbuf = (int *)malloc(gsize*stride*sizeof(int));
```

1 2

10

11

12 13

14

15

16

17

18

19

20

21

22

23

 24

25

26

27

28

29

30

31 32

33 34

35

36

```
1
         displs = (int *)malloc(gsize*sizeof(int));
\mathbf{2}
         rcounts = (int *)malloc(gsize*sizeof(int));
3
         for (i=0; i<gsize; ++i) {</pre>
4
              displs[i] = i*stride;
5
              rcounts[i] = 100-i;
6
         }
7
          /* Create datatype for one int, with extent of entire row
8
           */
9
         MPI_Type_create_resized(MPI_INT, 0, 150*sizeof(int), &stype);
10
         MPI_Type_commit(&stype);
11
         sptr = &sendarray[0][myrank];
12
         MPI_Gatherv(sptr, 100-myrank, stype, rbuf, rcounts, displs, MPI_INT,
13
                       root, comm);
14
15
     Example 6.9 Same as Example 6.7 at sending side, but at receiving side we make the
16
     stride between received blocks vary from block to block. See Figure 6.8.
17
18
         MPI_Comm comm;
19
         int gsize, sendarray[100][150], *sptr;
20
         int root, *rbuf, *stride, myrank, bufsize;
21
         MPI_Datatype stype;
22
         int *displs,i,*rcounts,offset;
23
24
          . . .
25
26
         MPI_Comm_size(comm, &gsize);
27
         MPI_Comm_rank(comm, &myrank);
28
29
         stride = (int *)malloc(gsize*sizeof(int));
30
31
          /* stride[i] for i = 0 to gsize-1 is set somehow
32
           */
33
34
         /* set up displs and rcounts vectors first
35
           */
36
         displs = (int *)malloc(gsize*sizeof(int));
37
         rcounts = (int *)malloc(gsize*sizeof(int));
38
         offset = 0;
39
         for (i=0; i<gsize; ++i) {</pre>
40
              displs[i] = offset;
41
              offset += stride[i];
42
              rcounts[i] = 100-i;
43
         }
44
         /* the required buffer size for rbuf is now easily obtained
45
           */
46
         bufsize = displs[gsize-1]+rcounts[gsize-1];
47
         rbuf = (int *)malloc(bufsize*sizeof(int));
48
```

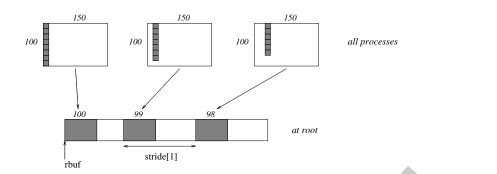


Figure 6.8: The root process gathers 100-i ints from column i of a 100×150 C array, and each set is placed stride[i] ints apart (a varying stride).

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.

```
MPI_Comm comm;
                                                                                  27
int gsize,sendarray[100][150],*sptr;
                                                                                  28
                                                                                  29
int root, *rbuf, myrank;
                                                                                  30
MPI_Datatype stype;
                                                                                  31
int *displs,i,*rcounts,num;
                                                                                  32
                                                                                  33
 . .
                                                                                  34
MPI_Comm_size(comm, &gsize);
                                                                                  35
                                                                                  36
MPI_Comm_rank(comm, &myrank);
                                                                                  37
                                                                                  38
/* First, gather nums to root
                                                                                  39
 */
                                                                                  40
rcounts = (int *)malloc(gsize*sizeof(int));
                                                                                  41
MPI_Gather(&num, 1, MPI_INT, rcounts, 1, MPI_INT, root, comm);
                                                                                 42
/* root now has correct roounts, using these we set displs[] so
 * that data is placed contiguously (or concatenated) at receive end
                                                                                  43
                                                                                  44
 */
displs = (int *)malloc(gsize*sizeof(int));
                                                                                  45
                                                                                  46
displs[0] = 0;
                                                                                  47
for (i=1; i<gsize; ++i) {</pre>
                                                                                  48
    displs[i] = displs[i-1]+rcounts[i-1];
```

Unofficial Draft for Comment Only

 24

```
1
          }
2
          /* And, create receive buffer
3
           */
4
          rbuf = (int *)malloc(gsize*(displs[gsize-1]+rcounts[gsize-1])
5
                                                                           *sizeof(int));
6
          /* Create datatype for one int, with extent of entire row
7
           */
8
          MPI_Type_create_resized(MPI_INT, 0, 150*sizeof(int), &stype);
9
          MPI_Type_commit(&stype);
10
          sptr = &sendarray[0][myrank];
11
          MPI_Gatherv(sptr, num, stype, rbuf, rcounts, displs, MPI_INT,
12
                        root, comm);
13
14
          Scatter
     6.6
15
16
17
     MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
18
19
       IN
                 sendbuf
                                             address of send buffer (choice, significant only at
20
                                             root)
21
       IN
                 sendcount
                                             number of elements sent to each process
22
                                             (non-negative integer, significant only at root)
23
24
       IN
                 sendtype
                                             datatype of send buffer elements (handle, significant
25
                                             only at root)
26
       OUT
                 recvbuf
                                             address of receive buffer (choice)
27
                                             number of elements in receive buffer (non-negative
       IN
                 recvcount
28
                                             integer)
29
30
       IN
                                             datatype of receive buffer elements (handle)
                 recvtype
31
       IN
                 root
                                             rank of sending process (integer)
32
       IN
                 comm
                                             communicator (handle)
33
34
     C binding
35
     int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
36
37
                    void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
                    MPI_Comm comm)
38
39
     int MPI_Scatter_c(const void *sendbuf, MPI_Count sendcount,
40
                    MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
41
                    MPI_Datatype recvtype, int root, MPI_Comm comm)
42
     Fortran 2008 binding
43
44
     MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
45
                    root, comm, ierror)
46
          TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
47
          INTEGER, INTENT(IN) :: sendcount, recvcount, root
48
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
```

```
TYPE(*), DIMENSION(...) :: recvbuf
                                                                                            1
                                                                                            2
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
               root, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(*), DIMENSION(...) :: recvbuf
                                                                                            10
    INTEGER, INTENT(IN) :: root
                                                                                            11
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                            12
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                            13
                                                                                            14
Fortran binding
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
                                                                                            15
                                                                                            16
               ROOT, COMM, IERROR)
                                                                                            17
     <type> SENDBUF(*), RECVBUF(*)
     INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
                                                                                            18
                                                                                            19
    MPI_SCATTER is the inverse operation to MPI_GATHER.
                                                                                            20
    If comm is an intra-communicator, the outcome is as if the root executed n send
                                                                                           21
operations,
                                                                                            22
                                                                                            23
   MPI_Send(sendbuf+i· sendcount· extent(sendtype), sendcount, sendtype, i,...),
                                                                                            24
                                                                                            25
and each process executed a receive,
                                                                                            26
                                                                                            27
   MPI_Recv(recvbuf, recvcount, recvtype, i,...).
                                                                                            28
    An alternative description is that the root sends a message with MPI_Send(sendbuf,
                                                                                            29
sendcount \cdot n, sendtype, ...). This message is split into n equal segments, the i-th segment is
                                                                                           30
sent to the i-th process in the group, and each process receives this message as above.
                                                                                            31
    The send buffer is ignored for all non-root processes.
                                                                                            32
    The type signature associated with sendcount, sendtype at the root must be equal to
                                                                                           33
the type signature associated with recvcount, recvtype at all processes (however, the type
                                                                                           34
maps may be different). This implies that the amount of data sent must be equal to the
                                                                                           35
amount of data received, pairwise between each process and the root. Distinct type maps
                                                                                           36
between sender and receiver are still allowed.
                                                                                           37
    All arguments to the function are significant on process root, while on other processes,
                                                                                           38
only arguments recvouf, recvcount, recvtype, root, and comm are significant. The arguments
                                                                                            39
root and comm must have identical values on all processes.
                                                                                            40
                                                                                           41
    The specification of counts and types should not cause any location on the root to be
read more than once.
                                                                                            42
                                                                                            43
     Rationale.
                   Though not needed, the last restriction is imposed so as to achieve
                                                                                            44
     symmetry with MPI_GATHER, where the corresponding restriction (a multiple-write
                                                                                            45
     restriction) is necessary. (End of rationale.)
                                                                                            46
                                                                                            47
    The "in place" option for intra-communicators is specified by passing MPI_IN_PLACE
                                                                                            48
as the value of recvbuf at the root. In such a case, recvcount and recvtype are ignored, and
```

¹ root "sends" no data to itself. The scattered vector is still assumed to contain n segments, ² where n is the group size; the *root*-th segment, which root should "send to itself," is not ³ moved.

⁴ If comm is an inter-communicator, then the call involves all processes in the inter-⁵ communicator, but with one group (group A) defining the root process. All processes in ⁶ the other group (group B) pass the same value in argument root, which is the rank of the ⁷ root in group A. The root passes the value MPI_ROOT in root. All other processes in group ⁸ A pass the value MPI_PROC_NULL in root. Data is scattered from the root to all processes ⁹ in group B. The receive buffer arguments of the processes in group B must be consistent ¹⁰ with the send buffer argument of the root.

11 12

MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)

15 16	IN	sendbuf	address of send buffer (choice, significant only at root)		
17 18 19	IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank (significant only at root)		
20 21 22 23 24	IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to sendbuf) from which to take the outgoing data to process i (significant only at root)		
25 26	IN	sendtype	datatype of send buffer elements (handle, significant only at root)		
27 28	OUT	recvbuf	address of receive buffer (choice)		
29 30	IN	recvcount	number of elements in receive buffer (non-negative integer)		
31	IN	recvtype	datatype of receive buffer elements (handle)		
32 33	IN	root	rank of sending process (integer)		
34 35	IN	comm	communicator (handle)		
36	C binding	g			
37	int MPI_S	catterv(const void *send	ouf, const int sendcounts[],		
38 39	conto int albpib[], in i_batatype bonatype, voia icovbai,				
40 41	<pre>int MPI_Scatterv_c(const void *sendbuf, const MPI_Count sendcounts[],</pre>				
41	<pre>const MPI_Aint displs[], MPI_Datatype sendtype, void *recvbuf,</pre>				
43			MPI_Datatype recvtype, int root,		
44		MPI_Comm comm)			
45		2008 binding			
46 47	MP1_Scatt	erv(sendbuf, sendcounts, recvtype, root, comm	displs, sendtype, recvbuf, recvcount,		
48	TYPE(*), DIMENSION(), INTENT			

```
INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                          2
    TYPE(*), DIMENSION(..) :: recvbuf
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
               recvtype, root, comm, ierror)
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*), recvcount
                                                                                          10
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
                                                                                          11
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                          12
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                          13
    INTEGER, INTENT(IN) :: root
                                                                                          14
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                          15
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                          16
                                                                                          17
Fortran binding
                                                                                          18
MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
                                                                                          19
               RECVTYPE, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
                                                                                          20
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
                                                                                          21
                                                                                          22
                COMM, IERROR
                                                                                          23
    MPI_SCATTERV is the inverse operation to MPI_GATHERV.
                                                                                          24
    MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying
                                                                                          25
count of data to be sent to each process, since sendcounts is now an array. It also allows
                                                                                          26
more flexibility as to where the data is taken from on the root, by providing an additional
                                                                                          27
argument, displs.
                                                                                          28
    If comm is an intra-communicator, the outcome is as if the root executed n send oper-
                                                                                          29
ations,
                                                                                          30
                                                                                          31
   MPI_Send(sendbuf+displs[i] extent(sendtype), sendcounts[i], sendtype, i,...),
                                                                                          32
                                                                                          33
and each process executed a receive,
                                                                                          34
                                                                                          35
   MPI_Recv(recvbuf, recvcount, recvtype, i,...).
                                                                                          36
    The send buffer is ignored for all non-root processes.
                                                                                          37
    The type signature implied by sendcount[i], sendtype at the root must be equal to the
                                                                                          38
type signature implied by recvcount, recvtype at process i (however, the type maps may be
                                                                                          39
different). This implies that the amount of data sent must be equal to the amount of data
                                                                                          40
                                                                                          41
received, pairwise between each process and the root. Distinct type maps between sender
and receiver are still allowed.
                                                                                          42
    All arguments to the function are significant on process root, while on other processes,
                                                                                          43
only arguments recvbuf, recvcount, recvtype, root, and comm are significant. The arguments
                                                                                          44
root and comm must have identical values on all processes.
                                                                                          45
    The specification of counts, types, and displacements should not cause any location on
                                                                                          46
the root to be read more than once.
                                                                                          47
                                                                                          48
```

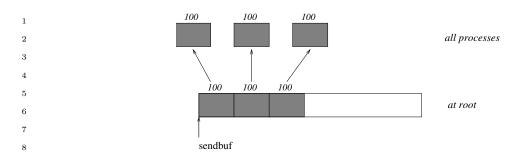


Figure 6.9: The root process scatters sets of 100 ints to each process in the group.

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.

¹⁷ If comm is an inter-communicator, then the call involves all processes in the inter-¹⁸ communicator, but with one group (group A) defining the root process. All processes in ¹⁹ the other group (group B) pass the same value in argument root, which is the rank of the ²⁰ root in group A. The root passes the value MPI_ROOT in root. All other processes in group ²¹ A pass the value MPI_PROC_NULL in root. Data is scattered from the root to all processes ²² in group B. The receive buffer arguments of the processes in group B must be consistent ²³ with the send buffer argument of the root.

 $\frac{24}{25}$

26

31

32

33

34 35

36

37 38

39 40

9

10 11

12

13

14

15 16

6.6.1 Examples using MPI_SCATTER, MPI_SCATTERV

 $_{27}$ The examples in this section use intra-communicators.

Example 6.11 The reverse of Example 6.2. Scatter sets of 100 ints from the root to each process in the group. See Figure 6.9.

```
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100];
...
MPI_Comm_size(comm, &gsize);
sendbuf = (int *)malloc(gsize*100*sizeof(int));
...
MPI_Scatter(sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

⁴¹ **Example 6.12** The reverse of Example 6.5. The root process scatters sets of 100 ints to ⁴² the other processes, but the sets of 100 are *stride ints* apart in the sending buffer. Requires ⁴³ use of MPI_SCATTERV. Assume *stride* \geq 100. See Figure 6.10.

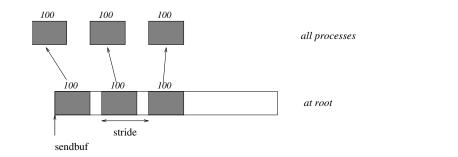


Figure 6.10: The root process scatters sets of 100 ints, moving by stride ints from send to send in the scatter.

```
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100], i, *displs, *scounts;
. . .
MPI_Comm_size(comm, &gsize);
sendbuf = (int *)malloc(gsize*stride*sizeof(int));
. . .
displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {</pre>
    displs[i] = i*stride;
    scounts[i] = 100;
}
MPI_Scatterv(sendbuf, scounts, displs, MPI_INT, rbuf, 100, MPI_INT,
             root, comm);
```

Example 6.13 The reverse of Example 6.9. We have a varying stride between blocks at sending (root) side, at the receiving side we receive into the *i*-th column of a 100×150 C array. See Figure 6.11.

```
MPI_Comm comm;
                                                                              35
int gsize,recvarray[100][150],*rptr;
int root, *sendbuf, myrank, *stride;
                                                                              37
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
 */
```

1

2

5

6

a

10

11 12

13

14

151617

18

19

20

21

22

23

 24

25

26

27

28

29 30 31

32

33

34

36

38

39

40

41

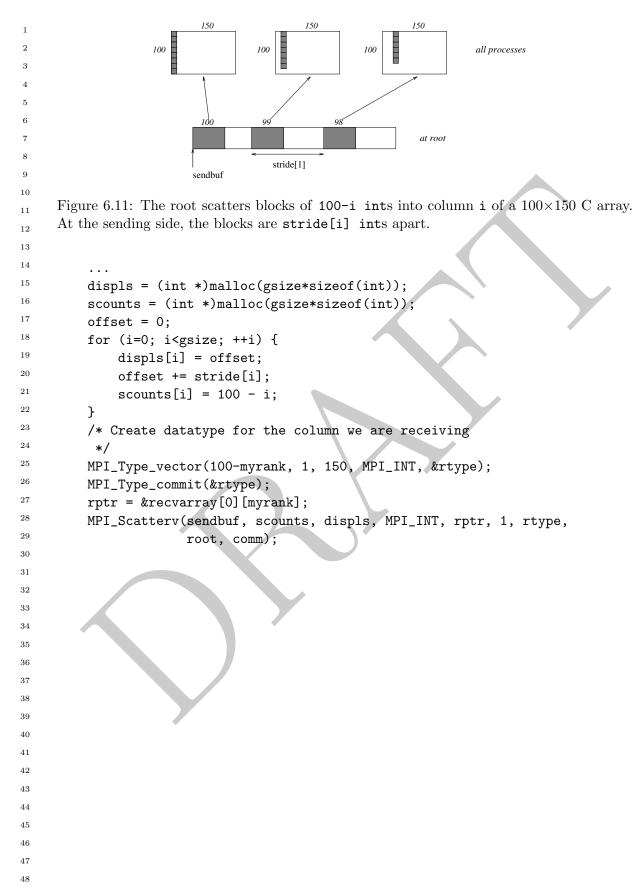
42

43 44

45

46

47



6.7 Gather-to-all

MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

IN	sendbuf	starting address of send buffer (choice)	6
IN	sendcount	number of elements in send buffer (non-negative	7
		integer)	8
IN	sendtype	datatype of send buffer elements (handle)	9 10
OUT	recvbuf	address of receive buffer (choice)	11
IN	recvcount	number of elements received from any process	12
		(non-negative integer)	13
IN	recvtype	datatype of receive buffer elements (handle)	14 15
IN	comm	communicator (handle)	15 16
			17
C bindir	ıg		18

C binding

0	
<pre>int MPI_Allgather(const void *sendbuf, int sendcount,</pre>	:
MPI_Datatype sendtype, void *recvbuf, int recvcount;	,
MPI_Datatype recvtype, MPI_Comm comm)	:
<pre>int MPI_Allgather_c(const void *sendbuf, MPI_Count sendcount,</pre>	-
MPI Datatype sendtype, void *recybuf, MPI Count recy	vcount

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) 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, RECVCOUNT, RECVTYPE, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR

1 2 3 4 5 6 7 8	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
9 10	<pre>MPI_Gather(sendbuf,sendcount,sendtype,recvbuf,recvcount, recvtype,root,comm)</pre>
11 12 13 14 15 16 17 18 19 20 21 22	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.
23 24 25 26 27 28 29 30 31	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. (<i>End of advice to users.</i>)
32 33 34 35 36 37 38 39 40 41 42	
43 44 45 46 47 48	

MPI_ALL	GATHERV(sendbuf, sendco comm)	unt, sendtype, recvbuf, recvcounts, displs, recvtype,	1 2	
IN	sendbuf	starting address of send buffer (choice)	$\frac{3}{4}$	
IN	sendcount	number of elements in send buffer (non-negative integer)	4 5 6	
IN	sendtype	datatype of send buffer elements (handle)	7	
OUT	recvbuf	address of receive buffer (choice)	8 9	
IN	recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process	9 10 11 12	
IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i	13 14 15	
IN	recvtype	datatype of receive buffer elements (handle)	16 17	
IN	comm	communicator (handle)	18	
			19	
C bindin	•	*sendbuf, int sendcount,	20 21	
IIIC MFI_	•	<pre>#sendbul, int sendcount, dtype, void *recvbuf, const int recvcounts[],</pre>	21	
		[], MPI_Datatype recvtype, MPI_Comm comm)	23	
<pre>int MPI_Allgatherv_c(const void *sendbuf, MPI_Count sendcount,</pre>				
MPI_Datatype sendtype, void *recvbuf,				
<pre>const MPI_Count recvcounts[], const MPI_Aint displs[],</pre>				
MPI_Datatype recvtype, MPI_Comm comm)				
Fortran	Fortran 2008 binding			
MPI_Allg		ount, sendtype, recvbuf, recvcounts, displs,	30	
TVDE	recvtype, comm, f		31 32	
	C(*), DIMENSION(), IN CGER INTENT(IN) ·· sen	dcount, recvcounts(*), displs(*)	33	
		C(IN) :: sendtype, recvtype	34	
	S(*), DIMENSION() ::		35	
	C(MPI_Comm), INTENT(IN)		36	
INTE	CGER, OPTIONAL, INTENT(OUT) :: ierror	37	
MPI_Allg	gatherv(sendbuf, sendco	ount, sendtype, recvbuf, recvcounts, displs,	38 39	
	recvtype, comm, i		40	
	E(*), DIMENSION(), IN		41	
		<pre>ID), INTENT(IN) :: sendcount, recvcounts(*)</pre>	42	
	C(MPI_Datatype), INTENI C(*), DIMENSION() ::	<pre>C(IN) :: sendtype, recvtype recvbuf</pre>	43	
		<pre>Intervolut IND), INTENT(IN) :: displs(*)</pre>	44	
	C(MPI_Comm), INTENT(IN)	-	45	
	GER, OPTIONAL, INTENT(46 47	

1	Fortran binding
2	MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
$\frac{3}{4}$	RECVTYPE, COMM, IERROR)
5	<type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,</type>
6	INTEGER SENDCOUNT, SENDTIFE, RECOCOUNTS(*), DISPLS(*), RECVITE, COMM, IERROR
7	MPI_ALLGATHERV can be thought of as MPI_GATHERV, but where all processes re-
8 9	ceive the result, instead of just the root. The block of data sent from the j-th process is
9 10	received by every process and placed in the j-th block of the buffer recvbuf. These blocks
11	need not all be the same size.
12	The type signature associated with sendcount, sendtype, at process j must be equal to
13	the type signature associated with recvcounts[j], recvtype at any other process.
14	If comm is an intra-communicator, the outcome is as if all processes executed calls to
15 16	MPI_Gatherv(sendbuf,sendcount,sendtype,recvbuf,recvcounts,displs,
17	recvtype,root,comm),
18	for root = 0,, n-1. The rules for correct usage of MPI_ALLGATHERV are easily
19	found from the corresponding rules for MPI_GATHERV.
20	The "in place" option for intra-communicators is specified by passing the value
21	MPI_IN_PLACE to the argument sendbuf at all processes. In such a case, sendcount and sendtype are ignored, and the input data of each process is assumed to be in the area where
22 23	that process would receive its own contribution to the receive buffer.
24	If comm is an inter-communicator, then each process of one group (group A) contributes
25	sendcount data items; these data are concatenated and the result is stored at each process
26	in the other group (group B). Conversely the concatenation of the contributions of the
27	processes in group B is stored at each process in group A. The send buffer arguments in
28	group A must be consistent with the receive buffer arguments in group B, and vice versa.
29	
30 31	6.7.1 Example using MPI_ALLGATHER
32	The example in this section uses intra-communicators.
33	Example 6.14 The all-gather version of Example 6.2. Using MPI_ALLGATHER, we will
34	gather 100 ints from every process in the group to every process.
35 36	
37	MPI_Comm comm;
38	<pre>int gsize,sendarray[100]; int tabué</pre>
39	<pre>int *rbuf;</pre>
40	 MPI_Comm_size(comm, &gsize);
41	rbuf = (int *)malloc(gsize*100*sizeof(int));
42	MPI_Allgather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, comm);
43 44	
45	After the call, every process has the group-wide concatenation of the sets of data.
46	
47	
48	

6.8 All-to-All Scatter/Gather

2 MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm) 5 sendbuf IN starting address of send buffer (choice) 6 7 IN sendcount number of elements sent to each process (non-negative integer) IN sendtype datatype of send buffer elements (handle) 10 OUT recvbuf address of receive buffer (choice) 11 IN number of elements received from any process 12recvcount 13 (non-negative integer) 14datatype of receive buffer elements (handle) IN recvtype 15IN communicator (handle) comm 1617C binding 18 int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype, 19 void *recvbuf, int recvcount, MPI_Datatype recvtype, 20MPI_Comm comm) 2122 int MPI_Alltoall_c(const void *sendbuf, MPI_Count sendcount, 23MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount, 24MPI_Datatype recvtype, MPI_Comm comm) 2526Fortran 2008 binding MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 27comm, ierror) 28TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf 29 INTEGER, INTENT(IN) :: sendcount, recvcount 30 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 31TYPE(*), DIMENSION(..) :: recvbuf 32 TYPE(MPI_Comm), INTENT(IN) :: comm 33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 34 35MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 36 comm, ierror) 37 TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf 38 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount 39 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 40 TYPE(*), DIMENSION(..) :: recvbuf 41 TYPE(MPI_Comm), INTENT(IN) :: comm 42INTEGER, OPTIONAL, INTENT(OUT) :: ierror 43 44 Fortran binding MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 45COMM, IERROR) 4647<type> SENDBUF(*), RECVBUF(*) 48 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR

1	$MPI_ALLTOALL$ is an extension of $MPI_ALLGATHER$ to the case where each process
2	sends distinct data to each of the receivers. The j-th block sent from process i is received
3	by process j and is placed in the i-th block of recvbuf.
4	The type signature associated with sendcount, sendtype, at a process must be equal to
5	the type signature associated with recvcount, recvtype at any other process. This implies
6	that the amount of data sent must be equal to the amount of data received, pairwise between
7	every pair of processes. As usual, however, the type maps may be different.
8	If comm is an intra-communicator, the outcome is as if each process executed a send
9	to each process (itself included) with a call to,
10 11	MPI_Send(sendbuf+i· sendcount· extent(sendtype),sendcount,sendtype,i,),
11	MF1_Send(sendbur+1) sendcount: extent(sendtype),sendcount,sendtype,1,),
13	and a receive from every other process with a call to,
14	
15	$MPI_Recv(recvbuf+i\cdotrecvcount\cdotextent(recvtype),recvcount,recvtype,i,).$
16 17	All arguments on all processes are significant. The argument comm must have identical
18	values on all processes.
19	The "in place" option for intra-communicators is specified by passing MPI_IN_PLACE to
20	the argument sendbuf at all processes. In such a case, sendcount and sendtype are ignored.
21	The data to be sent is taken from the recvbuf and replaced by the received data. Data sent
22	and received must have the same type map as specified by recvcount and recvtype .
23	Bationale For large MDL ALLTOALL instances allocating both and and receive
24	<i>Rationale.</i> For large MPI_ALLTOALL instances, allocating both send and receive buffers may consume too much memory. The "in place" option effectively halves the
25	application memory consumption and is useful in situations where the data to be sent
26	will not be used by the sending process after the MPI_ALLTOALL exchange (e.g., in
27	parallel Fast Fourier Transforms). (End of rationale.)
28	parallel Fast Fourier Transforms). (End of Tationale.)
29	Advice to implementors. Users may opt to use the "in place" option in order to
30	conserve memory. Quality MPI implementations should thus strive to minimize system
31	buffering. (End of advice to implementors.)
32	
33	If comm is an inter-communicator, then the outcome is as if each process in group A
34	sends a message to each process in group B, and vice versa. The j-th send buffer of process
35	i in group A should be consistent with the i-th receive buffer of process j in group B, and
36	vice versa.
37	Advice to ware. When a complete exchange is executed on an intercommunication
$\frac{38}{39}$	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
40	in group B need not equal the number of items sent in the reverse direction. In
41	particular, one can have unidirectional communication by specifying sendcount $= 0$ in
42	the reverse direction. (<i>End of advice to users.</i>)
43	
44	
45	
46	
47	
48	

IN	sendbuf	starting address of send buffer (choice)
N	sendcounts	non-negative integer array (of length group size)
		specifying the number of elements to send to each
		rank
IN	sdispls	integer array (of length group size). Entry j specifies
		the displacement (relative to $sendbuf)$ from which to
		take the outgoing data destined for process j
IN	sendtype	datatype of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcounts	non-negative integer array (of length group size)
		specifying the number of elements that can be
		received from each rank
IN	rdispls	integer array (of length group size). Entry i specifies
		the displacement (relative to recvbuf) at which to
		place the incoming data from process i
IN	recvtype	datatype of receive buffer elements (handle)
IN	comm	communicator (handle)
bindi	ng	
nt MPI		<pre>d *sendbuf, const int sendcounts[],</pre>
		pls[], MPI_Datatype sendtype, void *recvbuf,
		counts[], const int rdispls[], ecvtype, MPI_Comm comm)
nt MPI		oid *sendbuf, const MPI_Count sendcounts[],
		sdispls[], MPI_Datatype sendtype,
		<pre>const MPI_Count recvcounts[], rdicpls[] MPI_Datatupe recutupe</pre>
	CONST MP1_AINT MP1_Comm comm)	rdispls[], MPI_Datatype recvtype,
	2008 binding	
	toallv(sendbuf, send	counts, sdispls, sendtype, recvbuf, recvcounts,
PI_All	toallv(sendbuf, send rdispls, recvt	ype, comm, ierror)
PI_All TYP	toallv(sendbuf, send rdispls, recvt E(*), DIMENSION(),	ype, comm, ierror) INTENT(IN) :: sendbuf
PI_All TYP	toallv(sendbuf, send rdispls, recvt E(*), DIMENSION(),	ype, comm, ierror)
PI_All TYP INT	toallv(sendbuf, send rdispls, recvt E(*), DIMENSION(), EGER, INTENT(IN) :: rdispls(*)	ype, comm, ierror) INTENT(IN) :: sendbuf
PI_All TYP INT TYP	toallv(sendbuf, send rdispls, recvt E(*), DIMENSION(), EGER, INTENT(IN) :: rdispls(*)	<pre>ype, comm, ierror) INTENT(IN) :: sendbuf sendcounts(*), sdispls(*), recvcounts(*), ENT(IN) :: sendtype, recvtype</pre>
PI_All TYP INT TYP TYP TYP	<pre>toallv(sendbuf, send rdispls, recvt E(*), DIMENSION(), EGER, INTENT(IN) :: rdispls(*) E(MPI_Datatype), INT E(*), DIMENSION() E(MPI_Comm), INTENT(</pre>	<pre>ype, comm, ierror) INTENT(IN) :: sendbuf sendcounts(*), sdispls(*), recvcounts(*), ENT(IN) :: sendtype, recvtype :: recvbuf IN) :: comm</pre>
PI_All TYP INT TYP TYP TYP	<pre>toallv(sendbuf, send rdispls, recvt E(*), DIMENSION(), EGER, INTENT(IN) :: rdispls(*) E(MPI_Datatype), INT E(*), DIMENSION()</pre>	<pre>ype, comm, ierror) INTENT(IN) :: sendbuf sendcounts(*), sdispls(*), recvcounts(*), ENT(IN) :: sendtype, recvtype :: recvbuf IN) :: comm</pre>
PI_All TYP INT TYP TYP INT	<pre>toallv(sendbuf, send rdispls, recvt E(*), DIMENSION(), EGER, INTENT(IN) :: rdispls(*) E(MPI_Datatype), INT E(*), DIMENSION() E(MPI_Comm), INTENT(EGER, OPTIONAL, INTE</pre>	<pre>ype, comm, ierror) INTENT(IN) :: sendbuf sendcounts(*), sdispls(*), recvcounts(*), ENT(IN) :: sendtype, recvtype :: recvbuf IN) :: comm</pre>
PI_All TYP INT TYP TYP INT	<pre>toallv(sendbuf, send rdispls, recvt E(*), DIMENSION(), EGER, INTENT(IN) :: rdispls(*) E(MPI_Datatype), INT E(*), DIMENSION() E(MPI_Comm), INTENT(EGER, OPTIONAL, INTE toallv(sendbuf, send</pre>	<pre>ype, comm, ierror) INTENT(IN) :: sendbuf sendcounts(*), sdispls(*), recvcounts(*), ENT(IN) :: sendtype, recvtype :: recvbuf IN) :: comm NT(OUT) :: ierror</pre>

TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf

```
1
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*),
2
                      recvcounts(*)
3
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
4
5
          TYPE(*), DIMENSION(..) :: recvbuf
6
          TYPE(MPI_Comm), INTENT(IN) :: comm
7
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
     Fortran binding
9
     MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,
10
                     RDISPLS, RECVTYPE, COMM, IERROR)
11
          <type> SENDBUF(*), RECVBUF(*)
12
          INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
13
                      RECVTYPE, COMM, IERROR
14
15
          MPI_ALLTOALLV adds flexibility to MPI_ALLTOALL in that the location of data for
16
      the send is specified by sdispls and the location of the placement of the data on the receive
17
     side is specified by rdispls.
18
          If comm is an intra-communicator, then the j-th block sent from process i is received
19
      by process j and is placed in the i-th block of recvbuf. These blocks need not all have the
20
     same size.
21
          The type signature associated with sendcounts[j], sendtype at process i must be equal
22
      to the type signature associated with recvcounts[i], recvtype at process j. This implies that
23
      the amount of data sent must be equal to the amount of data received, pairwise between
^{24}
      every pair of processes. Distinct type maps between sender and receiver are still allowed.
25
          The outcome is as if each process sent a message to every other process with,
26
         MPI_Send(sendbuf+sdispls[i] · extent(sendtype),sendcounts[i],sendtype,i,...),
27
28
      and received a message from every other process with a call to
29
30
         MPI_Recv(recvbuf+rdispls[i] extent(recvtype),recvcounts[i],recvtype,i,...).
^{31}
32
          All arguments on all processes are significant. The argument comm must have identical
33
     values on all processes.
34
          The "in place" option for intra-communicators is specified by passing MPI_IN_PLACE to
35
      the argument sendbuf at all processes. In such a case, sendcounts, sdispls and sendtype are
36
      ignored. The data to be sent is taken from the recvbuf and replaced by the received data.
37
      Data sent and received must have the same type map as specified by the recvcounts array
38
      and the recvtype, and is taken from the locations of the receive buffer specified by rdispls.
39
40
                                Specifying the "in place" option (which must be given on all
           Advice to users.
41
           processes) implies that the same amount and type of data is sent and received between
42
           any two processes in the group of the communicator. Different pairs of processes can
43
           exchange different amounts of data. Users must ensure that recvcounts[j] and recvtype
44
           on process i match recvcounts[i] and recvtype on process j. This symmetric exchange
45
           can be useful in applications where the data to be sent will not be used by the sending
46
           process after the MPI_ALLTOALLV exchange. (End of advice to users.)
47
```

If comm is an inter-communicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The j-th send buffer of process i in group A should be consistent with the i-th receive buffer of process j in group B, and vice versa.

Rationale. The definitions of MPI_ALLTOALL and MPI_ALLTOALLV give as much flexibility as one would achieve by specifying n independent, point-to-point communications, with two exceptions: all messages use the same datatype, and messages are scattered from (or gathered to) sequential storage. (End of rationale.)

Advice to implementors. Although the discussion of collective communication in terms of point-to-point operation implies that each message is transferred directly from sender to receiver, implementations may use a tree communication pattern. Messages can be forwarded by intermediate nodes where they are split (for scatter) or concatenated (for gather), if this is more efficient. (End of advice to implementors.)

WIFI_ALL	recvtypes, comm)	ts, saispis, senatypes, recybur, recycounts, raispis,	18
IN	sendbuf	starting address of send buffer (choice)	19
		ũ ()	20
IN	sendcounts	non-negative integer array (of length group size)	21
		specifying the number of elements to send to each	22
		rank	23
IN	sdispls	integer array (of length group size). Entry j specifies	24
		the displacement in bytes (relative to $sendbuf)$ from	25
		which to take the outgoing data destined for process	26
		j (array of integers)	27 28
IN	sendtypes	array of datatypes (of length group size). Entry j	28 29
		specifies the type of data to send to process j (array	29 30
		of handles)	31
OUT	recvbuf	address of receive buffer (choice)	32
IN	recvcounts	non-negative integer array (of length group size)	33
		specifying the number of elements that can be	34
		received from each rank	35
IN	rdispls	integer array (of length group size). Entry i specifies	36
	laispis	the displacement in bytes (relative to recvbuf) at	37
		which to place the incoming data from process i	38
		(array of integers)	39
IN	recvtypes	array of datatypes (of length group size). Entry i	40
IIN	recvtypes	specifies the type of data received from process i	41
		(array of handles)	42
			43
IN	comm	communicator (handle)	44
~ • • •			45
C bindi	binding 46		

MPI_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls,

C binding

int MPI_Alltoallw(const void *sendbuf, const int sendcounts[], const int sdispls[], const MPI_Datatype sendtypes[],

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6

7

8

9 10

11

1213

14

151617

47

```
1
                    void *recvbuf, const int recvcounts[], const int rdispls[],
\mathbf{2}
                    const MPI_Datatype recvtypes[], MPI_Comm comm)
3
     int MPI_Alltoallw_c(const void *sendbuf, const MPI_Count sendcounts[],
4
                    const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],
5
                    void *recvbuf, const MPI_Count recvcounts[],
6
                    const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],
7
                    MPI_Comm comm)
8
9
     Fortran 2008 binding
10
     MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
11
                    rdispls, recvtypes, comm, ierror)
12
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
13
         INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
14
                     rdispls(*)
15
         TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
16
         TYPE(*), DIMENSION(...) :: recvbuf
17
         TYPE(MPI_Comm), INTENT(IN) :: comm
18
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
     MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
20
                    rdispls, recvtypes, comm, ierror)
21
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
22
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*),
23
                     recvcounts(*)
24
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
25
         TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
26
         TYPE(*), DIMENSION(...) :: recvbuf
27
         TYPE(MPI_Comm), INTENT(IN) :: comm
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
     Fortran binding
31
     MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS,
32
                    RDISPLS, RECVTYPES, COMM, IERROR)
33
          <type> SENDBUF(*), RECVBUF(*)
34
         INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
35
                     RDISPLS(*), RECVTYPES(*), COMM, IERROR
36
         MPI_ALLTOALLW is the most general form of complete exchange. Like
37
     MPI_TYPE_CREATE_STRUCT, the most general type constructor, MPI_ALLTOALLW al-
38
     lows separate specification of count, displacement and datatype. In addition, to allow max-
39
     imum flexibility, the displacement of blocks within the send and receive buffers is specified
40
     in bytes.
41
         If comm is an intra-communicator, then the j-th block sent from process i is received
42
     by process j and is placed in the i-th block of recvbuf. These blocks need not all have the
43
     same size.
44
         The type signature associated with sendcounts[i], sendtypes[i] at process i must be equal
45
     to the type signature associated with recvcounts[i], recvtypes[i] at process j. This implies that
46
     the amount of data sent must be equal to the amount of data received, pairwise between
47
     every pair of processes. Distinct type maps between sender and receiver are still allowed.
48
```

The outcome is as if each process sent a message to every other process with

MPI_Send(sendbuf+sdispls[i],sendcounts[i],sendtypes[i],i,...),

and received a message from every other process with a call to

MPI_Recv(recvbuf+rdispls[i],recvcounts[i],recvtypes[i],i,...).

All arguments on all processes are significant. The argument **comm** must describe the same communicator on all processes.

Like for MPI_ALLTOALLV, the "in place" option for intra-communicators is specified by passing MPI_IN_PLACE to the argument sendbuf at *all* processes. In such a case, sendcounts, sdispls and sendtypes are ignored. The data to be sent is taken from the recvbuf and replaced by the received data. Data sent and received must have the same type map as specified by the received and receives arrays, and is taken from the locations of the receive buffer specified by rdispls.

If comm is an inter-communicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The j-th send buffer of process i in group A should be consistent with the i-th receive buffer of process j in group B, and vice versa.

Rationale. The MPI_ALLTOALLW function generalizes several MPI functions by carefully selecting the input arguments. For example, by making all but one process have sendcounts[i] = 0, this achieves an MPI_SCATTERW function. (*End of rationale.*)

6.9 Global Reduction Operations

The functions in this section perform a global reduce operation (for example sum, maximum, and logical and) across all members of a group. The reduction operation can be either one of a predefined list of operations, or a user-defined operation. The global reduction functions come in several flavors: a reduce that returns the result of the reduction to one member of a group, an all-reduce that returns this result to all members of a group, and two scan (parallel prefix) operations. In addition, a reduce-scatter operation combines the functionality of a reduce and of a scatter operation.

 $\mathbf{2}$

```
222
                                        CHAPTER 6. COLLECTIVE COMMUNICATION
1
     6.9.1 Reduce
\mathbf{2}
3
4
     MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)
5
       IN
                sendbuf
                                            address of send buffer (choice)
6
       OUT
                recvbuf
                                            address of receive buffer (choice, significant only at
7
8
                                            root)
9
       IN
                count
                                            number of elements in send buffer (non-negative
10
                                            integer)
11
                                            datatype of elements of send buffer (handle)
       IN
                datatype
12
       IN
                                            reduce operation (handle)
13
                ор
14
       IN
                                            rank of root process (integer)
                root
15
       IN
                                            communicator (handle)
                comm
16
17
     C binding
18
     int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
19
                    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
20
21
     int MPI_Reduce_c(const void *sendbuf, void *recvbuf, MPI_Count count,
22
                    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
23
     Fortran 2008 binding
^{24}
     MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
25
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
26
         TYPE(*), DIMENSION(..) :: recvbuf
27
         INTEGER, INTENT(IN) :: count, root
28
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
         TYPE(MPI_Op), INTENT(IN) :: op
30
         TYPE(MPI_Comm), INTENT(IN) :: comm
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33
     MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
34
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
35
         TYPE(*), DIMENSION(..) :: recvbuf
36
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
37
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
38
         TYPE(MPI_Op), INTENT(IN) :: op
39
         INTEGER, INTENT(IN) :: root
40
         TYPE(MPI_Comm), INTENT(IN) :: comm
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
     Fortran binding
43
     MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
44
          <type> SENDBUF(*), RECVBUF(*)
45
          INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
46
47
48
```

If comm is an intra-communicator, MPI_REDUCE combines the elements provided in the input buffer of each process in the group, using the operation op, and returns the 3 combined value in the output buffer of the process with rank root. The input buffer is 4 defined by the arguments sendbuf, count and datatype; the output buffer is defined by the arguments recvbuf, count and datatype; both have the same number of elements, with the same type. The routine is called by all group members using the same arguments for 6 7 count, datatype, op, root and comm. Thus, all processes provide input buffers of the same length, with elements of the same type as the output buffer at the root. Each process can provide one element, or a sequence of elements, in which case the combine operation 10 is executed element-wise on each entry of the sequence. For example, if the operation 11is MPI_MAX and the send buffer contains two elements that are floating point numbers 12(count = 2 and datatype = MPI_FLOAT), then recvbuf(1) = $global \max(sendbuf(1))$ and 13 $recvbuf(2) = global \max(sendbuf(2)).$

Section 6.9.2, lists the set of predefined operations provided by MPI. That section also enumerates the datatypes to which each operation can be applied.

In addition, users may define their own operations that can be overloaded to operate on several datatypes, either basic or derived. This is further explained in Section 6.9.5.

The operation **op** is always assumed to be associative. All predefined operations are also assumed to be commutative. Users may define operations that are assumed to be associative, but not commutative. The "canonical" evaluation order of a reduction is determined by the ranks of the processes in the group. However, the implementation can take advantage of associativity, or associativity and commutativity in order to change the order of evaluation. This may change the result of the reduction for operations that are not strictly associative and commutative, such as floating point addition.

Advice to implementors. It is strongly recommended that MPI_REDUCE be implemented so that the same result be obtained whenever the function is applied on the same arguments, appearing in the same order. Note that this may prevent optimizations that take advantage of the physical location of ranks. (End of advice to *implementors.*)

Advice to users. Some applications may not be able to ignore the non-associative nature of floating-point operations or may use user-defined operations (see Section 6.9.5) that require a special reduction order and cannot be treated as associative. Such applications should enforce the order of evaluation explicitly. For example, in the case of operations that require a strict left-to-right (or right-to-left) evaluation order, this could be done by gathering all operands at a single process (e.g., with MPI_GATHER), applying the reduction operation in the desired order (e.g., with MPI_REDUCE_LOCAL), and if needed, broadcast or scatter the result to the other processes (e.g., with MPI_BCAST). (End of advice to users.)

The datatype argument of MPI_REDUCE must be compatible with op. Predefined operators work only with the MPI types listed in Section 6.9.2 and Section 6.9.4. Furthermore, the datatype and op given for predefined operators must be the same on all processes.

Note that it is possible for users to supply different user-defined operations to MPI_REDUCE in each process. MPI does not define which operations are used on which operands in this case. User-defined operators may operate on general, derived datatypes. In this case, each argument that the reduce operation is applied to is one element described

Unofficial Draft for Comment Only

1 $\mathbf{2}$

 $\mathbf{5}$

8 9

1415

16

17

18

19

20

21

22

23 24

25

26

27

28

29

30 31

32

33

34

35

36

37

38

39

40 41

42

43

44

45

46

47

¹ by such a datatype, which may contain several basic values. This is further explained in
 ² Section 6.9.5.
 ³

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 intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Only send buffer arguments are significant in group B and only receive buffer arguments are significant at the root.

¹⁹ ₂₀ 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,

²³ MPI_SCAN, MPI_EXSCAN, all nonblocking variants of those (see Section 6.12), and

²⁴ MPI_REDUCE_LOCAL. These operations are invoked by placing the following in op.

25	
26	

45 46

27	Name	Meaning
28		
29	MPI_MAX	maximum
30	MPI_MIN	minimum
31	MPI_SUM	sum
32	MPI_PROD	product
33	MPI_LAND	logical and
34	MPI_BAND	bit-wise and
35	MPI_LOR	logical or
36	MPI_BOR	bit-wise or
37	MPI_LXOR	logical exclusive or (xor)
38	MPI_BXOR	bit-wise exclusive or (xor)
	MPI_MAXLOC	max value and location
39	MPI_MINLOC	min value and location
40	-	

The two operations MPI_MINLOC and MPI_MAXLOC are discussed separately in 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.

47	C integer:	MPI_INT, MPI_LONG, MPI_SHORT,
48		MPI_UNSIGNED_SHORT, MPI_UNSIGNED,

4

5

6 7

8

9

10

11

12

13

14

15

16

17

	MPI_UNSIGNED_LONG,	1
	MPI_LONG_LONG_INT,	2
	MPI_LONG_LONG (as synonym),	3
	MPI_UNSIGNED_LONG_LONG,	4
	MPI_SIGNED_CHAR,	5
	MPI_UNSIGNED_CHAR,	6
	MPI_INT8_T, MPI_INT16_T,	7
	MPI_INT32_T, MPI_INT64_T,	8
	MPI_UINT8_T, MPI_UINT16_T,	9
	MPI_UINT32_T, and MPI_UINT64_T	10
Fortran integer:	MPI_INTEGER	11
0	and handles returned from	12
	MPI_TYPE_CREATE_F90_INTEGER	12
	and, if available, MPI_INTEGER1,	
	MPI_INTEGER2, MPI_INTEGER4,	14
	MPI_INTEGER8, and MPI_INTEGER16	15
Floating point:	MPI_FLOAT, MPI_DOUBLE, MPI_REAL,	16
	MPI_DOUBLE_PRECISION,	17
	MPI_LONG_DOUBLE,	18
	and handles returned from	19
	MPI_TYPE_CREATE_F90_REAL	20
	and, if available, MPI_REAL2,	21
	MPI_REAL4, MPI_REAL8, and MPI_REAL16	22
Logical:	MPI_LOGICAL, MPI_C_BOOL,	23
0	and MPI_CXX_BOOL	24
Complex:	MPI_COMPLEX, MPI_C_COMPLEX,	25
	MPI_C_FLOAT_COMPLEX (as synonym),	26
	MPI_C_DOUBLE_COMPLEX,	27
	MPI_C_LONG_DOUBLE_COMPLEX,	28
	MPI_CXX_FLOAT_COMPLEX,	29
	MPI_CXX_DOUBLE_COMPLEX,	30
	MPI_CXX_LONG_DOUBLE_COMPLEX,	31
	and handles returned from	32
	MPI_TYPE_CREATE_F90_COMPLEX	33
	and, if available, MPI_DOUBLE_COMPLEX,	34
	MPI_COMPLEX4, MPI_COMPLEX8,	35
	MPI_COMPLEX16, and MPI_COMPLEX32	36
Byte:	MPI_BYTE	37
Multi-language types:	MPI_AINT, MPI_OFFSET, and MPI_COUNT	38
		39
Now, the valid datatypes for each op	peration are specified below.	40
		40
Ora	Allowed Trues	41
Ор	Allowed Types	
	Cinterna Fostara interna Flortina acint	43
MPI_MAX, MPI_MIN	C integer, Fortran integer, Floating point,	44
	Multi-language types	45
MPI_SUM, MPI_PROD	C integer, Fortran integer, Floating point, Complex, Multi language types	46
	Multi-language types	47
MPI_LAND, MPI_LOR, MPI_LXOR	C integer, Logical	48

Unofficial Draft for Comment Only

```
1
       MPI_BAND, MPI_BOR, MPI_BXOR
                                             C integer, Fortran integer, Byte, Multi-language types
\mathbf{2}
          These operations together with all listed datatypes are valid in all supported program-
3
     ming languages, see also Reduce Operations on page 838 in Section 19.3.6.
4
          The following examples use intra-communicators.
5
6
     Example 6.15 A routine that computes the dot product of two vectors that are distributed
7
     across a group of processes and returns the answer at node zero.
8
9
     SUBROUTINE PAR_BLAS1(m, a, b, c, comm)
10
     REAL a(m), b(m)
                               ! local slice of array
11
     REAL c
                               ! result (at node zero)
12
     REAL sum
13
     INTEGER m, comm, i, ierr
14
15
     ! local sum
16
     sum = 0.0
17
     DO i = 1, m
18
         sum = sum + a(i)*b(i)
19
     END DO
20
21
     ! global sum
     CALL MPI_REDUCE(sum, c, 1, MPI_REAL, MPI_SUM, 0, comm, ierr)
22
23
     RETURN
^{24}
     END
25
26
     Example 6.16 A routine that computes the product of a vector and an array that are
27
     distributed across a group of processes and returns the answer at node zero.
28
     SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
29
30
                              ! local slice of array
     REAL a(m), b(m,n)
31
     REAL c(n)
                              ! result
32
     REAL sum(n)
33
     INTEGER n, comm, i, j, ierr
34
35
     ! local sum
36
     DO j=1,n
37
         sum(j) = 0.0
38
         DO i=1,m
39
            sum(j) = sum(j) + a(i)*b(i,j)
40
         END DO
41
     END DO
42
43
     ! global sum
44
     CALL MPI_REDUCE(sum, c, n, MPI_REAL, MPI_SUM, 0, comm, ierr)
45
46
     ! return result at node zero (and garbage at the other nodes)
47
     RETURN
48
     END
```

6.9.3 Signed Characters and Reductions

The types MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR can be used in reduction operations. MPI_CHAR, MPI_WCHAR, and MPI_CHARACTER (which represent printable characters) cannot be used in reduction operations. In a heterogeneous environment, MPI_CHAR, MPI_WCHAR, and MPI_CHARACTER will be translated so as to preserve the printable character, whereas MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR will be translated so as to preserve the integer value.

Advice to users. The types MPI_CHAR, MPI_WCHAR, and MPI_CHARACTER are intended for characters, and so will be translated to preserve the printable representation, rather than the integer value, if sent between machines with different character codes. The types MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR should be used in C if the integer value should be preserved. (*End of advice to users.*)

6.9.4 MINLOC and MAXLOC

The operator MPI_MINLOC is used to compute a global minimum and also an index attached to the minimum value. MPI_MAXLOC similarly computes a global maximum and index. One application of these is to compute a global minimum (maximum) and the rank of the process containing this value.

The operation that defines MPI_MAXLOC is:

$$\left(\begin{array}{c} u\\i\end{array}\right)\circ\left(\begin{array}{c} v\\j\end{array}\right)=\left(\begin{array}{c} w\\k\end{array}\right)$$

where

$$w = \max(u, v)$$

and

$$k = \begin{cases} i & \text{if } u > v \\ \min(i, j) & \text{if } u = v \\ j & \text{if } u < v \end{cases}$$

MPI_MINLOC is defined similarly:

$$\left(\begin{array}{c} u\\i\end{array}\right)\circ\left(\begin{array}{c} v\\j\end{array}\right)=\left(\begin{array}{c} w\\k\end{array}\right)$$

where

$$w = \min(u, v)$$

and

$$k = \begin{cases} i & \text{if } u < v \\ \min(i,j) & \text{if } u = v \\ j & \text{if } u > v \end{cases}$$

Both operations are associative and commutative. Note that if MPI_MAXLOC is applied to reduce a sequence of pairs $(u_0, 0), (u_1, 1), \ldots, (u_{n-1}, n-1)$, then the value returned is

Unofficial Draft for Comment Only

1 (u, r), where $u = \max_i u_i$ and r is the index of the first global maximum in the sequence. $\mathbf{2}$ Thus, if each process supplies a value and its rank within the group, then a reduce operation 3 with $op = MPI_MAXLOC$ will return the maximum value and the rank of the first process with 4 that value. Similarly, MPI_MINLOC can be used to return a minimum and its index. More 5generally, MPI_MINLOC computes a *lexicographic minimum*, where elements are ordered 6 according to the first component of each pair, and ties are resolved according to the second 7component. 8 The reduce operation is defined to operate on arguments that consist of a pair: value 9 and index. For both Fortran and C, types are provided to describe the pair. The potentially 10 mixed-type nature of such arguments is a problem in Fortran. The problem is circumvented, 11for Fortran, by having the MPI-provided type consist of a pair of the same type as value, 12and coercing the index to this type also. In C, the MPI-provided pair type has distinct 13 types and the index is an int. 14In order to use MPI_MINLOC and MPI_MAXLOC in a reduce operation, one must provide 15a datatype argument that represents a pair (value and index). MPI provides nine such 16predefined datatypes. The operations MPI_MAXLOC and MPI_MINLOC can be used with 17each of the following datatypes. 18 Fortran: 19 Name Description 20MPI_2REAL pair of REALs 21MPI_2DOUBLE_PRECISION pair of DOUBLE PRECISION variables 22 pair of INTEGERS MPI_2INTEGER 23242526C: 27Name Description float and int MPI_FLOAT_INT 28double and int MPI_DOUBLE_INT 29long and int MPI_LONG_INT 30 MPI_2INT pair of int 31MPI_SHORT_INT short and int 32 MPI_LONG_DOUBLE_INT long double and int 33 34 The datatype MPI_2REAL is as if defined by the following (see Section 5.1). 35 36 MPI_Type_contiguous(2, MPI_REAL, MPI_2REAL); 37 38 Similar statements apply for MPI_2INTEGER, MPI_2DOUBLE_PRECISION, and MPI_2INT. 39 The datatype MPI_SHORT_INT is as if defined by the following sequence of instructions. 40 struct mystruct { 41 short val; 42int rank; 43 }; 44 type[0] = MPI_SHORT; 45 type[1] = MPI_INT; 46 disp[0] = 0;47disp[1] = offsetof(struct mystruct, rank); 48

```
1
block[0] = 1;
                                                                                          \mathbf{2}
block[1] = 1;
                                                                                          3
MPI_Type_create_struct(2, block, disp, type, MPI_SHORT_INT);
                                                                                          4
Similar statements apply for MPI_FLOAT_INT, MPI_LONG_INT and MPI_DOUBLE_INT.
                                                                                          5
    The following examples use intra-communicators.
                                                                                          6
                                                                                          7
Example 6.17 Each process has an array of 30 doubles, in C. For each of the 30 locations,
                                                                                          8
compute the value and rank of the process containing the largest value.
                                                                                          9
                                                                                          10
    . . .
                                                                                          11
    /* each process has an array of 30 double: ain[30]
                                                                                          12
     */
                                                                                          13
    double ain[30], aout[30];
                                                                                          14
    int ind[30];
                                                                                          15
    struct {
                                                                                          16
        double val;
                                                                                          17
         int
               rank;
                                                                                          18
    } in[30], out[30];
                                                                                          19
    int i, myrank, root;
                                                                                          20
                                                                                          21
    MPI_Comm_rank(comm, &myrank);
                                                                                          22
    for (i=0; i<30; ++i) {
                                                                                          23
         in[i].val = ain[i];
                                                                                          ^{24}
         in[i].rank = myrank;
                                                                                          25
    }
                                                                                          26
    MPI_Reduce(in, out, 30, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm);
                                                                                          27
    /* At this point, the answer resides on process root
                                                                                          28
     */
                                                                                          29
    if (myrank == root) {
                                                                                          30
         /* read ranks out
                                                                                          31
          */
                                                                                          32
         for (i=0; i<30; ++i) {
                                                                                          33
             aout[i] = out[i].val;
                                                                                          34
             ind[i] = out[i].rank;
                                                                                          35
                                                                                          36
    }
                                                                                          37
                                                                                          38
                                                                                          39
Example 6.18 Same example, in Fortran.
                                                                                          40
                                                                                          41
. . .
                                                                                          42
! each process has an array of 30 double: ain(30)
                                                                                          43
DOUBLE PRECISION ain(30), aout(30)
                                                                                          44
INTEGER ind(30)
                                                                                          45
DOUBLE PRECISION in(2,30), out(2,30)
                                                                                          46
INTEGER i, myrank, root, ierr
                                                                                          47
                                                                                          48
```

```
1
     CALL MPI_COMM_RANK(comm, myrank, ierr)
\mathbf{2}
     DO i=1,30
3
        in(1,i) = ain(i)
4
        in(2,i) = myrank
                               ! myrank is coerced to a double
\mathbf{5}
     END DO
6
7
     CALL MPI_REDUCE(in, out, 30, MPI_2DOUBLE_PRECISION, MPI_MAXLOC, root,&
8
                       comm, ierr)
9
     ! At this point, the answer resides on process root
10
^{11}
     IF (myrank .EQ. root) THEN
12
         ! read ranks out
13
        DO i=1,30
14
            aout(i) = out(1,i)
15
            ind(i) = out(2,i) ! rank is coerced back to an integer
16
        END DO
17
     END IF
18
19
     Example 6.19 Each process has a non-empty array of values. Find the minimum global
20
     value, the rank of the process that holds it and its index on this process.
21
22
     #define LEN
                      1000
23
24
                               /* local array of values */
     float val[LEN];
25
                               /* local number of values */
     int count;
26
     int myrank, minrank, minindex;
27
     float minval;
28
29
     struct {
30
         float value;
^{31}
         int
               index;
32
     } in, out;
33
34
         /* local minloc */
35
     in.value = val[0];
36
     in.index = 0;
37
     for (i=1; i < count; i++)</pre>
38
         if (in.value > val[i]) {
39
              in.value = val[i];
40
              in.index = i;
41
         }
42
43
         /* global minloc */
44
     MPI_Comm_rank(comm, &myrank);
45
     in.index = myrank*LEN + in.index;
46
     MPI_Reduce(&in, &out, 1, MPI_FLOAT_INT, MPI_MINLOC, root, comm);
47
         /* At this point, the answer resides on process root
48
```

```
*/
if (myrank == root) {
    /* read answer out
    */
    minval = out.value;
    minrank = out.index / LEN;
    minindex = out.index % LEN;
}
```

Rationale. The definition of MPI_MINLOC and MPI_MAXLOC given here has the advantage that it does not require any special-case handling of these two operations: they are handled like any other reduce operation. A programmer can provide his or her own definition of MPI_MAXLOC and MPI_MINLOC, if so desired. The disadvantage is that values and indices have to be first interleaved, and that indices and values have to be coerced to the same type, in Fortran. (*End of rationale.*)

6.9.5 User-Defined Reduction Operations

TYPE(MPI_Op), INTENT(OUT) :: op

MPI_OP_CREATE(USER_FN, COMMUTE, OP, IERROR)

Fortran binding

EXTERNAL USER_FN

INTEGER OP, IERROR

LOGICAL COMMUTE

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_OP_CREATE(user_fn, commute, op) IN user_fn user defined function (function) IN commute true if commutative; false otherwise. OUT operation (handle) op 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), INTENT(IN) :: 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) PROCEDURE(MPI_User_function_c), INTENT(IN) :: user_fn LOGICAL, INTENT(IN) :: commute

1

2

6

10

11

12

13

14

15 16 17

18 19 20

21

22

23 24

25 26

27

28

29 30

31

32

33

34

35

36 37

38

39 40

41

42

43

44

45

46

1	MPI_OP_CREATE binds a user-defined reduction operation to an
2	op handle that can subsequently be used in MPI_REDUCE, MPI_ALLREDUCE,
3	MPI_REDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, MPI_SCAN,
4	MPI_EXSCAN, all nonblocking variants of those (see Section 6.12), and
5	MPI_REDUCE_LOCAL. The user-defined operation is assumed to be associative. If commute
6	*
7	= true, then the operation should be both commutative and associative. If commute = false,
8	then the order of operands is fixed and is defined to be in ascending, process rank order,
9	beginning with process zero. The order of evaluation can be changed, talking advantage of
10	the associativity of the operation. If commute = true then the order of evaluation can be changed, taking advantage of commutativity and associativity.
11	
12	The argument user_fn is the user-defined function, which must have the following four
13	arguments: invec, inoutvec, len, and datatype.
14	MPI_USER_FUNCTION also supports large count types in separate additional MPI precedures in C (sufficient with the " e ") and interface relevant types in Ferture when using
15	procedures in C (suffixed with the "_c") and interface polymorphism in Fortran when using
16	USE mpi_f08. The ISO C prototypes for the functions are the following.
17	typedef void MPI_User_function(void *invec, void *inoutvec, int *len,
18	
19	<pre>MPI_Datatype *datatype);</pre>
20	<pre>typedef void MPI_User_function_c(void *invec, void *inoutvec,</pre>
21	<pre>MPI_Count *len, MPI_Datatype *datatype);</pre>
22	The Fortran declarations of the user-defined function user_fn appear below.
23	ABSTRACT INTERFACE
24	SUBROUTINE MPI_User_function(invec, inoutvec, len, datatype)
25	USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
26	TYPE(C_PTR), VALUE :: invec, inoutvec
27	INTEGER :: len
28	TYPE(MPI_Datatype) :: datatype
29	III L(MII_Datatype) datatype
30	ABSTRACT INTERFACE
31	SUBROUTINE MPI_User_function_c(invec, inoutvec, len, datatype)
32	USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
33	TYPE(C_PTR), VALUE :: invec, inoutvec
34	INTEGER(KIND=MPI_COUNT_KIND) :: len
35	TYPE(MPI_Datatype) :: datatype
36	SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, DATATYPE)
37	<pre><type> INVEC(LEN), INOUTVEC(LEN)</type></pre>
38	INTEGER LEN, DATATYPE
39	
40	The datatype argument is a handle to the datatype that was passed into the call to
41	MPI_REDUCE. The user reduce function should be written such that the following holds:
42	Let $u[0], \ldots, u[len-1]$ be the len elements in the communication buffer described by the
43	arguments invec, len and datatype when the function is invoked; let $v[0], \ldots, v[len-1]$ be len
44	elements in the communication buffer described by the arguments inoutvec, len and datatype
45	when the function is invoked; let $w[0], \ldots, w[len-1]$ be len elements in the communication
46	buffer described by the arguments inoutvec, len and datatype when the function returns;
47	then $w[i]=u[i]\circ v[i],$ for $i{=}0$, \dots , $len{-}1,$ where \circ is the reduce operation that the function
48	computes.

Informally, we can think of invec and inoutvec as arrays of len elements that user_fn is combining. The result of the reduction over-writes values in inoutvec, hence the name. Each invocation of the function results in the pointwise evaluation of the reduce operator on len elements: i.e., the function returns in inoutvec[i] the value invec[i] \circ inoutvec[i], for i=0, ..., count-1, where \circ is the combining operation computed by the function.

Rationale. The len argument allows MPI_REDUCE to avoid calling the function for each element in the input buffer. Rather, the system can choose to apply the function to chunks of input. In C, it is passed in as a reference for reasons of compatibility with Fortran.

By internally comparing the value of the datatype argument to known, global handles, it is possible to overload the use of a single user-defined function for several, different datatypes. (*End of rationale.*)

When calling any reduction or prefix scan MPI procedure with a user-defined MPI operator, the type of the count parameter in the call to the reduction or prefix scan MPI procedure does not need to be identical to the type of the len parameter in the user function associated with the user-defined MPI operator. If the count parameter has a type of int in C or INTEGER in Fortran and the len parameter has a type of MPI_COUNT, then MPI will perform the appropriate widening type conversion of the len parameter. If the count parameter has a type of MPI_COUNT and the len parameter has a type of int in C or INTEGER in Fortran, then MPI will perform the appropriate narrowing type conversion of the len parameter. If this narrowing conversion would result in truncation of the len value, then MPI will call the user function multiple times with a sequence of values for len that sum to the value of count.

Advice to implementors. If the number of data items cannot be represented in len, the implementation may need to invoke user_fn multiple times. (End of advice to implementors.)

General datatypes may be passed to the user function. However, use of datatypes that are not contiguous is likely to lead to inefficiencies.

No MPI communication function may be called inside the user function. MPI_ABORT may be called inside the function in case of an error.

Advice to users. Suppose one defines a library of user-defined reduce functions 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. 48

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\overline{7}$

 24

 31

1 Users who plan to mix languages should define their reduction functions accordingly. 2 (End of advice to users.) 3 Advice to implementors. We outline below a naive and inefficient implementation of 4 MPI_REDUCE not supporting the "in place" option. 56 MPI_Comm_size(comm, &groupsize); 7 MPI_Comm_rank(comm, &rank); 8 if (rank > 0) { 9 MPI_Recv(tempbuf, count, datatype, rank-1,...); 10 User_reduce(tempbuf, sendbuf, count, datatype); 11 } 12if (rank < groupsize-1) {</pre> 13 MPI_Send(sendbuf, count, datatype, rank+1, ...); 14 } 15/* answer now resides in process groupsize-1 ... now send to root 16*/ 17 if (rank == root) { 18 MPI_Irecv(recvbuf, count, datatype, groupsize-1,..., &req); 19 } 20if (rank == groupsize-1) { 21MPI_Send(sendbuf, count, datatype, root, ...); 22 } 23 if (rank == root) { 24MPI_Wait(&req, &status); 25} 2627The reduction computation proceeds, sequentially, from process 0 to process 28groupsize-1. This order is chosen so as to respect the order of a possibly non-29 commutative operator defined by the function User_reduce(). A more efficient im-30 plementation is achieved by taking advantage of associativity and using a logarithmic 31tree reduction. Commutativity can be used to advantage, for those cases in which 32 the commute argument to MPI_OP_CREATE is true. Also, the amount of temporary 33 buffer required can be reduced, and communication can be pipelined with computa-34 tion, by transferring and reducing the elements in chunks of size len <count. 35 The predefined reduce operations can be implemented as a library of user-defined 36 operations. However, better performance might be achieved if MPI_REDUCE handles 37 these functions as a special case. (End of advice to implementors.) 38 39 40 41 MPI_OP_FREE(op) 42INOUT operation (handle) op 43 44 C binding 45 int MPI_Op_free(MPI_Op *op) 46 47Fortran 2008 binding 48 MPI_Op_free(op, ierror)

TYPE(MPI_Op), INTENT(INOUT) :: op	1
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	2 3
Fortran binding	
MPI_OP_FREE(OP, IERROR)	4 5
INTEGER OP, IERROR	6
Marks a user defined reduction expertion for deallocation and sets on to MPL OP NULL	7
Marks a user-defined reduction operation for deallocation and sets op to MPI_OP_NULL.	8
Example of User-Defined Reduce	9
	10
It is time for an example of user-defined reduction. The example in this section uses an	11
intra-communicator.	12
Example 6.20 Compute the product of an array of complex numbers, in C.	13
	14
typedef struct {	15
double real, imag;	16
<pre>} Complex;</pre>	17 18
/ the new defined function	19
/* the user-defined function	20
<pre>*/ void myProd(void *inP, void *inoutP, int *len, MPI_Datatype *dptr)</pre>	21
{	22
int i;	23
Complex c;	24
Complex c, Complex *in = (Complex *)inP, *inout = (Complex *)inoutP;	25
comprex (comprex) in , (comprex) induct,	26
<pre>for (i=0; i< *len; ++i) {</pre>	27
c.real = inout->real +in->real -	28
inout->imag*in->imag;	29
c.imag = inout->real*in->imag +	30
inout->imag*in->real;	31
<pre>*inout = c;</pre>	32
<pre>in++; inout++;</pre>	33
}	34
}	35
	36
/* and, to call it	37
*/	38
	39
	40
/* each process has an array of 100 Complexes	41
*/	42
Complex a[100], answer[100];	43
MPI_Op myOp;	44
MPI_Datatype ctype;	45 46
	40 47
<pre>/* explain to MPI how type Complex is defined */</pre>	48
*/	

```
1
          MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
\mathbf{2}
          MPI_Type_commit(&ctype);
3
          /* create the complex-product user-op
4
           */
5
          MPI_Op_create(myProd, 1, &myOp);
6
7
          MPI_Reduce(a, answer, 100, ctype, myOp, root, comm);
8
9
          /* At this point, the answer, which consists of 100 Complexes,
10
           * resides on process root
11
           */
12
13
     Example 6.21 How to use the mpi_f08 interface of the Fortran MPI_User_function.
14
     subroutine my_user_function(invec, inoutvec, len, type) bind(c)
15
         use, intrinsic :: iso_c_binding, only : c_ptr, c_f_pointer
16
         use mpi_f08
17
         type(c_ptr), value :: invec, inoutvec
18
         integer :: len
19
         type(MPI_Datatype) :: type
20
         real, pointer :: invec_r(:), inoutvec_r(:)
21
         if (type%MPI_VAL == MPI_REAL%MPI_VAL) then
22
            call c_f_pointer(invec, invec_r, (/ len /))
23
            call c_f_pointer(inoutvec, inoutvec_r, (/ len /))
24
            inoutvec_r = invec_r + inoutvec_r
25
         end if
26
     end subroutine
27
28
            All-Reduce
     6.9.6
29
30
     MPI includes a variant of the reduce operations where the result is returned to all processes
^{31}
     in a group. MPI requires that all processes from the same group participating in these
32
     operations receive identical results.
33
34
35
     MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)
36
       IN
                 sendbuf
                                             starting address of send buffer (choice)
37
       OUT
                 recvbuf
                                             starting address of receive buffer (choice)
38
39
       IN
                 count
                                             number of elements in send buffer (non-negative
40
                                             integer)
41
       IN
                                             datatype of elements of send buffer (handle)
                 datatype
42
       IN
                                             operation (handle)
                 op
43
       IN
                 comm
                                             communicator (handle)
44
45
46
     C binding
47
     int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,
48
                    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)	1 2
Fortran 2008 binding	3
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror)	4
TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf	5
TYPE(*), DIMENSION() :: recvbuf	6
INTEGER, INTENT(IN) :: count	7
TYPE(MPI_Datatype), INTENT(IN) :: datatype	8
TYPE(MPI_Op), INTENT(IN) :: op	9
TYPE(MPI_Comm), INTENT(IN) :: comm	10
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	11
	12
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror)	13
TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf	14
TYPE(*), DIMENSION() :: recvbuf	15 16
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count	10
TYPE(MPI_Datatype), INTENT(IN) :: datatype	18
TYPE(MPI_Op), INTENT(IN) :: op	19
TYPE(MPI_Comm), INTENT(IN) :: comm	20
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	20
Fortran binding	22
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)	23
<type> SENDBUF(*), RECVBUF(*)</type>	24
INTEGER COUNT, DATATYPE, OP, COMM, IERROR	25
	26
If comm is an intra-communicator, MPI_ALLREDUCE behaves the same as	27
MPI_REDUCE except that the result appears in the receive buffer of all the group members.	28
Advice to implementors. The all-reduce operations can be implemented as a re-	29
duce, followed by a broadcast. However, a direct implementation can lead to better	30
performance. (End of advice to implementors.)	31
perioritance. (End of addice to implementors.)	32
The "in place" option for intra-communicators is specified by passing the value	33
MPI_IN_PLACE to the argument sendbuf at all processes. In this case, the input data is	34
taken at each process from the receive buffer, where it will be replaced by the output data.	35
If comm is an inter-communicator, then the result of the reduction of the data provided	36
by processes in group A is stored at each process in group B, and vice versa. Both groups	37
should provide count and datatype arguments that specify the same type signature.	38
The following example uses an intra-communicator.	39
	40
Example 6.22 A routine that computes the product of a vector and an array that are	41
distributed across a group of processes and returns the answer at all nodes (see also Exam-	42
ple 6.16).	43
	44
	45
	46
	47
	48

```
1
     SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
\mathbf{2}
     REAL a(m), b(m,n)
                              ! local slice of array
3
     REAL c(n)
                              ! result
4
     REAL sum(n)
\mathbf{5}
     INTEGER n, comm, i, j, ierr
6
7
      ! local sum
     DO j=1,n
8
9
         sum(j) = 0.0
10
         DO i=1,m
11
            sum(j) = sum(j) + a(i)*b(i,j)
12
         END DO
13
     END DO
14
15
      ! global sum
16
     CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, comm, ierr)
17
^{18}
      ! return result at all nodes
19
     RETURN
20
     END
21
22
             Process-Local Reduction
     6.9.7
23
     The functions in this section are of importance to library implementors who may want to
24
      implement special reduction patterns that are otherwise not easily covered by the standard
25
      MPI operations.
26
          The following function applies a reduction operator to local arguments.
27
28
29
      MPI_REDUCE_LOCAL(inbuf, inoutbuf, count, datatype, op)
30
       IN
                  inbuf
                                              input buffer (choice)
^{31}
32
       INOUT
                 inoutbuf
                                              combined input and output buffer (choice)
33
       IN
                 count
                                              number of elements in inbuf and inoutbuf buffers
34
                                               (non-negative integer)
35
       IN
                  datatype
                                              datatype of elements of inbuf and inoutbuf buffers
36
                                               (handle)
37
38
       IN
                                              operation (handle)
                  op
39
40
     C binding
41
      int MPI_Reduce_local(const void *inbuf, void *inoutbuf, int count,
42
                     MPI_Datatype datatype, MPI_Op op)
43
      int MPI_Reduce_local_c(const void *inbuf, void *inoutbuf, MPI_Count count,
44
                     MPI_Datatype datatype, MPI_Op op)
45
46
      Fortran 2008 binding
47
      MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror)
48
```

```
1
    TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
                                                                                        2
    TYPE(*), DIMENSION(..) :: inoutbuf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                        4
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                        5
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                        6
MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror)
    TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
                                                                                        9
    TYPE(*), DIMENSION(..) :: inoutbuf
                                                                                        10
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                        11
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                        12
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                        13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                        14
                                                                                        15
Fortran binding
                                                                                        16
MPI_REDUCE_LOCAL(INBUF, INOUTBUF, COUNT, DATATYPE, OP, IERROR)
                                                                                        17
    <type> INBUF(*), INOUTBUF(*)
                                                                                        18
    INTEGER COUNT, DATATYPE, OP, IERROR
                                                                                        19
    The function applies the operation given by op element-wise to the elements of inbuf
                                                                                        20
and inoutbuf with the result stored element-wise in inoutbuf, as explained for user-defined
                                                                                        21
operations in Section 6.9.5. Both inbuf and inoutbuf (input as well as result) have the
                                                                                        22
same number of elements given by count and the same datatype given by datatype. The
                                                                                        23
MPI_IN_PLACE option is not allowed.
                                                                                        24
    Reduction operations can be queried for their commutativity.
                                                                                        25
                                                                                        26
                                                                                        27
MPI_OP_COMMUTATIVE(op, commute)
                                                                                        28
  IN
                                      operation (handle)
           op
                                                                                        29
  OUT
           commute
                                      true if op is commutative, false otherwise (logical)
                                                                                        30
                                                                                        31
                                                                                        32
C binding
                                                                                        33
int MPI_Op_commutative(MPI_Op op, int *commute)
                                                                                        34
Fortran 2008 binding
                                                                                        35
MPI_Op_commutative(op, commute, ierror)
                                                                                        36
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                        37
    LOGICAL, INTENT(OUT) :: commute
                                                                                        38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                        39
                                                                                        40
Fortran binding
                                                                                        41
MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR)
                                                                                        42
    INTEGER OP, IERROR
                                                                                        43
    LOGICAL COMMUTE
                                                                                        44
                                                                                        45
                                                                                        46
```

Reduce-Scatter 6.10 1 2 MPI includes variants of the reduce operations where the result is scattered to all processes 3 in a group on return. One variant scatters equal-sized blocks to all processes, while another 4 variant scatters blocks that may vary in size for each process. 56 6.10.1 MPI_REDUCE_SCATTER_BLOCK 7 8 9 10 MPI_REDUCE_SCATTER_BLOCK(sendbuf, recvbuf, recvcount, datatype, op, comm) 11 IN sendbuf starting address of send buffer (choice) 12OUT recvbuf starting address of receive buffer (choice) 13 14IN element count per block (non-negative integer) recvcount 15datatype of elements of send and receive buffers IN datatype 16 (handle) 17IN operation (handle) 18 op 19 IN comm communicator (handle) 2021C binding 22 int MPI_Reduce_scatter_block(const void *sendbuf, void *recvbuf, 23int recvcount, MPI_Datatype datatype, MPI_Op op, 24MPI_Comm comm) 2526int MPI_Reduce_scatter_block_c(const void *sendbuf, void *recvbuf, MPI_Count recvcount, MPI_Datatype datatype, MPI_Op op, 27MPI_Comm comm) 2829 Fortran 2008 binding 30 MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm, 31 ierror) 32 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf 33 TYPE(*), DIMENSION(..) :: recvbuf 34 INTEGER, INTENT(IN) :: recvcount 35TYPE(MPI_Datatype), INTENT(IN) :: datatype 36 TYPE(MPI_Op), INTENT(IN) :: op 37 TYPE(MPI_Comm), INTENT(IN) :: comm 38 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 39 40MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm, 41 ierror) 42TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf TYPE(*), DIMENSION(..) :: recvbuf 43 44INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount 45TYPE(MPI_Datatype), INTENT(IN) :: datatype 46TYPE(MPI_Op), INTENT(IN) :: op 47 TYPE(MPI_Comm), INTENT(IN) :: comm 48 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Unofficial Draft for Comment Only

Fortran binding MPI_REDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR If server is the server is the MDL DEDUCE SCATTED DLOCK for the form

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.

 $\mathbf{2}$

 24

```
1
     MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm)
\mathbf{2}
       IN
                sendbuf
                                            starting address of send buffer (choice)
3
       OUT
                recvbuf
                                            starting address of receive buffer (choice)
4
5
       IN
                                            non-negative integer array (of length group size)
                 recvcounts
6
                                            specifying the number of elements of the result
7
                                            distributed to each process.
8
       IN
                                            datatype of elements of send and receive buffers
                datatype
9
                                            (handle)
10
       IN
                                            operation (handle)
                 ор
11
12
       IN
                 comm
                                            communicator (handle)
13
14
     C binding
15
     int MPI_Reduce_scatter(const void *sendbuf, void *recvbuf,
16
                    const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
17
                    MPI_Comm comm)
18
     int MPI_Reduce_scatter_c(const void *sendbuf, void *recvbuf,
19
                    const MPI_Count recvcounts[], MPI_Datatype datatype,
20
                    MPI_Op op, MPI_Comm comm)
21
22
     Fortran 2008 binding
23
     MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
24
                    ierror)
25
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
26
         TYPE(*), DIMENSION(..) :: recvbuf
27
         INTEGER, INTENT(IN) :: recvcounts(*)
28
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
         TYPE(MPI_Op), INTENT(IN) :: op
30
         TYPE(MPI_Comm), INTENT(IN) :: comm
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
     MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
33
34
                    ierror)
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
35
         TYPE(*), DIMENSION(..) :: recvbuf
36
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcounts(*)
37
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
38
         TYPE(MPI_Op), INTENT(IN) :: op
39
         TYPE(MPI_Comm), INTENT(IN) :: comm
40
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
     Fortran binding
43
     MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
44
                    IERROR)
45
          <type> SENDBUF(*), RECVBUF(*)
46
         INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR
47
48
```

If comm is an intra-communicator, MPI_REDUCE_SCATTER first performs a global, element-wise reduction on vectors of count = $\sum_{i=0}^{n-1} \text{recvcounts}[i]$ elements in the send buffers defined by sendbuf, count and datatype, using the operation op, where n is the number of processes in the group of comm. The routine is called by all group members using the same arguments for recvcounts, datatype, op and comm. The resulting vector is treated as n consecutive blocks where the number of elements of the i-th block is recvcounts[i]. The blocks are scattered to the processes of the group. The i-th block is sent to process i and stored in the receive buffer defined by recvbuf, recvcounts[i] and datatype.

Advice to implementors. The MPI_REDUCE_SCATTER routine is functionally equivalent to: an MPI_REDUCE collective operation with count equal to the sum of recvcounts[i] followed by MPI_SCATTERV with sendcounts equal to recvcounts. However, a direct implementation may run faster. (*End of advice to implementors.*)

The "in place" option for 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)

				36
II	N	sendbuf	starting address of send buffer (choice)	37
C	DUT	recvbuf	starting address of receive buffer (choice)	38
II	N	count	number of elements in input buffer (non-negative	39
			integer)	40
П	N	datatype	datatype of elements of input buffer (handle)	41
П	N	ор	operation (handle)	42
		- F	\mathbf{r}	43
II	N	comm	communicator (handle)	44
				45

C binding

Unofficial Draft for Comment Only

 $\mathbf{5}$

 24

```
1
     int MPI_Scan_c(const void *sendbuf, void *recvbuf, MPI_Count count,
\mathbf{2}
                   MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
3
     Fortran 2008 binding
4
     MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
5
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
6
         TYPE(*), DIMENSION(...) :: recvbuf
7
         INTEGER, INTENT(IN) :: count
8
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
9
         TYPE(MPI_Op), INTENT(IN) :: op
10
         TYPE(MPI_Comm), INTENT(IN) :: comm
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13
     MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
14
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
15
         TYPE(*), DIMENSION(...) :: recvbuf
16
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
17
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
18
         TYPE(MPI_Op), INTENT(IN) :: op
19
         TYPE(MPI_Comm), INTENT(IN) :: comm
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
     Fortran binding
22
     MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
23
         <type> SENDBUF(*), RECVBUF(*)
24
         INTEGER COUNT, DATATYPE, OP, COMM, IERROR
25
```

²⁶ If comm is an intra-communicator, MPI_SCAN is used to perform a prefix reduction ²⁷ on data distributed across the group. The operation returns, in the receive buffer of the ²⁸ process with rank i, the reduction of the values in the send buffers of processes with ranks ²⁹ 0,...,i (inclusive). The routine is called by all group members using the same arguments ³⁰ for count, datatype, op and comm, except that for user-defined operations, the same rules ³¹ apply as for MPI_REDUCE. The type of operations supported, their semantics, and the ³² 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.

This operation is invalid for inter-communicators.

36

6.11.2	Exclusive Scan		1
			2 3
MPI_EX	SCAN(sendbuf, recvbuf, cou	nt, datatype, op, comm)	4
IN	sendbuf	starting address of send buffer (choice)	5
			6
OUT	recvbuf	starting address of receive buffer (choice)	7 8
IN	count	number of elements in input buffer (non-negative integer)	9 10
IN	datatype	datatype of elements of input buffer (handle)	10
IN	ор	operation (handle)	12
IN	comm	intra-communicator (handle)	13
IIN	comm	intra communicator (nantre)	14
C bind	ing		15
	0	dbuf, void *recvbuf, int count,	16
		atype, MPI_Op op, MPI_Comm comm)	17
int MD	Evecon clonet word to	endbuf, void *recvbuf, MPI_Count count,	18 19
IIIC PIF.		atype, MPI_Op op, MPI_Comm comm)	20
_			21
	n 2008 binding		22
	scan(sendbuf, recvbuf, c PE(*), DIMENSION(), IN	count, datatype, op, comm, ierror)	23
	PE(*), DIMENSION() ::		24
	TEGER, INTENT(IN) :: cou		25
	PE(MPI_Datatype), INTENT		26
	PE(MPI_Op), INTENT(IN) :		27 28
	PE(MPI_Comm), INTENT(IN)		20
IN	TEGER, OPTIONAL, INTENT(OUT) :: ierror	30
MPI_Ex:	scan(sendbuf, recvbuf, c	count, datatype, op, comm, ierror)	31
	PE(*), DIMENSION(), IN		32
TYI	<pre>PE(*), DIMENSION() ::</pre>	recvbuf	33
	TEGER(KIND=MPI_COUNT_KIN		34
	PE(MPI_Datatype), INTENT	<i>v</i> 1	35
	PE(MPI_Op), INTENT(IN) :	-	36 37
	PE(MPI_Comm), INTENT(IN) FEGER, OPTIONAL, INTENT(38
			39
	n binding		40
		COUNT, DATATYPE, OP, COMM, IERROR)	41
•	/pe> SENDBUF(*), RECVBUF		42
TN	TEGER COUNT, DATATYPE, C	r, ourin, learur	43

44If comm is an intra-communicator, MPI_EXSCAN is used to perform a prefix reduction 45on data distributed across the group. The value in recvbuf on the process with rank 0 is 46undefined, 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 4748with rank i > 1, the operation returns, in the receive buffer of the process with rank i, the

Unofficial Draft for Comment Only

reduction of the values in the send buffers of processes with ranks $0, \ldots, i-1$ (inclusive). The $\mathbf{2}$ 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 non-invertable 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:

values	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
logicals	0	0	1	1	1	0	0	1
result	v_1	$v_1 + v_2$	v_3	$v_3 + v_4$	$v_3 + v_4 + v_5$	v_6	$v_6 + v_7$	v_8

The operator that produces this effect is

$$\left(\begin{array}{c} u\\i\end{array}\right)\circ\left(\begin{array}{c} v\\j\end{array}\right)=\left(\begin{array}{c} w\\j\end{array}\right)$$

$$w = \begin{cases} u+v & \text{if } i=j\\ v & \text{if } i\neq j \end{cases}.$$

Note that this is a non-commutative operator. C code that implements it is given below.

 $\overline{7}$

where

```
typedef struct {
    double val;
    int log;
} SegScanPair;
/* the user-defined function
*/
void segScan(SegScanPair *in, SegScanPair *inout, int *len,
             MPI_Datatype *dptr)
{
    int i;
    SegScanPair c;
    for (i=0; i< *len; ++i) {</pre>
        if (in->log == inout->log)
            c.val = in->val + inout->val;
        else
            c.val = inout->val;
        c.log = inout->log;
        *inout = c;
        in++; inout++;
    }
}
```

Note that the inout argument to the user-defined function corresponds to the righthand operand of the operator. When using this operator, we must be careful to specify that it is non-commutative, as in the following.

```
28
int i,base;
                                                                                   29
SegScanPair
             a, answer;
                                                                                   30
MPI_Op
               myOp;
MPI_Datatype type[2] = {MPI_DOUBLE, MPI_INT};
                                                                                   31
                                                                                   32
MPI_Aint
               disp[2];
                                                                                   33
               blocklen[2] = { 1, 1};
int
                                                                                   34
MPI_Datatype sspair;
                                                                                   35
                                                                                   36
/* explain to MPI how type SegScanPair is defined
                                                                                   37
 */
                                                                                   38
MPI_Get_address(&a, disp);
                                                                                   39
MPI_Get_address(&a.log, disp+1);
                                                                                   40
base = disp[0];
                                                                                   41
for (i=0; i<2; ++i) disp[i] -= base;</pre>
                                                                                   42
MPI_Type_create_struct(2, blocklen, disp, type, &sspair);
MPI_Type_commit(&sspair);
                                                                                   43
                                                                                   44
/* create the segmented-scan user-op
                                                                                   45
 */
                                                                                   46
MPI_Op_create(segScan, 0, &myOp);
                                                                                   47
 . . .
                                                                                   48
MPI_Scan(&a, &answer, 1, sspair, myOp, comm);
```

Unofficial Draft for Comment Only

1

2

3

45

6

9

10 11

12 13

14 15

16

17

18

19

20 21

22

23 24

25

26

1 2

7

6.12 Nonblocking Collective Operations

As described in Section 3.7, performance of many applications can be improved by over-3 lapping communication and computation, and many systems enable this. Nonblocking 4 collective operations combine the potential benefits of nonblocking point-to-point opera-5tions, to exploit overlap and to avoid synchronization, with the optimized implementation 6 and message scheduling provided by collective operations [34, 38]. One way of doing this 8 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 9 overheads, and thread management) is to use nonblocking collective communication [36]. 10

The nonblocking collective communication model is similar to the model used for non-11 blocking point-to-point communication. A nonblocking call initiates a collective operation, 12which must be completed in a separate completion call. Once initiated, the operation 13 may progress independently of any computation or other communication at participating 14processes. In this manner, nonblocking collective operations can mitigate possible synchro-15nizing effects of collective operations by running them in the "background." In addition to 16enabling communication-computation overlap, nonblocking collective operations can per-17form collective operations on overlapping communicators, which would lead to deadlocks 18 with blocking operations. Their semantic advantages can also be useful in combination with 19point-to-point communication. 20

As in the nonblocking point-to-point case, all calls are local and return immediately, 21irrespective of the status of other processes. The call initiates the operation, which indicates 22that the system may start to copy data out of the send buffer and into the receive buffer. 23Once initiated, all associated send buffers and buffers associated with input arguments (such 24 as arrays of counts, displacements, or datatypes in the vector versions of the collectives) 25should not be modified, and all associated receive buffers should not be accessed, until the 26collective operation completes. The call returns a request handle, which must be passed to 27a completion call. 28

All completion calls (e.g., MPI_WAIT) described in Section 3.7.3 are supported for 29 nonblocking collective operations. Similarly to the blocking case, nonblocking collective 30 operations are considered to be complete when the local part of the operation is finished, 31 i.e., for the caller, the semantics of the operation are guaranteed and all buffers can be 32 safely accessed and modified. Completion does not indicate that other processes have 33 completed or even started the operation (unless otherwise implied by the description of 34the operation). Completion of a particular nonblocking collective operation also does not 35 indicate completion of any other posted nonblocking collective (or send-receive) operations, 36 whether they are posted before or after the completed operation. 37

- 38
- 39 40 41

42

Advice to users. Users should be aware that implementations are allowed, but not required (with exception of MPI_IBARRIER), to synchronize processes during the completion of a nonblocking collective operation. (End of advice to users.)

Upon returning from a completion call in which a nonblocking collective operation 43 completes, the values of the MPI_SOURCE and MPI_TAG fields in the associated status object, 44 if any, are undefined. The value of MPI_ERROR may be defined, if appropriate, according 45to the specification in Section 3.2.5. It is valid to mix different request types (i.e., any 46 combination of collective requests, I/O requests, generalized requests, or point-to-point 47requests) in functions that enable multiple completions (e.g., MPI_WAITALL). It is erroneous 48

to call MPI_REQUEST_FREE or MPI_CANCEL for a request associated with a nonblocking collective operation. Nonblocking collective requests created using the APIs described in this section are not persistent. However, persistent collective requests can be created using persistent collective operations described in Sections 6.13 and 8.8.

Rationale. Freeing an active nonblocking collective request could cause similar problems as discussed for point-to-point requests (see Section 3.7.3). Cancelling a request is not supported because the semantics of this operation are not well-defined. (End of rationale.)

Multiple nonblocking collective operations can be outstanding on a single communicator. If the nonblocking call causes some system resource to be exhausted, then it will fail and raise an error. Quality implementations of MPI should ensure that this happens only in pathological cases. That is, an MPI implementation should be able to support a large number of pending nonblocking operations.

Unlike point-to-point operations, nonblocking collective operations do not match with blocking collective operations, and collective operations do not have a tag argument. All processes must call collective operations (blocking and nonblocking) in the same order per communicator. In particular, once a process calls a collective operation, all other processes in the communicator must eventually call the same collective operation, and no other collective operation with the same communicator in between. This is consistent with 20the ordering rules for blocking collective operations in threaded environments.

Matching blocking and nonblocking collective operations is not allowed Rationale. because the implementation might use different communication algorithms for the two cases. Blocking collective operations may be optimized for minimal time to completion, while nonblocking collective operations may balance time to completion with CPU overhead and asynchronous progression.

The use of tags for collective operations can prevent certain hardware optimizations. (End of rationale.)

If program semantics require matching blocking and nonblocking Advice to users. collective operations, then a nonblocking collective operation can be initiated and immediately completed with a blocking wait to emulate blocking behavior. (End of advice to users.)

In terms of data movement, each nonblocking collective operation has the same effect as its blocking counterpart for intra-communicators and inter-communicators after completion. Likewise, upon completion, nonblocking collective reduction operations have the same effect as their blocking counterparts, and the same restrictions and recommendations on reduction orders apply.

The use of the "in place" option is allowed exactly as described for the corresponding blocking collective operations. When using the "in place" option, message buffers function as both send and receive buffers. Such buffers should not be modified or accessed until the operation completes.

Progression rules for nonblocking collective operations are similar to progression of nonblocking point-to-point operations, refer to Section 3.7.4.

46Advice to implementors. Nonblocking collective operations can be implemented with 47local execution schedules [37] using nonblocking point-to-point communication and a 48 reserved tag-space. (End of advice to implementors.)

Unofficial Draft for Comment Only

1 $\mathbf{2}$

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

2122

23

24

25

26

27

28

29 30

 31

32

33

34

35

36

37

38

39

40

41

42

43

44

1 2	6.12.1	Nonblocking Barrier	Synchronization						
3									
4	MPI_IBA	RRIER(comm, reques	st)						
5	IN	comm	communicator (handle)						
6 7	OUT	request	communication request (handle)						
8	001	request	communication request (nandie)						
9	C bindi	ng							
10 11	int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)								
12	Fortran	2008 binding							
13		rrier(comm, reques							
14	TYPE(MPI_Comm), INTENT(IN) :: comm								
15		-	NTENT(OUT) :: request						
16		EGER, UPIIUNAL, II	NTENT(OUT) :: ierror						
17 18		binding							
19		RRIER(COMM, REQUES							
20	TNU	EGER COMM, REQUES	I, IERRUR						
21			blocking version of MPI_BARRIER. By calling MPI_IBARRIER,						
22	-		reached the barrier. The call returns immediately, indepen-						
23		*	sees have called MPI_IBARRIER. The usual barrier semantics						
24		-	ding completion operation (test or wait), which in the intra-						
25 26		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							
20		mote group have call							
28									
29			blocking barrier can be used to hide latency. Moving indepen-						
30			ween the MPI_IBARRIER and the subsequent completion call						
31		can overlap the barrier latency and therefore shorten possible waiting times. The se- mantic properties are also useful when mixing collective operations and point-to-point							
32		manue properties are also useful when mixing conective operations and point-to-point messages. (End of advice to users.)							
33 34									
35									
36									
37									
38									
39									
40									
41 42									
42									
44									
45									
46									
47									
48									

.12.2 1	Ionblocking Broadcast		
/IPI_IBCA	ST(buffer, count, datatype	e, root, comm, 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)	
OUT	request	communication request (handle)	
C bindin	G		
	0	int count, MPI_Datatype datatype, int root,	
		PI_Request *request)	
nt MPT	Theast c(void *huffer	, MPI_Count count, MPI_Datatype datatype,	
		<pre>mm comm, MPI_Request *request)</pre>	
ontron '			
	2008 binding		
	et(butter count det;	atura root comm request jerror)	
		atype, root, comm, request, ierror) SYNCHRONOUS : buffer	
TYPE	(*), DIMENSION(), AS	SYNCHRONOUS :: buffer	
TYPE INTE	(*), DIMENSION(), AS GER, INTENT(IN) :: con	SYNCHRONOUS :: buffer unt, root	
TYPE INTE TYPE	(*), DIMENSION(), AS	SYNCHRONOUS :: buffer unt, root F(IN) :: datatype	
TYPE INTE TYPE TYPE	(*), DIMENSION(), AS GER, INTENT(IN) :: cou (MPI_Datatype), INTEN	SYNCHRONOUS :: buffer unt, root Γ(IN) :: datatype) :: comm	
TYPE INTE TYPE TYPE TYPE	(*), DIMENSION(), AS GER, INTENT(IN) :: con (MPI_Datatype), INTEN (MPI_Comm), INTENT(IN)	SYNCHRONOUS :: buffer unt, root F(IN) :: datatype) :: comm (OUT) :: request	
TYPE INTE TYPE TYPE TYPE INTE	(*), DIMENSION(), AS GER, INTENT(IN) :: con (MPI_Datatype), INTENT (MPI_Comm), INTENT(IN) (MPI_Request), INTENT GER, OPTIONAL, INTENT	SYNCHRONOUS :: buffer unt, root F(IN) :: datatype) :: comm (OUT) :: request (OUT) :: ierror	
TYPE INTE TYPE TYPE TYPE INTE PI_Ibca	(*), DIMENSION(), AS GER, INTENT(IN) :: cou (MPI_Datatype), INTEN (MPI_Comm), INTENT(IN) (MPI_Request), INTENT GER, OPTIONAL, INTENT st(buffer, count, data	SYNCHRONOUS :: buffer unt, root F(IN) :: datatype) :: comm (OUT) :: request (OUT) :: ierror atype, root, comm, request, ierror)	
TYPE INTE TYPE TYPE INTE INTE PI_Ibca TYPE	<pre>(*), DIMENSION(), AS GER, INTENT(IN) :: con (MPI_Datatype), INTENT (MPI_Comm), INTENT(IN) (MPI_Request), INTENT GER, OPTIONAL, INTENT st(buffer, count, data (*), DIMENSION(), AS</pre>	SYNCHRONOUS :: buffer unt, root T(IN) :: datatype) :: comm (OUT) :: request (OUT) :: ierror atype, root, comm, request, ierror) SYNCHRONOUS :: buffer	
TYPE INTE TYPE TYPE INTE INTE PI_Ibca TYPE INTE	<pre>(*), DIMENSION(), AS GER, INTENT(IN) :: con (MPI_Datatype), INTENT (MPI_Comm), INTENT(IN) (MPI_Request), INTENT GER, OPTIONAL, INTENT st(buffer, count, data (*), DIMENSION(), AS GER(KIND=MPI_COUNT_KIN</pre>	SYNCHRONOUS :: buffer unt, root T(IN) :: datatype) :: comm (OUT) :: request (OUT) :: ierror atype, root, comm, request, ierror) SYNCHRONOUS :: buffer ND), INTENT(IN) :: count	
TYPE INTE TYPE TYPE INTE PI_Ibca TYPE INTE TYPE	<pre>(*), DIMENSION(), AS GER, INTENT(IN) :: con (MPI_Datatype), INTENT (MPI_Comm), INTENT(IN) (MPI_Request), INTENT GER, OPTIONAL, INTENT st(buffer, count, data (*), DIMENSION(), AS GER(KIND=MPI_COUNT_KIN (MPI_Datatype), INTENT</pre>	SYNCHRONOUS :: buffer unt, root T(IN) :: datatype) :: comm (OUT) :: request (OUT) :: ierror atype, root, comm, request, ierror) SYNCHRONOUS :: buffer ND), INTENT(IN) :: count T(IN) :: datatype	
TYPE INTE TYPE TYPE INTE PI_Ibca TYPE INTE TYPE INTE	<pre>(*), DIMENSION(), AS GER, INTENT(IN) :: con (MPI_Datatype), INTENT (MPI_Comm), INTENT(IN) (MPI_Request), INTENT GER, OPTIONAL, INTENT st(buffer, count, data (*), DIMENSION(), AS GER(KIND=MPI_COUNT_KIN (MPI_Datatype), INTENT GER, INTENT(IN) :: rook</pre>	SYNCHRONOUS :: buffer unt, root T(IN) :: datatype) :: comm (OUT) :: request (OUT) :: ierror atype, root, comm, request, ierror) SYNCHRONOUS :: buffer ND), INTENT(IN) :: count T(IN) :: datatype ot	
TYPE INTE TYPE TYPE INTE INTE INTE TYPE INTE TYPE	<pre>(*), DIMENSION(), AS GER, INTENT(IN) :: con (MPI_Datatype), INTENT (MPI_Comm), INTENT(IN) (MPI_Request), INTENT GER, OPTIONAL, INTENT st(buffer, count, data (*), DIMENSION(), AS GER(KIND=MPI_COUNT_KIN (MPI_Datatype), INTENT</pre>	SYNCHRONOUS :: buffer unt, root T(IN) :: datatype) :: comm (OUT) :: request (OUT) :: ierror atype, root, comm, request, ierror) SYNCHRONOUS :: buffer ND), INTENT(IN) :: count T(IN) :: datatype ot) :: comm	
TYPE INTE TYPE TYPE INTE INTE INTE TYPE INTE TYPE TYPE TYPE	<pre>(*), DIMENSION(), AS GER, INTENT(IN) :: con (MPI_Datatype), INTENT (MPI_Comm), INTENT(IN) (MPI_Request), INTENT GER, OPTIONAL, INTENT st(buffer, count, data (*), DIMENSION(), AS GER(KIND=MPI_COUNT_KIN (MPI_Datatype), INTENT GER, INTENT(IN) :: roo (MPI_Comm), INTENT(IN)</pre>	SYNCHRONOUS :: buffer unt, root T(IN) :: datatype) :: comm (OUT) :: request (OUT) :: ierror atype, root, comm, request, ierror) SYNCHRONOUS :: buffer ND), INTENT(IN) :: count T(IN) :: datatype ot) :: comm (OUT) :: request	
TYPE INTE TYPE TYPE INTE INTE INTE INTE TYPE INTE TYPE INTE	<pre>(*), DIMENSION(), AS GER, INTENT(IN) :: con (MPI_Datatype), INTENT (MPI_Comm), INTENT(IN) (MPI_Request), INTENT GER, OPTIONAL, INTENT st(buffer, count, data (*), DIMENSION(), AS GER(KIND=MPI_COUNT_KIN (MPI_Datatype), INTENT GER, INTENT(IN) :: roo (MPI_Comm), INTENT(IN) (MPI_Request), INTENT GER, OPTIONAL, INTENT</pre>	SYNCHRONOUS :: buffer unt, root T(IN) :: datatype) :: comm (OUT) :: request (OUT) :: ierror atype, root, comm, request, ierror) SYNCHRONOUS :: buffer ND), INTENT(IN) :: count T(IN) :: datatype ot) :: comm (OUT) :: request	
TYPE INTE TYPE TYPE INTE PI_Ibca TYPE INTE INTE TYPE INTE TYPE INTE	(*), DIMENSION(), AS GER, INTENT(IN) :: con (MPI_Datatype), INTENT (MPI_Comm), INTENT(IN) (MPI_Request), INTENT GER, OPTIONAL, INTENT st(buffer, count, data (*), DIMENSION(), AS GER(KIND=MPI_COUNT_KIN (MPI_Datatype), INTENT GER, INTENT(IN) :: roo (MPI_Comm), INTENT(IN) (MPI_Request), INTENT GER, OPTIONAL, INTENT binding	SYNCHRONOUS :: buffer unt, root T(IN) :: datatype) :: comm (OUT) :: request (OUT) :: ierror atype, root, comm, request, ierror) SYNCHRONOUS :: buffer ND), INTENT(IN) :: count T(IN) :: datatype ot) :: comm (OUT) :: request (OUT) :: ierror	
TYPE INTE TYPE TYPE INTE INTE INTE INTE TYPE INTE TYPE INTE TYPE INTE TYPE INTE	<pre>(*), DIMENSION(), AS GER, INTENT(IN) :: con (MPI_Datatype), INTENT (MPI_Comm), INTENT(IN) (MPI_Request), INTENT GER, OPTIONAL, INTENT st(buffer, count, data (*), DIMENSION(), AS GER(KIND=MPI_COUNT_KIN (MPI_Datatype), INTENT GER, INTENT(IN) :: roo (MPI_Comm), INTENT(IN) (MPI_Request), INTENT GER, OPTIONAL, INTENT binding ST(BUFFER, COUNT, DATA</pre>	SYNCHRONOUS :: buffer unt, root T(IN) :: datatype) :: comm (OUT) :: request (OUT) :: ierror atype, root, comm, request, ierror) SYNCHRONOUS :: buffer ND), INTENT(IN) :: count T(IN) :: datatype ot) :: comm (OUT) :: request	
TYPE INTE TYPE TYPE INTE INTE INTE TYPE INTE TYPE INTE INTE TYPE INTE SOTTRAN	<pre>(*), DIMENSION(), AS GER, INTENT(IN) :: con (MPI_Datatype), INTENT (MPI_Comm), INTENT(IN) (MPI_Request), INTENT GER, OPTIONAL, INTENT st(buffer, count, data (*), DIMENSION(), AS GER(KIND=MPI_COUNT_KIN (MPI_Datatype), INTENT GER, INTENT(IN) :: roo (MPI_Comm), INTENT(IN) GER, OPTIONAL, INTENT GER, OPTIONAL, INTENT binding ST(BUFFER, COUNT, DATA e> BUFFER(*)</pre>	SYNCHRONOUS :: buffer unt, root $\Gamma(IN) :: datatype$) :: comm (OUT) :: request (OUT) :: ierror atype, root, comm, request, ierror) SYNCHRONOUS :: buffer ND), INTENT(IN) :: count $\Gamma(IN)$:: datatype ot) :: comm (OUT) :: request (OUT) :: ierror ATYPE, ROOT, COMM, REQUEST, IERROR)	
TYPE INTE TYPE TYPE INTE INTE INTE TYPE INTE TYPE INTE INTE Ortran PI_IBCA	<pre>(*), DIMENSION(), AS GER, INTENT(IN) :: con (MPI_Datatype), INTENT (MPI_Comm), INTENT(IN) (MPI_Request), INTENT GER, OPTIONAL, INTENT st(buffer, count, data (*), DIMENSION(), AS GER(KIND=MPI_COUNT_KIN (MPI_Datatype), INTENT GER, INTENT(IN) :: roo (MPI_Comm), INTENT(IN) GER, OPTIONAL, INTENT GER, OPTIONAL, INTENT binding ST(BUFFER, COUNT, DATA e> BUFFER(*) GER COUNT, DATATYPE, H GER COUNT, DATATYPE, H</pre>	SYNCHRONOUS :: buffer unt, root T(IN) :: datatype) :: comm (OUT) :: request (OUT) :: ierror atype, root, comm, request, ierror) SYNCHRONOUS :: buffer ND), INTENT(IN) :: count T(IN) :: datatype ot) :: comm (OUT) :: request (OUT) :: ierror ATYPE, ROOT, COMM, REQUEST, IERROR) ROOT, COMM, REQUEST, IERROR	
TYPE INTE TYPE TYPE INTE INTE INTE TYPE INTE TYPE INTE INTE INTE INTE	<pre>(*), DIMENSION(), AS GER, INTENT(IN) :: con (MPI_Datatype), INTENT (MPI_Comm), INTENT(IN) (MPI_Request), INTENT GER, OPTIONAL, INTENT st(buffer, count, data (*), DIMENSION(), AS GER(KIND=MPI_COUNT_KIN (MPI_Datatype), INTENT GER, INTENT(IN) :: roo (MPI_Comm), INTENT(IN) GER, OPTIONAL, INTENT GER, OPTIONAL, INTENT binding ST(BUFFER, COUNT, DATA e> BUFFER(*) GER COUNT, DATATYPE, H GER COUNT, DATATYPE, H</pre>	SYNCHRONOUS :: buffer unt, root $\Gamma(IN) :: datatype$) :: comm (OUT) :: request (OUT) :: ierror atype, root, comm, request, ierror) SYNCHRONOUS :: buffer ND), INTENT(IN) :: count $\Gamma(IN)$:: datatype ot) :: comm (OUT) :: request (OUT) :: ierror ATYPE, ROOT, COMM, REQUEST, IERROR)	
TYPE INTE TYPE TYPE INTE INTE INTE TYPE INTE TYPE INTE INTE YPE INTE TYPE INTE TYPE INTE TYPE INTE TYPE INTE TYPE	<pre>(*), DIMENSION(), AS GER, INTENT(IN) :: con (MPI_Datatype), INTENT (MPI_Comm), INTENT(IN) (MPI_Request), INTENT GER, OPTIONAL, INTENT st(buffer, count, data (*), DIMENSION(), AS GER(KIND=MPI_COUNT_KIN (MPI_Datatype), INTENT GER, INTENT(IN) :: roo (MPI_Comm), INTENT(IN) (MPI_Request), INTENT GER, OPTIONAL, INTENT GER, OPTIONAL, INTENT binding ST(BUFFER, COUNT, DATA e> BUFFER(*) GER COUNT, DATATYPE, H call starts a nonblocking of the starts a nonblocking of the start</pre>	SYNCHRONOUS :: buffer unt, root T(IN) :: datatype) :: comm (OUT) :: request (OUT) :: ierror atype, root, comm, request, ierror) SYNCHRONOUS :: buffer ND), INTENT(IN) :: count T(IN) :: datatype ot) :: comm (OUT) :: request (OUT) :: ierror ATYPE, ROOT, COMM, REQUEST, IERROR) ROOT, COMM, REQUEST, IERROR	
TYPE INTE TYPE TYPE INTE INTE INTE TYPE INTE TYPE INTE INTE YOTTAN PI_IBCA Styp INTE	<pre>(*), DIMENSION(), AS GER, INTENT(IN) :: con (MPI_Datatype), INTENT (MPI_Comm), INTENT(IN) (MPI_Request), INTENT GER, OPTIONAL, INTENT st(buffer, count, data (*), DIMENSION(), AS GER(KIND=MPI_COUNT_KIN (MPI_Datatype), INTENT GER, INTENT(IN) :: roo (MPI_Comm), INTENT(IN) GER, OPTIONAL, INTENT GER, OPTIONAL, INTENT binding ST(BUFFER, COUNT, DATA e> BUFFER(*) GER COUNT, DATATYPE, H GER COUNT, DATATYPE, H</pre>	SYNCHRONOUS :: buffer unt, root T(IN) :: datatype) :: comm (OUT) :: request (OUT) :: ierror atype, root, comm, request, ierror) SYNCHRONOUS :: buffer ND), INTENT(IN) :: count T(IN) :: datatype ot) :: comm (OUT) :: request (OUT) :: ierror ATYPE, ROOT, COMM, REQUEST, IERROR) ROOT, COMM, REQUEST, IERROR	
TYPE INTE TYPE TYPE INTE INTE INTE TYPE INTE TYPE INTE INTE YOTTAN PI_IBCA <typ INTE This</typ 	<pre>(*), DIMENSION(), AS GER, INTENT(IN) :: con (MPI_Datatype), INTENT (MPI_Comm), INTENT(IN) (MPI_Request), INTENT GER, OPTIONAL, INTENT st(buffer, count, data (*), DIMENSION(), AS GER(KIND=MPI_COUNT_KIN (MPI_Datatype), INTENT GER, INTENT(IN) :: roo (MPI_Comm), INTENT(IN) (MPI_Request), INTENT GER, OPTIONAL, INTENT GER, OPTIONAL, INTENT binding ST(BUFFER, COUNT, DATA e> BUFFER(*) GER COUNT, DATATYPE, H call starts a nonblocking of the starts a nonblocking of the start</pre>	SYNCHRONOUS :: buffer unt, root T(IN) :: datatype) :: comm (OUT) :: request (OUT) :: ierror atype, root, comm, request, ierror) SYNCHRONOUS :: buffer ND), INTENT(IN) :: count T(IN) :: datatype ot) :: comm (OUT) :: request (OUT) :: ierror ATYPE, ROOT, COMM, REQUEST, IERROR) ROOT, COMM, REQUEST, IERROR variant of MPI_BCAST (see Section 6.4).	

1 **Example 6.24** Start a broadcast of 100 ints from process 0 to every process in the $\mathbf{2}$ group, perform some computation on independent data, and then complete the outstanding 3 broadcast operation. 4 MPI_Comm comm; 5int array1[100], array2[100]; 6 int root=0; 7 8 MPI_Request req; 9 . . . MPI_Ibcast(array1, 100, MPI_INT, root, comm, &req); 10 11 compute(array2, 100); MPI_Wait(&req, MPI_STATUS_IGNORE); 1213 146.12.3 Nonblocking Gather 151617MPI_IGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, 18 request) 19IN sendbuf starting address of send buffer (choice) 20IN number of elements in send buffer (non-negative 21sendcount 22 integer) 23IN datatype of send buffer elements (handle) sendtype 24 OUT recvbuf address of receive buffer (choice, significant only at 25root) 2627IN recvcount number of elements for any single receive (non-negative integer, significant only at root) 2829IN datatype of recv buffer elements (handle, significant recvtype 30 only at root) 31 IN rank of receiving process (integer) root 32 IN 33 communicator (handle) comm 34 OUT request communication request (handle) 35 36 C binding 37 int MPI_Igather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, 38 void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, 39 MPI_Comm comm, MPI_Request *request) 40 41 int MPI_Igather_c(const void *sendbuf, MPI_Count sendcount, 42MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount, 43 MPI_Datatype recvtype, int root, MPI_Comm comm, 44 MPI_Request *request) 45Fortran 2008 binding 46 MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 47root, comm, request, ierror)

TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf INTEGER, INTENT(IN) :: sendcount, recvcount, root TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, request, ierror) TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN) :: root TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_IGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST, 24 IERROR This call starts a nonblocking variant of MPI_GATHER (see Section 6.5).

12	MPI_IGAT	HERV(sendbuf, sendo comm, request)	count, sendtype, recvbuf, recvcounts, displs, recvtype, root,
3	IN	sendbuf	starting address of send buffer (choice)
4 5 6	IN	sendcount	number of elements in send buffer (non-negative integer)
7	IN	sendtype	datatype of send buffer elements (handle)
8 9 10	OUT	recvbuf	address of receive buffer (choice, significant only at root)
10 11 12 13	IN	recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process (significant only at root)
14 15 16 17	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)
18 19 20	IN	recvtype	datatype of recv buffer elements (handle, significant only at root)
21	IN	root	rank of receiving process (integer)
22	IN	comm	communicator (handle)
23 24	OUT	request	communication request (handle)
25 26 27 28 29 30	C bindin int MPI_3	Igatherv(const voi void *recvbu:	<pre>.d *sendbuf, int sendcount, MPI_Datatype sendtype, f, const int recvcounts[], const int displs[], recvtype, int root, MPI_Comm comm, *request)</pre>
31 32	int MPI_1	-	void *sendbuf, MPI_Count sendcount, sendtype, void *recvbuf,
33			unt recvcounts[], const MPI_Aint displs[],
34			recvtype, int root, MPI_Comm comm,
35 36		MPI_Request >	*request)
37		2008 binding	
38	MPI_Igath		count, sendtype, recvbuf, recvcounts, displs,
39	TVDE		ot, comm, request, ierror) , INTENT(IN), ASYNCHRONOUS :: sendbuf
40		GER, INTENT(IN) ::	
41 42		-	ITENT(IN) :: sendtype, recvtype
43			, ASYNCHRONOUS :: recvbuf
44			SYNCHRONOUS :: recvcounts(*), displs(*)
45		(MPI_Comm), INTENT	'(IN) :: comm 'ENT(OUT) :: request
46		_	ENT(OUT) :: request ENT(OUT) :: ierror
47 48		,,,,	

```
MPI_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                           2
               recvtype, root, comm, request, ierror)
    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(*)
                                                                                           9
    INTEGER, INTENT(IN) :: root
                                                                                           10
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                           11
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                           12
                                                                                           13
Fortran binding
                                                                                           14
MPI_IGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
                                                                                           15
               RECVTYPE, ROOT, COMM, REQUEST, IERROR)
                                                                                           16
    <type> SENDBUF(*), RECVBUF(*)
                                                                                           17
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
                                                                                           18
                COMM, REQUEST, IERROR
                                                                                           19
    This call starts a nonblocking variant of MPI_GATHERV (see Section 6.5).
                                                                                           20
                                                                                           21
                                                                                           22
6.12.4 Nonblocking Scatter
                                                                                           23
                                                                                           ^{24}
                                                                                           25
MPI_ISCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm,
                                                                                           26
               request)
                                                                                           27
  IN
           sendbuf
                                        address of send buffer (choice, significant only at
                                                                                           28
                                        root)
                                                                                           29
                                                                                           30
  IN
            sendcount
                                        number of elements sent to each process
                                                                                           31
                                        (non-negative integer, significant only at root)
                                                                                           32
  IN
           sendtype
                                        datatype of send buffer elements (handle, significant
                                                                                           33
                                        only at root)
                                                                                           34
  OUT
            recvbuf
                                        address of receive buffer (choice)
                                                                                           35
                                                                                           36
  IN
                                        number of elements in receive buffer (non-negative
            recvcount
                                                                                           37
                                        integer)
                                                                                           38
  IN
                                        datatype of receive buffer elements (handle)
            recvtype
                                                                                           39
  IN
                                        rank of sending process (integer)
            root
                                                                                           40
                                                                                           41
  IN
                                        communicator (handle)
           comm
                                                                                           42
  OUT
            request
                                        communication request (handle)
                                                                                           43
                                                                                           44
C binding
```

```
1
     int MPI_Iscatter_c(const void *sendbuf, MPI_Count sendcount,
\mathbf{2}
                   MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
3
                   MPI_Datatype recvtype, int root, MPI_Comm comm,
4
                   MPI_Request *request)
5
     Fortran 2008 binding
6
     MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
7
                   root, comm, request, ierror)
8
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
9
         INTEGER, INTENT(IN) :: sendcount, recvcount, root
10
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
11
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
12
         TYPE(MPI_Comm), INTENT(IN) :: comm
13
         TYPE(MPI_Request), INTENT(OUT) :: request
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
17
                   root, comm, request, ierror)
18
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
19
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
20
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
21
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
22
         INTEGER, INTENT(IN) :: root
23
         TYPE(MPI_Comm), INTENT(IN) :: comm
24
         TYPE(MPI_Request), INTENT(OUT) :: request
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
     Fortran binding
27
     MPI_ISCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
28
                   ROOT, COMM, REQUEST, IERROR)
29
         <type> SENDBUF(*), RECVBUF(*)
30
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,
31
                    IERROR
32
33
         This call starts a nonblocking variant of MPI_SCATTER (see Section 6.6).
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

MPI_ISCAT	TTERV(sendbuf, sendcounts, d comm, request)	ispls, sendtype, recvbuf, recvcount, recvtype, root,	1 2
IN	sendbuf	address of send buffer (choice, significant only at root)	3 4 5
IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank (significant only at root)	5 6 7 8
IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to sendbuf) from which to take the outgoing data to process i (significant only at root)	9 10 11 12
IN	sendtype	datatype of send buffer elements (handle, significant only at root)	13 14 15
OUT	recvbuf	address of receive buffer (choice)	16
IN	recvcount	number of elements in receive buffer (non-negative integer)	17 18
IN	recvtype	datatype of receive buffer elements (handle)	19
IN	root	rank of sending process (integer)	20 21
IN	comm	communicator (handle)	21
OUT	request	communication request (handle)	23
001	request	communication request (nancie)	24
C binding	r		25
-		dbuf, const int sendcounts[],	26 27
	const int displs[],	MPI_Datatype sendtype, void *recvbuf,	21
	int recvcount, MPI_D MPI_Request *request	<pre>atatype recvtype, int root, MPI_Comm comm,)</pre>	29 30
int MPT T	scattery c(const void *se	endbuf, const MPI_Count sendcounts[],	31
		s[], MPI_Datatype sendtype, void *recvbuf,	32
		MPI_Datatype recvtype, int root,	33
	MPI_Comm comm, MPI_R	equest *request)	34
Fortran 2	008 binding		35
	<u> </u>	, displs, sendtype, recvbuf, recvcount,	36
	recvtype, root, comm		37 38
TYPE(*), DIMENSION(), INTENT	Γ(IN), ASYNCHRONOUS :: sendbuf	39
		NOUS :: sendcounts(*), displs(*)	40
	MPI_Datatype), INTENT(IN)		41
	*), DIMENSION(), ASYNCH		42
	ER, INTENT(IN) :: recvcou MPI_Comm), INTENT(IN) ::	-	43
	MPI_Request), INTENT(OUT)		44
	ER, OPTIONAL, INTENT(OUT)	-	45 46
			46 47
nri_iscat	recvtype, root, comm	, displs, sendtype, recvbuf, recvcount, , request, ierror)	48

1 2 3 4 5 6 7 8 9	INT INT TYP INT INT TYP TYP	EGER(KIND=MPI_CO EGER(KIND=MPI_AD E(MPI_Datatype), E(*), DIMENSION(EGER(KIND=MPI_CO EGER, INTENT(IN) E(MPI_Comm), INT E(MPI_Request),	
11			
12 13 14 15 16 17	MPI_ISC	RECVTYPE, pe> SENDBUF(*), EGER SENDCOUNTS(SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, ROOT, COMM, REQUEST, IERROR) RECVBUF(*) *), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, UEST, IERROR
18	This	s call starts a nonbl	ocking variant of $MPI_SCATTERV$ (see Section 6.6).
19			
20	6.12.5	Nonblocking Gathe	er-to-all
21 22			
22 23 24	MPI_IAL	LGATHER(sendbuf, request)	sendcount, sendtype, recvbuf, recvcount, recvtype, comm,
25 26	IN	sendbuf	starting address of send buffer (choice)
27	IN	sendcount	number of elements in send buffer (non-negative
28 29	IIN	senacount	integer)
30	IN	sendtype	datatype of send buffer elements (handle)
31	OUT	recvbuf	address of receive buffer (choice)
32 33	IN	recvcount	number of elements received from any process (non-negative integer)
34	IN	recvtype	datatype of receive buffer elements (handle)
35 36	IN	comm	communicator (handle)
37	OUT	request	communication request (handle)
38	001	request	communication request (narrate)
39 40 41 42 43	C bindi int MPI	_Iallgather(cons MPI_Dataty	t void *sendbuf, int sendcount, pe sendtype, void *recvbuf, int recvcount, pe recvtype, MPI_Comm comm, MPI_Request *request)
44 45 46 47 48	int MPI	MPI_Dataty	nst void *sendbuf, MPI_Count sendcount, pe sendtype, void *recvbuf, MPI_Count recvcount, pe recvtype, MPI_Comm comm, MPI_Request *request)

```
Fortran 2008 binding
                                                                                      \mathbf{2}
MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
              comm, request, ierror)
    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
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                      10
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                      11
MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                      12
              comm, request, ierror)
                                                                                      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
Fortran binding
MPI_IALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
                                                                                      22
                                                                                      23
              COMM, REQUEST, IERROR)
                                                                                      ^{24}
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
                                                                                      25
                                                                                      26
    This call starts a nonblocking variant of MPI_ALLGATHER (see Section 6.7).
                                                                                      27
                                                                                      28
                                                                                      29
                                                                                      30
                                                                                      31
                                                                                      32
                                                                                      33
                                                                                      34
                                                                                      35
                                                                                      36
                                                                                      37
                                                                                      38
                                                                                      39
                                                                                      40
                                                                                      41
                                                                                      42
                                                                                      43
                                                                                      44
                                                                                      45
                                                                                      46
                                                                                      47
                                                                                      48
```

1 2	MPI_IALLO	GATHERV(sendbuf, sendcount, comm, request)	sendtype, recvbuf, recvcounts, displs, recvtype,
$\frac{3}{4}$	IN	sendbuf	starting address of send buffer (choice)
5	IN	sendcount	number of elements in send buffer (non-negative integer)
7	IN	sendtype	datatype of send buffer elements (handle)
8	OUT	recvbuf	address of receive buffer (choice)
9 10 11 12	IN	recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process
13 14 15	IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i
16 17	IN	recvtype	datatype of receive buffer elements (handle)
18	IN	comm	communicator (handle)
19	OUT	request	communication request (handle)
20 21 22 23 24 25 26 27 28		allgatherv(const void *se MPI_Datatype sendtyp const int displs[], MPI_Request *request	e, void *recvbuf, const int recvcounts[], MPI_Datatype recvtype, MPI_Comm comm,) *sendbuf, MPI_Count sendcount,
29 30 31		const MPI_Count recv MPI_Datatype recvtyp	<pre>counts[], const MPI_Aint displs[], e, MPI_Comm comm, MPI_Request *request)</pre>
32 33 34 35 36 37 38 39 40 41 42 43	MPI_Iallg TYPE(INTEG TYPE(TYPE(INTEG TYPE(INTEG	recvtype, comm, requ *), DIMENSION(), INTENT ER, INTENT(IN) :: sendcou MPI_Datatype), INTENT(IN) *), DIMENSION(), ASYNCH ER, INTENT(IN), ASYNCHROM MPI_Comm), INTENT(IN) :: MPI_Request), INTENT(OUT) ER, OPTIONAL, INTENT(OUT) atherv(sendbuf, sendcount	<pre>T(IN), ASYNCHRONOUS :: sendbuf int) :: sendtype, recvtype HRONOUS :: recvbuf NOUS :: recvcounts(*), displs(*) comm) :: request) :: ierror t, sendtype, recvbuf, recvcounts, displs,</pre>
44 45 46 47 48	INTEG TYPE(<pre>recvtype, comm, requ *), DIMENSION(), INTENT ER(KIND=MPI_COUNT_KIND), MPI_Datatype), INTENT(IN) *), DIMENSION(), ASYNCH</pre>	Γ(IN), ASYNCHRONOUS :: sendbuf INTENT(IN) :: sendcount) :: sendtype, recvtype

INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*) 1 $\mathbf{2}$ 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 6 Fortran binding MPI_IALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) 10 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, 11 REQUEST, IERROR 12This call starts a nonblocking variant of MPI_ALLGATHERV (see Section 6.7). 13 14156.12.6 Nonblocking All-to-All Scatter/Gather 1617 18 MPI_IALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request) 19 20IN sendbuf starting address of send buffer (choice) 2122 IN sendcount number of elements sent to each process 23(non-negative integer) 24datatype of send buffer elements (handle) IN sendtype 25OUT recvbuf address of receive buffer (choice) 2627IN number of elements received from any process recvcount 28(non-negative integer) 29 IN recvtype datatype of receive buffer elements (handle) 30 IN communicator (handle) comm 3132 OUT request communication request (handle) 33 34 C binding 35int MPI_Ialltoall(const void *sendbuf, int sendcount, 36 MPI_Datatype sendtype, void *recvbuf, int recvcount, 37 MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) 38 int MPI_Ialltoall_c(const void *sendbuf, MPI_Count sendcount, 39 MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount, 40 MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) 41 42Fortran 2008 binding 43 MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 44 comm, request, ierror) 45TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 46INTEGER, INTENT(IN) :: sendcount, recvcount 47TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 48

1 2	TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf TYPE(MPI_Comm), INTENT(IN) :: comm
3	TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5 6 7	<pre>MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,</pre>
8 9	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
10 11	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf TYPE(MPI_Comm), INTENT(IN) :: comm
12 13 14	TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15 16 17 18	<pre>Fortran binding MPI_IALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,</pre>
19 20	INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
21 22	This call starts a nonblocking variant of $MPI_ALLTOALL$ (see Section 6.8).
23 24	
25 26	
27 28 29	
30 31	
32 33	
34 35	
36 37 38	
39 40	
40 41 42	
42 43 44	
44 45 46	
47 48	

MPI_I	ALLTOALLV(sendbuf, ser recvtype, comn	ndcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, n, request)	1 2
IN	sendbuf	starting address of send buffer (choice)	3
IN	sendcounts	non-negative integer array (of length group size)	4 5
		specifying the number of elements to send to each	6
		rank	7
IN	sdispls	integer array (of length group size). Entry j specifies	8
		the displacement (relative to $sendbuf)$ from which to	9
		take the outgoing data destined for process j	10 11
IN	sendtype	datatype of send buffer elements (handle)	12
OUT	Г recvbuf	address of receive buffer (choice)	13
IN	recvcounts	non-negative integer array (of length group size)	14
		specifying the number of elements that can be	15
		received from each rank	16
IN	rdispls	integer array (of length group size). Entry i specifies	17 18
		the displacement (relative to recvbuf) at which to	19
		place the incoming data from process i	20
IN	recvtype	datatype of receive buffer elements (handle)	21
IN	comm	communicator (handle)	22
001	request	communication request (handle)	23 24
			24 25
	nding		26
int M		void *sendbuf, const int sendcounts[],	27
		<pre>ispls[], MPI_Datatype sendtype, void *recvbuf, cvcounts[], const int rdispls[],</pre>	28
		recvtype, MPI_Comm comm, MPI_Request *request)	29
N			30 31
int M		<pre>t void *sendbuf, const MPI_Count sendcounts[], nt sdispls[], MPI_Datatype sendtype,</pre>	32
		f, const MPI_Count recvcounts[],	33
		<pre>nt rdispls[], MPI_Datatype recvtype,</pre>	34
	MPI_Comm com	m, MPI_Request *request)	35
Fortr	an 2008 binding		36 37
	U	endcounts, sdispls, sendtype, recvbuf, recvcounts,	38
	rdispls, rec	vtype, comm, request, ierror)	39
), INTENT(IN), ASYNCHRONOUS :: sendbuf	40
I		ASYNCHRONOUS :: sendcounts(*), sdispls(*),	41
т		*), rdispls(*) NTENT(IN) :: sendtype, recvtype	42
), ASYNCHRONOUS :: recvbuf	43
	YPE(MPI_Comm), INTEN		44 45
	YPE(MPI_Request), IN	-	46
I	NTEGER, OPTIONAL, IN	TENT(OUT) :: ierror	47
			48

```
1
     MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
\mathbf{2}
                    rdispls, recvtype, comm, request, ierror)
3
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
4
5
                    sendcounts(*), recvcounts(*)
6
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
7
                    rdispls(*)
8
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
9
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
10
         TYPE(MPI_Comm), INTENT(IN) :: comm
11
         TYPE(MPI_Request), INTENT(OUT) :: request
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
     Fortran binding
14
     MPI_IALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,
15
                    RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)
16
         <type> SENDBUF(*), RECVBUF(*)
17
         INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
18
                    RECVTYPE, COMM, REQUEST, IERROR
19
         This call starts a nonblocking variant of MPI_ALLTOALLV (see Section 6.8).
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_IALL1	FOALLW(sendbuf, sendcounts, recvtypes, comm, request	sdispls, sendtypes, recvbuf, recvcounts, rdispls,)	1 2
IN	sendbuf	starting address of send buffer (choice)	3
IN	sendcounts	integer array (of length group size) specifying the number of elements to send to each rank (array of non-negative integers)	4 5 6 7
IN	sdispls	integer array (of length group size). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for process j (array of integers)	8 9 10 11
IN	sendtypes	array of datatypes (of length group size). Entry j specifies the type of data to send to process j (array of handles)	12 13 14 15
OUT	recvbuf	address of receive buffer (choice)	16
IN	recvcounts	integer array (of length group size) specifying the number of elements that can be received from each rank (array of non-negative integers)	17 18 19
IN	rdispls	integer array (of length group size). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from process i (array of integers)	20 21 22 23 24
IN	recvtypes	array of datatypes (of length group size). Entry i specifies the type of data received from process i (array of handles)	25 26 27
IN	comm	communicator (handle)	28
OUT	request	communication request (handle)	29 30
C binding			31
		ndbuf, const int sendcounts[],	32 33
	<pre>const int sdispls[],</pre>	<pre>const MPI_Datatype sendtypes[],</pre>	34
		<pre>int recvcounts[], const int rdispls[],</pre>	35
		ecvtypes[], MPI_Comm comm,	36
	MPI_Request *request)	37
int MPI_I		sendbuf, const MPI_Count sendcounts[],	38 39
	-	<pre>ls[], const MPI_Datatype sendtypes[], NDI Growt encounts[]</pre>	40
		<pre>MPI_Count recvcounts[], ls[], const MPI_Datatype recvtypes[],</pre>	41
	MPI_Comm comm, MPI_R		42
			43
	008 binding	s, sdispls, sendtypes, recvbuf,	44
III I_IAIIU		recvtypes, comm, request, ierror)	45
TYPE(_	C(IN), ASYNCHRONOUS :: sendbuf	46 47
			48

```
1
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
2
                    recvcounts(*), rdispls(*)
3
         TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
4
                    recvtypes(*)
5
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
6
         TYPE(MPI_Comm), INTENT(IN) :: comm
7
         TYPE(MPI_Request), INTENT(OUT) :: request
8
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
     MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
10
                   recvcounts, rdispls, recvtypes, comm, request, ierror)
11
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
12
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
13
                    sendcounts(*), recvcounts(*)
14
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
15
                    rdispls(*)
16
         TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
17
                    recvtypes(*)
18
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
19
         TYPE(MPI_Comm), INTENT(IN) :: comm
20
         TYPE(MPI_Request), INTENT(OUT) :: request
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
     Fortran binding
24
     MPI_IALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
25
                   RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)
26
         <type> SENDBUF(*), RECVBUF(*)
27
         INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
28
                    RDISPLS(*), RECVTYPES(*), COMM, REQUEST, IERROR
29
         This call starts a nonblocking variant of MPI_ALLTOALLW (see Section 6.8).
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

6.12.7 Nonblocking Reduce

6.12.	Nonblocking Reduce		1 2 3
MPI_	IREDUCE(sendbuf, recvbu	f, count, datatype, op, root, comm, request)	4
IN	sendbuf	address of send buffer (choice)	5 6
OU ⁻	Г recvbuf	address of receive buffer (choice, significant only at root)	7 8
IN	count	number of elements in send buffer (non-negative integer)	9 10
IN	datatype	datatype of elements of send buffer (handle)	11 12
IN	ор	reduce operation (handle)	13
IN	root	rank of root process (integer)	14
IN	comm	communicator (handle)	15 16
OU ⁻	Г request	communication request (handle)	17
			18
	nding		19 20
int r		<pre>1 *sendbuf, void *recvbuf, int count, datatype, MPI_Op op, int root, MPI_Comm comm,</pre>	21
	MPI_Request		22
int N	IPI Treduce c(const vo	oid *sendbuf, void *recvbuf, MPI_Count count,	23
1110 1		datatype, MPI_Op op, int root, MPI_Comm comm,	24 25
	MPI_Request	*request)	26
Forti	an 2008 binding		27
MPI_]		ouf, count, datatype, op, root, comm, request,	28
-	ierror)		29 30
), INTENT(IN), ASYNCHRONOUS :: sendbuf), ASYNCHRONOUS :: recvbuf	31
	INTEGER, INTENT(IN) :		32
]	CYPE(MPI_Datatype), IN	NTENT(IN) :: datatype	33
	YPE(MPI_Op), INTENT(-	34 35
	YPE(MPI_Comm), INTEN YPE(MPI_Request), IN		36
	INTEGER, OPTIONAL, INT		37
		ouf, count, datatype, op, root, comm, request,	38
III I _ I	ierror)	our, count, datatype, op, root, comm, request,	39 40
1	YPE(*), DIMENSION()), INTENT(IN), ASYNCHRONOUS :: sendbuf	41
), ASYNCHRONOUS :: recvbuf	42
		<pre>[Lind), INTENT(IN) :: count</pre>	43
	YPE(MPI_Datatype), II YPE(MPI_Op), INTENT(I	NTENT(IN) :: datatype IN) :: op	44 45
	INTEGER, INTENT(IN) :	-	45 46
	TYPE(MPI_Comm), INTEN		47
]	YPE(MPI_Request), INT	TENT(OUT) :: request	48

Unofficial Draft for Comment Only

1	тытт		
2		GER, UPIIUNAL, .	INTENT(OUT) :: ierror
3	Fortran	•	
4	MPI_IREI		CVBUF, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST,
5		IERROR)	
6	• 1	De> SENDBUF(*), H	RECVBOR(*) TYPE, OP, ROOT, COMM, REQUEST, IERROR
7			
8 9	This	call starts a nonble	ocking variant of MPI_REDUCE (see Section 6.9.1).
10	Adt	vice to implementor	rs. The implementation is explicitly allowed to use different
11	0		g and nonblocking reduction operations that might change the
12			the operations. However, as for MPI_REDUCE, it is strongly
13			I_IREDUCE be implemented so that the same result be obtained is applied on the same arguments, appearing in the same order.
14			event optimizations that take advantage of the physical location
15 16			advice to implementors.)
17	-	、 · ·	
18			operations which are not truly associative, the result delivered e nonblocking reduction may not exactly equal the result deliv-
19			eduction, even when specifying the same arguments in the same
20		er. (End of advice	,
21			
22 23	6.12.8	Nonblocking All-Re	educe
24			
25			
26	MPI_IALI	_REDUCE(sendbuf,	recvbuf, count, datatype, op, comm, request)
27	IN	sendbuf	starting address of send buffer (choice)
28 29	OUT	recvbuf	starting address of receive buffer (choice)
30	IN	count	number of elements in send buffer (non-negative
31			integer)
32	IN	datatype	datatype of elements of send buffer (handle)
33	IN	ор	operation (handle)
34 35	IN	comm	communicator (handle)
36	OUT	request	communication request (handle)
37			
38	C bindi	ng	
39	int MPI_		t void *sendbuf, void *recvbuf, int count,
40		•	pe datatype, MPI_Op op, MPI_Comm comm,
41 42		MP1_Reques	t *request)
42	int MPI_	Iallreduce_c(com	nst void *sendbuf, void *recvbuf, MPI_Count count,
44		•	pe datatype, MPI_Op op, MPI_Comm comm,
45		MPI_Reques	t *request)
46	Fortran	2008 binding	
47	MPI_Iall		recvbuf, count, datatype, op, comm, request,
48		ierror)	

1 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 2 INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Op), INTENT(IN) :: op 5 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, 10 ierror) 11 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 12TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 13 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count 14TYPE(MPI_Datatype), INTENT(IN) :: datatype 15TYPE(MPI_Op), INTENT(IN) :: op 16TYPE(MPI_Comm), INTENT(IN) :: comm 17 TYPE(MPI_Request), INTENT(OUT) :: request 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 19 20Fortran binding 21MPI_IALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, 22 IERROR) 23<type> SENDBUF(*), RECVBUF(*) 24 INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR 25This call starts a nonblocking variant of MPI_ALLREDUCE (see Section 6.9.6). 2627Nonblocking Reduce-Scatter with Equal Blocks 6.12.9 28 29 30 MPI_IREDUCE_SCATTER_BLOCK(sendbuf, recvbuf, recvcount, datatype, op, comm, 31request) 32 33 IN sendbuf starting address of send buffer (choice) 34 OUT recvbuf starting address of receive buffer (choice) 35IN recvcount element count per block (non-negative integer) 36 37 IN datatype datatype of elements of send and receive buffers 38 (handle) 39 IN op operation (handle) 40 41 IN communicator (handle) comm 42OUT communication request (handle) request 43 44

C binding

Unofficial Draft for Comment Only

45

46

47

```
1
     int MPI_Ireduce_scatter_block_c(const void *sendbuf, void *recvbuf,
\mathbf{2}
                   MPI_Count recvcount, MPI_Datatype datatype, MPI_Op op,
3
                   MPI_Comm comm, MPI_Request *request)
4
     Fortran 2008 binding
5
     MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
6
                   request, ierror)
7
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
8
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
9
         INTEGER, INTENT(IN) :: recvcount
10
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
11
         TYPE(MPI_Op), INTENT(IN) :: op
12
         TYPE(MPI_Comm), INTENT(IN) :: comm
13
         TYPE(MPI_Request), INTENT(OUT) :: request
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
17
                   request, ierror)
18
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
19
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
20
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
21
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
22
         TYPE(MPI_Op), INTENT(IN) :: op
23
         TYPE(MPI_Comm), INTENT(IN) :: comm
24
         TYPE(MPI_Request), INTENT(OUT) :: request
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
     Fortran binding
27
     MPI_IREDUCE_SCATTER_BLOCK (SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
28
                   REQUEST, IERROR)
29
         <type> SENDBUF(*), RECVBUF(*)
30
         INTEGER RECVCOUNT, DATATYPE, OP, COMM, REQUEST, IERROR
31
32
         This call starts a nonblocking variant of MPI_REDUCE_SCATTER_BLOCK (see Sec-
33
     tion 6.10.1).
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

6.12.10 Nonblocking Reduce-Scatter

2 MPI_IREDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm, request) 5 IN sendbuf starting address of send buffer (choice) 6 OUT recvbuf starting address of receive buffer (choice) IN recvcounts non-negative integer array specifying the number of 9 elements in result distributed to each process. This 10 array must be identical on all calling processes. 11 datatype datatype of elements of input buffer (handle) IN 12IN operation (handle) ор 13 14IN comm communicator (handle) 15OUT communication request (handle) request 1617 C binding 18 int MPI_Ireduce_scatter(const void *sendbuf, void *recvbuf, 19 const int recvcounts[], MPI_Datatype datatype, MPI_Op op, 20MPI_Comm comm, MPI_Request *request) 2122 int MPI_Ireduce_scatter_c(const void *sendbuf, void *recvbuf, 23const MPI_Count recvcounts[], MPI_Datatype datatype, 24MPI_Op op, MPI_Comm comm, MPI_Request *request) 25Fortran 2008 binding 26MPI_Ireduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm, 27request, ierror) 28TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 29 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf 30 INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*) 31TYPE(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 35INTEGER, OPTIONAL, INTENT(OUT) :: ierror 36 37 MPI_Ireduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm, 38 request, ierror) 39 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 40 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 41 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*) 42TYPE(MPI_Datatype), INTENT(IN) :: datatype 43 TYPE(MPI_Op), INTENT(IN) :: op 44 TYPE(MPI_Comm), INTENT(IN) :: comm 45TYPE(MPI_Request), INTENT(OUT) :: request 46INTEGER, OPTIONAL, INTENT(OUT) :: ierror 47

1

REQUEST, IERROR) e> SENDBUF(*), RECVBUF(*	YBUF, RECVCOUNTS, DATATYPE, OP, COMM,) YPE, OP, COMM, REQUEST, IERROR
call starts a nonblocking vari	ant of MPI_REDUCE_SCATTER (see Section $6.10.2$).
Nonblocking Inclusive Scan	
N(sendbuf, recvbuf, count, da	atatype, op, comm, request)
sendbuf	starting address of send buffer (choice)
recvbuf	starting address of receive buffer (choice)
count	number of elements in input buffer (non-negative integer)
datatype	datatype of elements of input buffer (handle)
ор	operation (handle)
comm	communicator (handle)
request	communication request (handle)
<pre>Iscan(const void *sendbu MPI_Datatype dataty MPI_Request *reques Iscan_c(const void *send MPI_Datatype dataty MPI_Request *reques 2008 binding n(sendbuf, recvbuf, coun (*), DIMENSION(), INTE (*), DIMENSION(), ASYN GER, INTENT(IN) :: count (MPI_Datatype), INTENT(I (MPI_Op), INTENT(IN) :: (MPI_Comm), INTENT(IN) :: (MPI_Request), INTENT(OU GER, OPTIONAL, INTENT(OU n(sendbuf, recvbuf, coun (*), DIMENSION(), INTE (*), DIMENSION(), ASYN</pre>	<pre>buf, void *recvbuf, MPI_Count count, ype, MPI_Op op, MPI_Comm comm, st) t, datatype, op, comm, request, ierror) NT(IN), ASYNCHRONOUS :: sendbuf CHRONOUS :: recvbuf N) :: datatype op : comm T) :: request T) :: ierror t, datatype, op, comm, request, ierror) NT(IN), ASYNCHRONOUS :: sendbuf CHRONOUS :: recvbuf</pre>
	<pre>UCE_SCATTER(SENDBUF, REC REQUEST, IERROR) e> SENDBUF(*), RECVBUF(* GER RECVCOUNTS(*), DATAT call starts a nonblocking vari Nonblocking Inclusive Scan Nonblocking Inclusive Scan N(sendbuf, recvbuf, count, da sendbuf recvbuf count datatype op comm request g Iscan(const void *sendbu MPI_Datatype dataty MPI_Request *reques Iscan_c(const void *send MPI_Datatype dataty MPI_Request *reques 2008 binding n(sendbuf, recvbuf, coun (*), DIMENSION(), INTE (*), DIMENSION(), ASYN GER, INTENT(IN) :: count (MPI_Op), INTENT(IN) :: (MPI_Comm), INTENT(OU GER, OPTIONAL, INTENT(OU GER, OPTIONAL, INTENT(OU n(sendbuf, recvbuf, coun (*), DIMENSION(), INTE (*), DIMENSION(), INTE (MPI_Request), INTENT(OU GER, OPTIONAL, INTENT(OU GER, OPTIONAL, INTENT(OU n(sendbuf, recvbuf, coun (*), DIMENSION(), INTE</pre>

```
TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                        1
                                                                                        2
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
                                                                                        5
MPI_ISCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)
                                                                                        6
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR
                                                                                        9
    This call starts a nonblocking variant of MPI_SCAN (see Section 6.11).
                                                                                        10
                                                                                        11
6.12.12 Nonblocking Exclusive Scan
                                                                                        12
                                                                                        13
                                                                                        14
MPI_IEXSCAN(sendbuf, recvbuf, count, datatype, op, comm, request)
                                                                                        15
  IN
           sendbuf
                                      starting address of send buffer (choice)
                                                                                        16
                                                                                        17
  OUT
           recvbuf
                                      starting address of receive buffer (choice)
                                                                                        18
  IN
           count
                                      number of elements in input buffer (non-negative
                                                                                        19
                                      integer)
                                                                                        20
  IN
           datatype
                                      datatype of elements of input buffer (handle)
                                                                                        21
                                                                                        22
  IN
                                      operation (handle)
           op
                                                                                        23
                                      intra-communicator (handle)
  IN
           comm
                                                                                        ^{24}
  OUT
                                      communication request (handle)
           request
                                                                                        25
                                                                                        26
C binding
                                                                                        27
int MPI_Iexscan(const void *sendbuf, void *recvbuf, int count,
                                                                                        28
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
                                                                                        29
              MPI_Request *request)
                                                                                        30
                                                                                        31
int MPI_Iexscan_c(const void *sendbuf, void *recvbuf, MPI_Count count,
                                                                                        32
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
                                                                                        33
              MPI_Request *request)
                                                                                        34
Fortran 2008 binding
                                                                                        35
                                                                                        36
MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
                                                                                        37
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                        38
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                        39
    INTEGER, INTENT(IN) :: count
                                                                                        40
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                        41
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                        42
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                        43
                                                                                        44
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                        45
MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
                                                                                        46
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                        47
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                        48
```

1	
T	INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
2	TYPE(MPI_Datatype), INTENT(IN) :: datatype
3	TYPE(MPI_Op), INTENT(IN) :: op
4	TYPE(MPI_Comm), INTENT(IN) :: comm
5	TYPE(MPI_Request), INTENT(OUT) :: request
6	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7	
8	Fortran binding
9	MPI_IEXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)
10	<type> SENDBUF(*), RECVBUF(*)</type>
10	
11	INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR
12	This call starts a nonblocking variant of MPI_EXSCAN (see Section 6.11.2).

 $13 \\ 14$

15

6.13 Persistent Collective Operations

16Many parallel computation algorithms involve repetitively executing a collective commu-17nication operation with the same arguments each time. As with persistent point-to-point 18 operations (see Section 3.9), persistent collective operations allow the MPI programmer to 19 specify operations that will be reused frequently (with fixed arguments). MPI can be de-20signed to select a more efficient way to perform the collective operation based on the param-21eters specified when the operation is initialized. This "planned-transfer" approach [52, 41]22 can offer significant performance benefits for programs with repetitive communication pat-23terns. 24

In terms of data movement, each persistent collective operation has the same effect as its blocking and nonblocking counterparts for intra-communicators and inter-communicators after completion. Likewise, upon completion, persistent collective reduction operations perform the same operation as their blocking and nonblocking counterparts, and the same restrictions and recommendations on reduction orders apply (see also Section 6.9.1).

Initialization calls for MPI persistent collective operations are non-local and follow all the existing rules for collective operations, in particular ordering; programs that do not conform to these restrictions are erroneous. After initialization, all arrays associated with input arguments (such as arrays of counts, displacements, and datatypes in the vector versions of the collectives) must not be modified until the corresponding persistent request is freed with MPI_REQUEST_FREE.

According to the definitions in Section 2.4.2, the persistent collective initialization procedures are incomplete. They are also non-local procedures because they may or may not return before they are called in all MPI processes of the process group associated with the specified communicator.

40 41

42

43

44

45

Advice to users. This is one of the exceptions in which incomplete procedures are non-local and therefore blocking. (End of advice to users.)

The request argument is an output argument that can be used zero or more times with MPI_START or MPI_STARTALL in order to start the collective operation. The request is initially inactive after the initialization call. Once initialized, persistent collective operations can be started in any order and the order can differ among processes in the communicator.

46 47 48

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 non-standard. Persistent collective operations may be optimized during communicator creation or by the initialization operation of an individual persistent collective. Note that communicator-scoped hints should be provided using MPI_COMM_SET_INFO while, for operation-scoped hints, they are supplied to the persistent collective communication initialization functions using the info argument.

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)

Unofficial Draft for Comment Only

 24

1 2 3	TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Info), INTENT(IN) :: info TYPE(MPI_Request), INTENT(OUT) :: request				
4	INTEG	GER, OPTIONA	L, INTENT(OUT) :: ierror		
5 6 7 8	Fortran binding MPI_BARRIER_INIT(COMM, INFO, REQUEST, IERROR) INTEGER COMM, INFO, REQUEST, IERROR				
9 10	Creat	es a persistent	collective communication request for the barrier operation.		
11 12 13	6.13.2 P	ersistent Broa	dcast		
14 15	MPI_BCA	ST_INIT(buffe	r, count, datatype, root, comm, info, request)		
16	INOUT	buffer	starting address of buffer (choice)		
17	IN	count	number of entries in buffer (non-negative integer)		
18	IN	datatype	datatype of buffer (handle)		
19 20	IN	root	rank of broadcast root (integer)		
21	IN	comm	communicator (handle)		
22	IN	info	info argument (handle)		
23 24	OUT	request	communication request (handle)		
25		I			
26	C bindin	g			
27 28	int MPI_H		oid *buffer, int count, MPI_Datatype datatype, t, MPI_Comm comm, MPI_Info info, MPI_Request *request)		
29 30 31	int MPI_H		<pre>(void *buffer, MPI_Count count, MPI_Datatype datatype, t, MPI_Comm comm, MPI_Info info, MPI_Request *request)</pre>		
32	Fortran (2008 binding			
33		-	r, count, datatype, root, comm, info, request, ierror)		
34			DN(), ASYNCHRONOUS :: buffer		
35 36			IN) :: count, root		
37			e), INTENT(IN) :: datatype INTENT(IN) :: comm		
38			INTENT(IN) :: info		
39), INTENT(OUT) :: request		
40	INTEC	GER, OPTIONA	L, INTENT(OUT) :: ierror		
41 42	MPI_Bcast	t_init(buffe	r, count, datatype, root, comm, info, request, ierror)		
43	TYPE	(*), DIMENSI	ON(), ASYNCHRONOUS :: buffer		
44			_COUNT_KIND), INTENT(IN) :: count		
45		(MPI_Datatyp GER, INTENT(e), INTENT(IN) :: datatype		
46		-	INJ :: FOOU INTENT(IN) :: comm		
47 48			INTENT(IN) :: info		

	-	NTENT(OUT) :: request NTENT(OUT) :: ierror	12	
Fortran binding				
	0		4	
	MPI_BCAST_INIT(BUFFER, COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR)			
•	ype> BUFFER(*)	YPE, ROOT, COMM, INFO, REQUEST, IERROR	6	
TW	IEGER COUNT, DATAI	IFE, RUUI, CUMM, INFU, REQUESI, IERRUR	7	
Cre	eates a persistent colle	ctive communication request for the broadcast operation.	8	
			9	
6.13.3	Persistent Gather		10	
			11	
			12	
MPI_GA	ATHER_INIT(sendbuf, info, request)	sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm,	13 14 15	
IN	sendbuf	starting address of send buffer (choice)	16	
			17	
IN	sendcount	number of elements in send buffer (non-negative integer)	18	
IN	sendtype	datatype of send buffer elements (handle)	19	
OUT	recvbuf	address of receive buffer (choice, significant only at	20	
001	recvbur	root)	21 22	
IN	recvcount	number of elements for any single receive	23	
		(non-negative integer, significant only at root)	24	
IN	recvtype	datatype of recv buffer elements (handle, significant	25	
		only at root)	26	
IN	root	rank of receiving process (integer)	27	
	1001		28 29	
IN	comm	communicator (handle)	29 30	
IN	info	info argument (handle)	31	
OUT	request	communication request (handle)	32	
			33	
C bind	ling		34	
		void *sendbuf, int sendcount,	35	
		e sendtype, void *recvbuf, int recvcount,	36	
		e recvtype, int root, MPI_Comm comm, MPI_Info info,	37	
	MPI_Request		38	
· ·			39	
int MP.		nst void *sendbuf, MPI_Count sendcount,	40	
	• •	e sendtype, void *recvbuf, MPI_Count recvcount,	41	
	• •	e recvtype, int root, MPI_Comm comm, MPI_Info info,	42	
	MPI_Request	∞τε ζπερι)	43	
Fortra	n 2008 binding		44	
MPI_Gat		sendcount, sendtype, recvbuf, recvcount, recvtype,	45	
		info, request, ierror)	46	
), INTENT(IN), ASYNCHRONOUS :: sendbuf	47	
IN	TEGER, INTENT(IN)	: sendcount, recvcount, root	48	

```
1
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
2
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
3
         TYPE(MPI_Comm), INTENT(IN) :: comm
4
         TYPE(MPI_Info), INTENT(IN) :: info
5
         TYPE(MPI_Request), INTENT(OUT) :: request
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
     MPI_Gather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
8
                   root, comm, info, request, ierror)
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
         INTEGER, INTENT(IN) :: root
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
     Fortran binding
20
     MPI_GATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
21
                   ROOT, COMM, INFO, REQUEST, IERROR)
22
         <type> SENDBUF(*), RECVBUF(*)
23
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO,
24
                    REQUEST, IERROR
25
         Creates a persistent collective communication request for the gather operation.
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

MPI_GATH	IERV_INIT(sendbuf, sendcount root, comm, info, request	z, sendtype, recvbuf, recvcounts, displs, recvtype,	$\frac{1}{2}$	
IN	sendbuf	starting address of send buffer (choice)	3	
IN	sendcount	number of elements in send buffer (non-negative integer)	4 5 6	
IN	sendtype	datatype of send buffer elements (handle)	7	
OUT	recvbuf	address of receive buffer (choice, significant only at root)	8 9 10	
IN	recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process (significant only at root)	10 11 12 13	
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)	14 15 16 17 18	
IN	recvtype	datatype of recv buffer elements (handle, significant only at root)	19 20	
IN	root	rank of receiving process (integer)	21	
IN	comm	communicator (handle)	22	
IN	info	info argument (handle)	23	
OUT	request	communication request (handle)	24 25	
			26	
C binding int MPI_Gatherv_init(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)				
<pre>int MPI_Gatherv_init_c(const void *sendbuf, MPI_Count sendcount,</pre>				
	MPI_Datatype recvtype MPI_Request *request	e, int root, MPI_Comm comm, MPI_Info info,)	36 37	
	008 binding		38	
MPI_Gathe		nt, sendtype, recvbuf, recvcounts, displs,	39	
TVDF	v 1	, info, request, ierror) T(IN), ASYNCHRONOUS :: sendbuf	40 41	
	ER, INTENT(IN) :: sendcou		42	
	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype			
	*), DIMENSION(), ASYNCH		44	
		NOUS :: recvcounts(*), displs(*)	45	
	TYPE(MPI_Comm), INTENT(IN) :: comm 46			
	<pre>MPI_Info), INTENT(IN) :: MDI Beguart) INTENT(OUT)</pre>		47 48	
IIPE(MPI_Request), INTENT(OUT)	request	40	

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Gatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
3
                   recvtype, root, comm, info, request, ierror)
4
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
5
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
6
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
7
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
8
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
9
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
10
         INTEGER, INTENT(IN) :: root
11
         TYPE(MPI_Comm), INTENT(IN) :: comm
12
         TYPE(MPI_Info), INTENT(IN) :: info
13
         TYPE(MPI_Request), INTENT(OUT) :: request
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     Fortran binding
17
     MPI_GATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
18
                   RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)
19
         <type> SENDBUF(*), RECVBUF(*)
20
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
21
                    COMM, INFO, REQUEST, IERROR
22
         Creates a persistent collective communication request for the gathery operation.
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

6.13.4 Persistent Scatter

MPI_SCATTER_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root,	,
comm, info, request)	

			6
IN	sendbuf	address of send buffer (choice, significant only at	7
		root)	8
IN	sendcount	number of elements sent to each process	9
		(non-negative integer, significant only at root)	10
IN	sendtype	datatype of send buffer elements (handle, significant	11
		only at root)	12
OUT	recvbuf	address of receive buffer (aboice)	13
001	Tecvbul	address of receive buffer (choice)	14
IN	recvcount	number of elements in receive buffer (non-negative	15
		integer)	16
IN	recvtype	datatype of receive buffer elements (handle)	17
IN	root	rank of sending process (integer)	18
	1001		19
IN	comm	communicator (handle)	20
IN	info	info argument (handle)	21
OUT	roquest	communication request (handle)	22
001	request	communication request (handle)	23
			24
C binding	r		25

C binding

C binding	25
<pre>int MPI_Scatter_init(const void *sendbuf, int sendcount,</pre>	26
<pre>MPI_Datatype sendtype, void *recvbuf, int recvcount,</pre>	27
MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,	28
MPI_Request *request)	29
<pre>int MPI_Scatter_init_c(const void *sendbuf, MPI_Count sendcount,</pre>	30
MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,	31
MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,	32
MPI_Request *request)	33
	34
Fortran 2008 binding	35
<pre>MPI_Scatter_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,</pre>	36
recvtype, root, comm, info, request, ierror)	37
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf	38
INTEGER, INTENT(IN) :: sendcount, recvcount, root	39
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype	40
TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf	41
TYPE(MPI_Comm), INTENT(IN) :: comm	42
TYPE(MPI_Info), INTENT(IN) :: info	43
TYPE(MPI_Request), INTENT(OUT) :: request	44
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	45
MPI_Scatter_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,	46
recvtype, root, comm, info, request, ierror)	47
10000, po, 1000, comm, 1110, 104000, 101101/	48

Unofficial Draft for Comment Only

1 2 3 4 5 6 7 8 9	INTI TYPI TYPI INTI INTI TYPI TYPI	EGER(KIND=MPI_CO E(MPI_Datatype), E(*), DIMENSION(EGER, INTENT(IN) E(MPI_Comm), INT E(MPI_Info), INT E(MPI_Request),	<pre>(), INTENT(IN), ASYNCHRONOUS :: sendbuf DUNT_KIND), INTENT(IN) :: sendcount, recvcount INTENT(IN) :: sendtype, recvtype (), ASYNCHRONOUS :: recvbuf :: root CENT(IN) :: comm CENT(IN) :: comm CENT(IN) :: info INTENT(OUT) :: request INTENT(OUT) :: ierror</pre>
11		binding	
12	MP1_SCAT		JF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, ROOT, COMM, INFO, REQUEST, IERROR)
13	<tvr< td=""><td><pre>ne> SENDBUF(*),</pre></td><td></td></tvr<>	<pre>ne> SENDBUF(*),</pre>	
14 15	• -		SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO,
16		REQUEST,	IERROR
17	Crea	ates a persistent co	llective communication request for the scatter operation.
18			
19 20	MPL SCA	ATTERV INIT(send	lbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype,
20	1011_0.6/	•	, info, request)
22	IN	sendbuf	address of send buffer (choice, significant only at
23			root)
24	IN	sendcounts	non-negative integer array (of length group size)
25 26 27			specifying the number of elements to send to each rank (significant only at root)
28	IN	displs	integer array (of length group size). Entry i specifies
29			the displacement (relative to sendbuf) from which to
30			take the outgoing data to process i (significant only
31			at root)
32 33	IN	sendtype	datatype of send buffer elements (handle, significant only at root)
34	OUT	recvbuf	address of receive buffer (choice, significant only at
35 36			root)
37	IN	recvcount	number of elements in receive buffer (non-negative
38			$\mathrm{integer})$
39	IN	recvtype	datatype of receive buffer elements (handle)
40	IN	root	rank of sending process (integer)
41 42	IN	comm	communicator (handle)
43	IN	info	info argument (handle)
44	OUT	request	communication request (handle)
45			
46 47	C bindi	0	
47 48	int MPI		<pre>const void *sendbuf, const int sendcounts[], displs[], MPI_Datatype sendtype, void *recvbuf,</pre>
			aroproll, milloadadype senadype, vota *recvoul,

```
1
              int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm,
                                                                                   2
              MPI_Info info, MPI_Request *request)
                                                                                   3
int MPI_Scatterv_init_c(const void *sendbuf, const MPI_Count sendcounts[],
              const MPI_Aint displs[], MPI_Datatype sendtype, void *recvbuf,
                                                                                   5
              MPI_Count recvcount, MPI_Datatype recvtype, int root,
                                                                                   6
              MPI_Comm comm, MPI_Info info, MPI_Request *request)
                                                                                   7
Fortran 2008 binding
                                                                                   9
MPI_Scatterv_init(sendbuf, sendcounts, displs, sendtype, recvbuf,
                                                                                   10
              recvcount, recvtype, root, comm, info, request, ierror)
                                                                                   11
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
                                                                                   12
                                                                                   13
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                   14
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                   15
    INTEGER, INTENT(IN) :: recvcount, root
                                                                                   16
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   17
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   18
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   19
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   20
MPI_Scatterv_init(sendbuf, sendcounts, displs, sendtype, recvbuf,
                                                                                   21
              recvcount, recvtype, root, comm, info, request, ierror)
                                                                                   22
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                   23
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*)
                                                                                   24
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
                                                                                   25
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                   26
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                   27
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
                                                                                   28
    INTEGER, INTENT(IN) :: root
                                                                                   29
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   30
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   31
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   32
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   33
                                                                                   34
Fortran binding
                                                                                   35
MPI_SCATTERV_INIT(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF,
                                                                                   36
              RECVCOUNT, RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)
                                                                                   37
    <type> SENDBUF(*), RECVBUF(*)
                                                                                   38
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
                                                                                   39
               COMM, INFO, REQUEST, IERROR
                                                                                   40
    Creates a persistent collective communication request for the scattery operation.
                                                                                   41
                                                                                   42
                                                                                   43
                                                                                   44
                                                                                   45
```

```
284
                                        CHAPTER 6. COLLECTIVE COMMUNICATION
1
     6.13.5 Persistent Gather-to-all
2
3
4
     MPI_ALLGATHER_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm,
5
                    info, request)
6
       IN
                sendbuf
                                            starting address of send buffer (choice)
7
8
       IN
                sendcount
                                            number of elements in send buffer (non-negative
9
                                            integer)
10
       IN
                                            datatype of send buffer elements (handle)
                sendtype
11
       OUT
                recvbuf
                                            address of receive buffer (choice)
12
       IN
                                            number of elements received from any process
13
                 recvcount
14
                                            (non-negative integer)
15
       IN
                recvtype
                                            datatype of receive buffer elements (handle)
16
       IN
                                            communicator (handle)
                comm
17
       IN
                info
18
                                            info argument (handle)
19
       OUT
                request
                                            communication request (handle)
20
21
     C binding
22
     int MPI_Allgather_init(const void *sendbuf, int sendcount,
23
                    MPI_Datatype sendtype, void *recvbuf, int recvcount,
24
                    MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
25
                    MPI_Request *request)
26
     int MPI_Allgather_init_c(const void *sendbuf, MPI_Count sendcount,
27
                    MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
28
                    MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
29
                    MPI_Request *request)
30
^{31}
     Fortran 2008 binding
32
     MPI_Allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
33
                    recvtype, comm, info, request, ierror)
34
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
35
         INTEGER, INTENT(IN) :: sendcount, recvcount
36
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
37
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
38
         TYPE(MPI_Comm), INTENT(IN) :: comm
39
         TYPE(MPI_Info), INTENT(IN) :: info
40
         TYPE(MPI_Request), INTENT(OUT) :: request
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
     MPI_Allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
44
                    recvtype, comm, info, request, ierror)
45
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
46
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
47
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
48
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
```

TYF	E(MPI_Comm), INTENT	(IN) :: comm	1
TYF	E(MPI_Info), INTENT	(IN) :: info	2
TYF	E(MPI_Request), INT	ENT(OUT) :: request	3
INT	EGER, OPTIONAL, INT	ENT(OUT) :: ierror	4
Fortrar	ı binding		5
		, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,	6 7
_		M, INFO, REQUEST, IERROR)	8
<ty< td=""><td>pe> SENDBUF(*), REC</td><td>VBUF(*)</td><td>9</td></ty<>	pe> SENDBUF(*), REC	VBUF(*)	9
INT	EGER SENDCOUNT, SENI	DTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,	10
	IERROR		11
Cre	ates a persistent collect	ive communication request for the allgather operation.	12
	1		13
			14
MPI_AL	LGATHERV_INIT(sendb comm, info, requ	uf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, uest)	15 16
IN	sendbuf	starting address of send buffer (choice)	17
IN	sendcount	number of elements in send buffer (non-negative	18
		integer)	19 20
IN	sendtype	datatype of send buffer elements (handle)	21
OUT	recvbuf	address of receive buffer (choice)	22
IN	recvcounts	non-negative integer array (of length group size)	23
		containing the number of elements that are received	24 25
		from each process	25 26
IN	displs	integer array (of length group size). Entry i specifies	20
		the displacement (relative to recvbuf) at which to	28
		place the incoming data from process i	29
IN	recvtype	datatype of receive buffer elements (handle)	30
IN	comm	communicator (handle)	31
IN	info	info argument (handle)	32 33
			34
OUT	request	communication request (handle)	35
C Lind			36
C bind	U	nst void *sendbuf, int sendcount,	37
IIIC FII I	-	<pre>sendtype, void *recvbuf, const int recvcounts[],</pre>	38
	•••	pls[], MPI_Datatype recvtype, MPI_Comm comm,	39
		, MPI_Request *request)	40
int MDT			41
INT MPI	-	<pre>const void *sendbuf, MPI_Count sendcount, sendtype, void *recvbuf,</pre>	42 43
		<pre>senatype, void *recvour, nt recvcounts[], const MPI_Aint displs[],</pre>	43 44
		recvtype, MPI_Comm comm, MPI_Info info,	44
	MPI_Request *	• -	46
	-	-	47
			48

```
1
     Fortran 2008 binding
\mathbf{2}
     MPI_Allgatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
3
                   displs, recvtype, comm, info, request, ierror)
4
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
5
         INTEGER, INTENT(IN) :: sendcount
6
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
7
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
8
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
9
         TYPE(MPI_Comm), INTENT(IN) :: comm
10
         TYPE(MPI_Info), INTENT(IN) :: info
11
         TYPE(MPI_Request), INTENT(OUT) :: request
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
     MPI_Allgatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
14
                   displs, recvtype, comm, info, request, ierror)
15
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
16
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
17
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
18
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
19
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
20
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
21
         TYPE(MPI_Comm), INTENT(IN) :: comm
22
         TYPE(MPI_Info), INTENT(IN) :: info
23
         TYPE(MPI_Request), INTENT(OUT) :: request
24
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26
     Fortran binding
27
     MPI_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
28
                   DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)
29
         <type> SENDBUF(*), RECVBUF(*)
30
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
31
                    INFO, REQUEST, IERROR
32
         Creates a persistent collective communication request for the allgathery operation.
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

$6.13.6 \quad {\sf Persistent \ All-to-All \ Scatter/Gather}$

0.15.			2
			3 4
MPI_		endcount, sendtype, recvbuf, recvcount, recvtype, comm,	4 5
	info, request)		6
IN	sendbuf	starting address of send buffer (choice)	7
IN	sendcount	number of elements sent to each process	8
		(non-negative integer)	9
IN	sendtype	datatype of send buffer elements (handle)	10 11
OU	T recvbuf	address of receive buffer (choice)	12
IN	recvcount	number of elements received from any process	13
		(non-negative integer)	14
IN	recvtype	datatype of receive buffer elements (handle)	15 16
IN	comm	communicator (handle)	10
IN	info	info argument (handle)	18
OU	T request	communication request (handle)	19
00			20
C bi	nding		21
		void *sendbuf, int sendcount,	22 23
	MPI_Datatype s	sendtype, void *recvbuf, int recvcount,	23 24
		recvtype, MPI_Comm comm, MPI_Info info,	25
	MPI_Request *1	cequest)	26
int	MPI_Alltoall_init_c(con	st void *sendbuf, MPI_Count sendcount,	27
		sendtype, void *recvbuf, MPI_Count recvcount,	28
		cecvtype, MPI_Comm comm, MPI_Info info,	29
	MPI_Request *1	request)	30 31
	ran 2008 binding		32
MPI_		sendcount, sendtype, recvbuf, recvcount,	33
		n, info, request, ierror)	34
	INTEGER, INTENT(IN) ::	INTENT(IN), ASYNCHRONOUS :: sendbuf	35
		ENT(IN) :: sendtype, recvtype	36
		ASYNCHRONOUS :: recvbuf	37
	TYPE(MPI_Comm), INTENT(38 39
	TYPE(MPI_Info), INTENT(40
	<pre>TYPE(MPI_Request), INTE</pre>	-	41
	INTEGER, OPTIONAL, INTE	NT(OUT) :: ierror	42
MPI_	Alltoall_init(sendbuf,	sendcount, sendtype, recvbuf, recvcount,	43
	• -	n, info, request, ierror)	44
		INTENT(IN), ASYNCHRONOUS :: sendbuf	45 46
		KIND), INTENT(IN) :: sendcount, recvcount	46 47
		ENT(IN) :: sendtype, recvtype ASYNCHRONOUS :: recvbuf	48

Unofficial Draft for Comment Only

1 2 3 4	TYPE(TYPE([MPI_Comm), INTENT(IN) :: [MPI_Info), INTENT(IN) :: [MPI_Request), INTENT(OUT) ER, OPTIONAL, INTENT(OUT)	info) :: request
5 6 7 8 9 10 11	<type< th=""><th>DALL_INIT(SENDBUF, SENDCOU RECVTYPE, COMM, INFO >> SENDBUF(*), RECVBUF(*)</th><th>JNT, SENDTYPE, RECVBUF, RECVCOUNT, , REQUEST, IERROR) RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,</th></type<>	DALL_INIT(SENDBUF, SENDCOU RECVTYPE, COMM, INFO >> SENDBUF(*), RECVBUF(*)	JNT, SENDTYPE, RECVBUF, RECVCOUNT, , REQUEST, IERROR) RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
12 13 14		-	nunication request for the alltoall operation.
15 16	MPI_ALLI	OALLV_INI (sendbut, sendcou recvtype, comm, info, req	nts, sdispls, sendtype, recvbuf, recvcounts, rdispls, juest)
17	IN	sendbuf	starting address of send buffer (choice)
18 19 20 21	IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank
22 23 24	IN	sdispls	Integer array (of length group size). Entry j specifies the displacement (relative to sendbuf) from which to take the outgoing data destined for process j
25	IN	sendtype	datatype of send buffer elements (handle)
26	OUT	recvbuf	address of receive buffer (choice)
27 28 29 30	IN	recvcounts	non-negative integer array (of length group size) specifying the number of elements that can be received from each rank
31 32 33	IN	rdispls	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i
34 35	IN	recvtype	datatype of receive buffer elements (handle)
36	IN	comm	communicator (handle)
37	IN	info	info argument (handle)
38 39	OUT	request	communication request (handle)
40 41	C binding		
42	int MPI_A		<pre>*sendbuf, const int sendcounts[], MPI_Datatype sendtype, void *recvbuf,</pre>
43		-	[], const int rdispls[],
44 45		MPI_Datatype recvtyp	e, MPI_Comm comm, MPI_Info info,
46		MPI_Request *request)
47 48	int MPI_A		id *sendbuf, const MPI_Count sendcounts[], ls[], MPI_Datatype sendtype,

<pre>void *recvbuf, const MPI_Count recvcounts[],</pre>	1
<pre>const MPI_Aint rdispls[], MPI_Datatype recvtype,</pre>	2
MPI_Comm comm, MPI_Info info, MPI_Request *request)	3
Fortran 2008 binding	4 5
MPI_Alltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf,	6
recvcounts, rdispls, recvtype, comm, info, request, ierror)	7
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf	8
<pre>INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),</pre>	9
recvcounts(*), rdispls(*) TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype	10
TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf	11
TYPE(MPI_Comm), INTENT(IN) :: comm	12
TYPE(MPI_Info), INTENT(IN) :: info	13 14
TYPE(MPI_Request), INTENT(OUT) :: request	15
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	16
MPI_Alltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf,	17
recvcounts, rdispls, recvtype, comm, info, request, ierror)	18
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf	19
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::	20
<pre>sendcounts(*), recvcounts(*)</pre>	21 22
<pre>INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*), rdispla(*)</pre>	22
rdispls(*) TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype	24
TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf	25
TYPE(MPI_Comm), INTENT(IN) :: comm	26
TYPE(MPI_Info), INTENT(IN) :: info	27
TYPE(MPI_Request), INTENT(OUT) :: request	28
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	29 30
Fortran binding	31
MPI_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,	32
RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)	33
<type> SENDBUF(*), RECVBUF(*)</type>	34
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),	35
RECVTYPE, COMM, INFO, REQUEST, IERROR	36
Creates a persistent collective communication request for the all toally operation.	37
	38 39
	40
	41
	42
	43
	44
	45
	46 47
	41

	290	CH	IAPTER 6.	COLLECTIVE COMMUNICATION
1 2	MPI_ALLT(DALLW_INIT(sendbuf, sendcou recvtypes, comm, info, rec	-	sendtypes, recvbuf, recvcounts, rdispls,
3	IN	sendbuf	starting add	lress of send buffer (choice)
4 5 6 7	IN	sendcounts		y (of length group size) specifying the lements to send to each rank (array of e integers)
8 9 10 11	IN	sdispls	the displace	y (of length group size). Entry j specifies ment in bytes (relative to sendbuf) from we the outgoing data destined for process integers)
12 13 14 15	IN	sendtypes		tatypes (of length group size). Entry j e type of data to send to process j (array
16	OUT	recvbuf	address of r	eceive buffer (choice)
17 18 19	IN	recvcounts	number of e	y (of length group size) specifying the lements that can be received from each of non-negative integers)
20 21 22 23 24	IN	rdispls	the displace	y (of length group size). Entry i specifies ment in bytes (relative to recvbuf) at ace the incoming data from process i segers)
25 26 27	IN	recvtypes		atypes (of length group size). Entry i e type of data received from process i ndles)
28	IN	comm	communicat	for (handle)
29 30	IN	info	info argume	nt (handle)
31	OUT	request	communicat	tion request (handle)
32				
33	C binding			
34 35	int MPI_A	<pre>lltoallw_init(const void</pre>		<pre>const int sendcounts[], _Datatype sendtypes[],</pre>
36				<pre>putatype senatypes[], punts[], const int rdispls[],</pre>
37				, MPI_Comm comm, MPI_Info info,
38		MPI_Request *request))	
39	int MPT Al	lltoally init c(const voi	d *sendbuf	, const MPI_Count sendcounts[],
40	1110 111 1_111			MPI_Datatype sendtypes[],
41 42		void *recvbuf, const		
42		const MPI_Aint rdispl	Ls[], const	t MPI_Datatype recvtypes[],
44		MPI_Comm comm, MPI_Ir	nfo info, N	(PI_Request *request)
45	Fortran 20	008 binding		
46		allw_init(sendbuf, sendco	unts, sdis	pls, sendtypes, recvbuf,
47		-	• -	, comm, info, request, ierror)
48	TYPE(*	*), DIMENSION(), INTENT	'(IN), ASYN	CHRONOUS :: sendbuf

```
1
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
                                                                                     2
               recvcounts(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
              recvtypes(*)
    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
                                                                                    10
MPI_Alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                    11
              recvcounts, rdispls, recvtypes, comm, info, request, ierror)
                                                                                    12
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                    13
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
                                                                                    14
               sendcounts(*), recvcounts(*)
                                                                                    15
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                    16
               rdispls(*)
                                                                                    17
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                    18
               recvtypes(*)
                                                                                    19
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                    20
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                    21
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                    22
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                    23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    24
                                                                                    25
Fortran binding
                                                                                    26
MPI_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
                                                                                    27
              RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST, IERROR)
                                                                                    28
    <type> SENDBUF(*), RECVBUF(*)
                                                                                    29
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
                                                                                    30
               RDISPLS(*), RECVTYPES(*), COMM, INFO, REQUEST, IERROR
                                                                                    31
    Creates a persistent collective communication request for the alltoallw operation.
                                                                                    32
                                                                                    33
                                                                                    34
                                                                                    35
                                                                                    36
                                                                                    37
                                                                                    38
                                                                                    39
                                                                                    40
                                                                                    41
                                                                                    42
                                                                                    43
                                                                                    44
                                                                                    45
                                                                                    46
                                                                                    47
```

1 6.13.7 Persistent Reduce $\mathbf{2}$ 3 4 MPI_REDUCE_INIT(sendbuf, recvbuf, count, datatype, op, root, comm, info, request) 5IN sendbuf address of send buffer (choice) 6 OUT recvbuf 7 address of receive buffer (choice, significant only at 8 root) 9 IN count number of elements in send buffer (non-negative 10 integer) 11 datatype of elements of send buffer (handle) IN datatype 12IN reduce operation (handle) 13 ор 14IN root rank of root process (integer) 15IN communicator (handle) comm 16 17IN info info argument (handle) 18 OUT request communication request (handle) 19 20C binding 21int MPI_Reduce_init(const void *sendbuf, void *recvbuf, int count, 22 MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm, 23MPI_Info info, MPI_Request *request) 24int MPI_Reduce_init_c(const void *sendbuf, void *recvbuf, MPI_Count count, 2526MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm, MPI_Info info, MPI_Request *request) 2728Fortran 2008 binding 29 MPI_Reduce_init(sendbuf, recvbuf, count, datatype, op, root, comm, info, 30 request, ierror) 31TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 32 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 33 INTEGER, INTENT(IN) :: count, root 34 TYPE(MPI_Datatype), INTENT(IN) :: datatype 35TYPE(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, 42request, ierror) 43 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 44 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 45INTEGER(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

```
1
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                         2
    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(*)
                                                                                         a
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, INFO, REQUEST, IERROR
                                                                                        10
                                                                                        11
    Creates a persistent collective communication request for the reduce operation.
                                                                                        12
                                                                                        13
6.13.8 Persistent All-Reduce
                                                                                        14
                                                                                        15
                                                                                        16
MPI_ALLREDUCE_INIT(sendbuf, recvbuf, count, datatype, op, comm, info, request)
                                                                                        17
  IN
           sendbuf
                                       starting address of send buffer (choice)
                                                                                        18
                                                                                        19
  OUT
           recvbuf
                                       starting address of receive buffer (choice)
                                                                                        20
                                       number of elements in send buffer (non-negative
  IN
           count
                                                                                        21
                                       integer)
                                                                                        22
  IN
           datatype
                                       datatype of elements of send buffer (handle)
                                                                                        23
                                                                                        24
                                       operation (handle)
  IN
           ор
                                                                                        25
  IN
           comm
                                       communicator (handle)
                                                                                        26
  IN
           info
                                       info argument (handle)
                                                                                        27
                                                                                        28
  OUT
                                       communication request (handle)
           request
                                                                                        29
                                                                                        30
C binding
                                                                                        31
int MPI_Allreduce_init(const void *sendbuf, void *recvbuf, int count,
                                                                                        32
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
                                                                                        33
              MPI_Info info, MPI_Request *request)
                                                                                        34
int MPI_Allreduce_init_c(const void *sendbuf, void *recvbuf,
                                                                                        35
              MPI_Count count, MPI_Datatype datatype, MPI_Op op,
                                                                                        36
              MPI_Comm comm, MPI_Info info, MPI_Request *request)
                                                                                        37
                                                                                        38
Fortran 2008 binding
                                                                                        39
MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info,
                                                                                        40
              request, ierror)
                                                                                        41
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                        42
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                        43
    INTEGER, INTENT(IN) :: count
                                                                                        44
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                        45
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                        46
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                        47
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                        48
```

1 2		-	NTENT(OUT) :: request NTENT(OUT) :: ierror	
3				
4	<pre>MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info,</pre>			
5 6	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf			
7	TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf			
8			NT_KIND), INTENT(IN) :: count	
9		U 1	INTENT(IN) :: datatype	
10		E(MPI_Op), INTENT E(MPI_Comm), INTE	-	
11		E(MPI_COMM), INTE E(MPI_Info), INTE		
12 13			NTENT(OUT) :: request	
13		-	NTENT(OUT) :: ierror	
15	Fortran	binding		
16			UF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO,	
17		REQUEST, IE		
18 19	• •	pe> SENDBUF(*), R		
20	INTI	EGER COUNT, DATAT	YPE, OP, COMM, INFO, REQUEST, IERROR	
21	Crea	ates a persistent colle	ective communication request for the all reduce operation.	
22				
23	6.13.9	Persistent Reduce-S	catter with Equal Blocks	
24				
25 26			OCK INIT (conduct require the comm	
27		info, request)	_OCK_INIT(sendbuf, recvbuf, recvcount, datatype, op, comm,	
28 29	IN	sendbuf	starting address of send buffer (choice)	
30	OUT	recvbuf	starting address of receive buffer (choice)	
31	IN	recvcount	element count per block (non-negative integer)	
32 33 34	IN	datatype	datatype of elements of send and receive buffers (handle)	
35	IN	ор	operation (handle)	
36	IN	comm	communicator (handle)	
37 38	IN	info	info argument (handle)	
39	OUT	request	communication request (handle)	
40		•		
41	C bindi	0		
42	int MPI		<pre>lock_init(const void *sendbuf, void *recvbuf,</pre>	
43 44			nt, MPI_Datatype datatype, MPI_Op op, mm, MPI_Info info, MPI_Request *request)	
45				
46	int MPI		<pre>lock_init_c(const void *sendbuf, void *recvbuf,</pre>	
47			recvcount, MPI_Datatype datatype, MPI_Op op, mm, MPI_Info info, MPI_Request *request)	
48			,	

```
Fortran 2008 binding
                                                                                      \mathbf{2}
MPI_Reduce_scatter_block_init(sendbuf, recvbuf, recvcount, datatype, op,
              comm, info, request, ierror)
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                      10
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                      11
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                      12
                                                                                      13
MPI_Reduce_scatter_block_init(sendbuf, recvbuf, recvcount, datatype, op,
                                                                                      14
              comm, info, request, ierror)
                                                                                      15
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                      16
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                      17
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
                                                                                      18
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                      19
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                      20
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                     21
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                      22
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                      23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                      ^{24}
                                                                                      25
Fortran binding
                                                                                      26
MPI_REDUCE_SCATTER_BLOCK_INIT(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP,
                                                                                     27
              COMM, INFO, REQUEST, IERROR)
                                                                                     28
    <type> SENDBUF(*), RECVBUF(*)
                                                                                     29
    INTEGER RECVCOUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
                                                                                     30
    Creates a persistent collective communication request for the reduce-scatter with equal
                                                                                      31
blocks operation.
                                                                                      32
                                                                                      33
                                                                                     34
                                                                                     35
                                                                                      36
                                                                                     37
                                                                                      38
                                                                                      39
                                                                                      40
                                                                                      41
                                                                                      42
                                                                                      43
                                                                                      44
                                                                                      45
                                                                                      46
                                                                                      47
                                                                                      48
```

```
296
                                        CHAPTER 6. COLLECTIVE COMMUNICATION
1
     6.13.10 Persistent Reduce-Scatter
\mathbf{2}
3
4
     MPI_REDUCE_SCATTER_INIT(sendbuf, recvbuf, recvcounts, datatype, op, comm, info,
5
                    request)
6
       IN
                sendbuf
                                            starting address of send buffer (choice)
7
       OUT
8
                recvbuf
                                            starting address of receive buffer (choice)
9
       IN
                recvcounts
                                            non-negative integer array specifying the number of
10
                                            elements in result distributed to each process. This
11
                                            array must be identical on all calling processes.
12
       IN
                datatype
                                            datatype of elements of input buffer (handle)
13
14
       IN
                op
                                            operation (handle)
15
       IN
                comm
                                            communicator (handle)
16
       IN
                info
                                            info argument (handle)
17
       OUT
                                            communication request (handle)
18
                request
19
20
     C binding
21
     int MPI_Reduce_scatter_init(const void *sendbuf, void *recvbuf,
22
                    const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
23
                    MPI_Comm comm, MPI_Info info, MPI_Request *request)
24
     int MPI_Reduce_scatter_init_c(const void *sendbuf, void *recvbuf,
25
                    const MPI_Count recvcounts[], MPI_Datatype datatype,
26
                    MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request)
27
28
     Fortran 2008 binding
29
     MPI_Reduce_scatter_init(sendbuf, recvbuf, recvcounts, datatype, op, comm,
30
                    info, request, ierror)
^{31}
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
32
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
33
          INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
34
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
35
         TYPE(MPI_Op), INTENT(IN) :: op
36
         TYPE(MPI_Comm), INTENT(IN) :: comm
37
         TYPE(MPI_Info), INTENT(IN) :: info
38
         TYPE(MPI_Request), INTENT(OUT) :: request
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
     MPI_Reduce_scatter_init(sendbuf, recvbuf, recvcounts, datatype, op, comm,
41
                    info, request, ierror)
42
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
43
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
44
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
45
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
46
         TYPE(MPI_Op), INTENT(IN) :: op
47
         TYPE(MPI_Comm), INTENT(IN) :: comm
48
```

TVD	E(MDT Trfa) INTENT(TN) info	1		
TYPE(MPI_Info), INTENT(IN) :: info TYPE(MPI_Request), INTENT(OUT) :: request					
	EGER, OPTIONAL, INTER	-	3		
	binding	DBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,	5		
	INFO, REQUEST,		6		
<tv< td=""><td>pe> SENDBUF(*), RECVI</td><td></td><td>7</td></tv<>	pe> SENDBUF(*), RECVI		7		
v	•	DATATYPE, OP, COMM, INFO, REQUEST, IERROR	8 9		
Cre	ates a persistent collectiv	re communication request for the reduce-scatter operation.	10		
	1		11		
6.13.11	Persistent Inclusive Sca	an	12		
			13		
			14 15		
MPI_SC	AN_INIT(sendbuf, recvbut	f, count, datatype, op, comm, info, request)	16		
IN	sendbuf	starting address of send buffer (choice)	17		
OUT	recvbuf	starting address of receive buffer (choice)	18		
IN	count	number of elements in input buffer (non-negative	19		
	count	integer)	20		
IN	datatype	datatype of elements of input buffer (handle)	21 22		
IN		operation (handle)	23		
	ор		24		
IN	comm	communicator (handle)	25		
IN	info	info argument (handle)	26		
OUT	request	communication request (handle)	27		
			28 29		
C bind	<u> </u>		30		
int MPI		d *sendbuf, void *recvbuf, int count,	31		
		atatype, MPI_Op op, MPI_Comm comm, MPI_Request *request)	32		
			33		
int MPI		oid *sendbuf, void *recvbuf, MPI_Count count,	34		
		atatype, MPI_Op op, MPI_Comm comm, MPI_Request *request)	35 36		
_			37		
	2008 binding		38		
MP1_Sca	n_init(sendbuf, recvi ierror)	buf, count, datatype, op, comm, info, request,	39		
ТҮР	•	INTENT(IN), ASYNCHRONOUS :: sendbuf	40		
		ASYNCHRONOUS :: recvbuf	41		
INTEGER, INTENT(IN) :: count					
TYPE(MPI_Datatype), INTENT(IN) :: datatype 4					
TYPE(MPI_Op), INTENT(IN) :: op					
TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Info), INTENT(IN) :: info					
	E(MPI_INIO), INTENIC		47		

```
1
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Scan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
3
                    ierror)
4
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
5
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
6
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
7
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
8
          TYPE(MPI_Op), INTENT(IN) :: op
9
          TYPE(MPI_Comm), INTENT(IN) :: comm
10
          TYPE(MPI_Info), INTENT(IN) :: info
11
          TYPE(MPI_Request), INTENT(OUT) :: request
12
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     Fortran binding
15
     MPI_SCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,
16
                     IERROR)
17
          <type> SENDBUF(*), RECVBUF(*)
18
          INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
19
          Creates a persistent collective communication request for the inclusive scan operation.
20
21
     6.13.12 Persistent Exclusive Scan
22
23
^{24}
     MPI_EXSCAN_INIT(sendbuf, recvbuf, count, datatype, op, comm, info, request)
25
26
       IN
                 sendbuf
                                             starting address of send buffer (choice)
27
       OUT
                 recvbuf
                                             starting address of receive buffer (choice)
28
29
       IN
                 count
                                             number of elements in input buffer (non-negative
30
                                             integer)
31
       IN
                                             datatype of elements of input buffer (handle)
                 datatype
32
       IN
                 ор
                                             operation (handle)
33
34
       IN
                                             intra-communicator (handle)
                 comm
35
       IN
                 info
                                             info argument (handle)
36
       OUT
                 request
                                             communication request (handle)
37
38
     C binding
39
     int MPI_Exscan_init(const void *sendbuf, void *recvbuf, int count,
40
                    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
41
                    MPI_Info info, MPI_Request *request)
42
43
     int MPI_Exscan_init_c(const void *sendbuf, void *recvbuf, MPI_Count count,
44
                    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
45
                    MPI_Info info, MPI_Request *request)
46
47
48
```

1 Fortran 2008 binding $\mathbf{2}$ MPI_Exscan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request, ierror) TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 4 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 5 6 INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Op), INTENT(IN) :: op TYPE(MPI_Comm), INTENT(IN) :: comm 10 TYPE(MPI_Info), INTENT(IN) :: info 11 TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1213 MPI_Exscan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request, 14 ierror) 15TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 16TYPE(*), 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 20TYPE(MPI_Comm), INTENT(IN) :: comm 21TYPE(MPI_Info), INTENT(IN) :: info 22 TYPE(MPI_Request), INTENT(OUT) :: request 23INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2425Fortran binding 26MPI_EXSCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST, 27IERROR) 28 <type> SENDBUF(*), RECVBUF(*) 29 INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR 30 Creates a persistent collective communication request for the exclusive scan operation. 3132 33 6.14 Correctness 34 A correct, portable program must invoke collective communications so that deadlock will not 35 occur, whether collective communications are synchronizing or not. The following examples 36 illustrate dangerous use of collective routines on intra-communicators. 37 38 **Example 6.25** The following is erroneous. 39 40 switch(rank) { 41 case 0: 42MPI_Bcast(buf1, count, type, 0, comm); 43 MPI_Bcast(buf2, count, type, 1, comm); 44 break; 45case 1: 46MPI_Bcast(buf2, count, type, 1, comm); 47MPI_Bcast(buf1, count, type, 0, comm); 48

28

29

30

 31

32 33

34

break;

```
1
\mathbf{2}
     }
3
4
          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.
5
          Collective operations must be executed in the same order at all members of the com-
6
     munication group.
7
8
     Example 6.26 The following is erroneous.
9
10
     switch(rank) {
11
          case 0:
12
               MPI_Bcast(buf1, count, type, 0, comm0);
13
               MPI_Bcast(buf2, count, type, 2, comm2);
14
               break;
15
          case 1:
16
               MPI_Bcast(buf1, count, type, 1, comm1);
17
               MPI_Bcast(buf2, count, type, 0, comm0);
18
               break;
19
          case 2:
20
               MPI_Bcast(buf1, count, type, 2, comm2);
21
               MPI_Bcast(buf2, count, type, 1, comm1);
22
               break;
23
     }
24
25
          Assume that the group of comm0 is \{0,1\}, of comm1 is \{1, 2\} and of comm2 is \{2,0\}. If
26
     the broadcast is a synchronizing operation, then there is a cyclic dependency: the broadcast
27
```

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.

```
35
     switch(rank) {
36
          case 0:
37
              MPI_Bcast(buf1, count, type, 0, comm);
38
              MPI_Send(buf2, count, type, 1, tag, comm);
39
              break;
40
         case 1:
41
              MPI_Recv(buf2, count, type, 0, tag, comm, status);
42
              MPI_Bcast(buf1, count, type, 0, comm);
43
              break;
44
     }
45
46
47
48
```

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, non-deterministic program.

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
        break;
    case 1:
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
        break;
    case 2:
        MPI_Send(buf2, count, type, 1, tag, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Bcast(buf1, count, type, 0, 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 call at a process. In these situations, it is the user's responsibility to ensure that the same communicator is not used concurrently by two different collective communication calls at the same process.

Advice to implementors. Assume that broadcast is implemented using point-to-point MPI communication. Suppose the following two rules are followed.

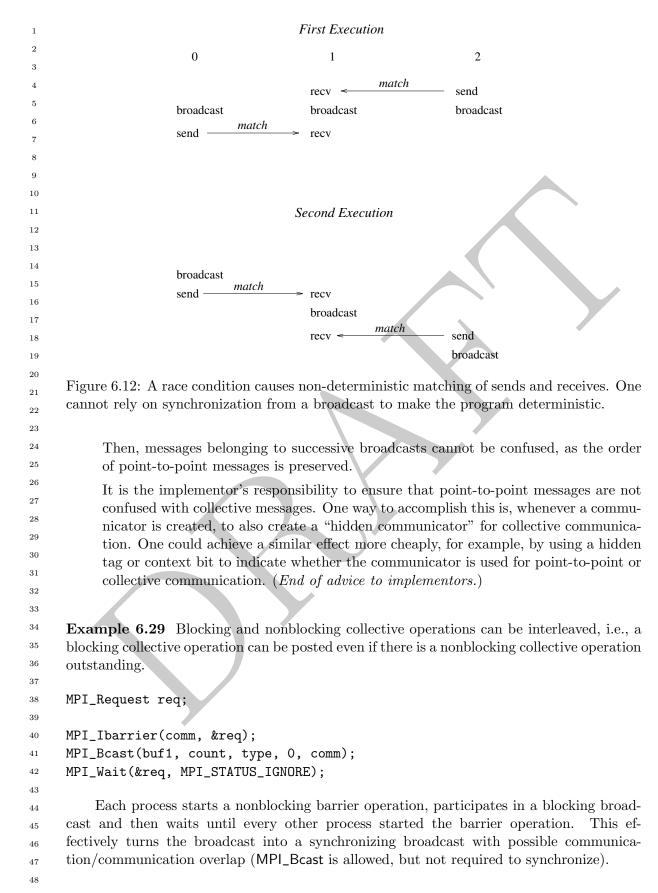
- 1. All receives specify their source explicitly (no wildcards).
- 2. Each process sends all messages that pertain to one collective call before sending any message that pertain to a subsequent collective call.

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\overline{7}$

 24



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.

```
MPI_Request req;
                                                                                          5
switch(rank) {
                                                                                          6
    case 0:
         /* erroneous matching */
         MPI_Ibarrier(comm, &req);
                                                                                          9
         MPI_Bcast(buf1, count, type, 0, comm);
                                                                                          10
         MPI_Wait(&req, MPI_STATUS_IGNORE);
                                                                                          11
         break;
                                                                                          12
    case 1:
                                                                                          13
         /* erroneous matching */
                                                                                          14
         MPI_Bcast(buf1, count, type, 0, comm);
                                                                                          15
         MPI_Ibarrier(comm, &req);
                                                                                          16
         MPI_Wait(&req, MPI_STATUS_IGNORE);
                                                                                          17
         break;
                                                                                          18
}
                                                                                          19
                                                                                          20
    This ordering would match MPI_Ibarrier on rank 0 with MPI_Bcast on rank 1 which is
                                                                                          21
erroneous and the program behavior is undefined. However, if such an order is required, the
                                                                                          22
user must create different duplicate communicators and perform the operations on them.
                                                                                          23
If started with two processes, the following program would be correct:
                                                                                          ^{24}
                                                                                          25
MPI_Request req;
                                                                                          26
MPI_Comm dupcomm;
MPI_Comm_dup(comm, &dupcomm);
                                                                                          27
switch(rank) {
                                                                                          28
    case 0:
                                                                                          29
         MPI_Ibarrier(comm, &req);
                                                                                          30
         MPI_Bcast(buf1, count, type, 0, dupcomm);
                                                                                          31
         MPI_Wait(&req, MPI_STATUS_IGNORE);
                                                                                          32
                                                                                          33
         break;
                                                                                          34
   case 1:
```

```
MPI_Bcast(buf1, count, type, 0, dupcomm);
MPI_Ibarrier(comm, &req);
MPI_Wait(&req, MPI_STATUS_IGNORE);
break;
```

```
}
```

Advice to users. The use of different communicators offers some flexibility regarding the matching of nonblocking collective operations. In this sense, communicators could be used as an equivalent to tags. However, communicator construction might induce overheads so that this should be used carefully. (*End of advice to users.*)

Example 6.31 Nonblocking collective operations can rely on the same progression rules as nonblocking point-to-point messages. Thus, if started with two processes, the following program is a valid MPI program and is guaranteed to terminate:

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

35

36

37

38

39 40

41

42

43

44 45 46

47

```
1
     MPI_Request req;
\mathbf{2}
3
     switch(rank) {
4
          case 0:
5
            MPI_Ibarrier(comm, &req);
6
            MPI_Wait(&req, MPI_STATUS_IGNORE);
7
            MPI_Send(buf, count, dtype, 1, tag, comm);
8
            break;
9
          case 1:
10
            MPI_Ibarrier(comm, &req);
11
            MPI_Recv(buf, count, dtype, 0, tag, comm, MPI_STATUS_IGNORE);
12
            MPI_Wait(&req, MPI_STATUS_IGNORE);
13
            break;
14
     }
15
         The MPI library must progress the barrier in the MPI_Recv call. Thus, the MPI_Wait
16
     call in rank 0 will eventually complete, which enables the matching MPI_Send so all calls
17
     eventually return.
18
19
     Example 6.32 Blocking and nonblocking collective operations do not match. The fol-
20
     lowing example is erroneous.
21
22
     MPI_Request req;
23
24
     switch(rank) {
25
          case 0:
26
            /* erroneous false matching of Alltoall and Ialltoall */
27
            MPI_Ialltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm, &req);
28
            MPI_Wait(&req, MPI_STATUS_IGNORE);
29
            break;
30
          case 1:
31
            /* erroneous false matching of Alltoall and Ialltoall */
32
            MPI_Alltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm);
33
            break;
34
     }
35
36
37
     Example 6.33 Collective and point-to-point requests can be mixed in functions that
38
     enable multiple completions. If started with two processes, the following program is valid.
39
40
     MPI_Request reqs[2];
41
42
     switch(rank) {
43
          case 0:
44
            MPI_Ibarrier(comm, &reqs[0]);
45
            MPI_Send(buf, count, dtype, 1, tag, comm);
46
            MPI_Wait(&reqs[0], MPI_STATUS_IGNORE);
47
            break;
48
          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]);
        44
        MPI_Iallreduce(sbuf3, rbuf3, count, dtype, MPI_SUM, comm3, &reqs[1]);
        45
        break;
        case 1:
            MPI_Iallreduce(sbuf1, rbuf1, count, dtype, MPI_SUM, comm1, &reqs[0]);
        48
```

Unofficial Draft for Comment Only

1 2

4

5 6

78

9

10 11

12 13

14

15

16

17

18

19

20 21

22

23

 $\frac{24}{25}$

26

27

28

29

30 31

32 33

34

35 36

37

38

39

```
306
                                  CHAPTER 6. COLLECTIVE COMMUNICATION
                                      comm1
                              0
                                                  1
                                             comm2
                                comm3
                                        2
               Figure 6.13: Example with overlapping communicators.
      MPI_Iallreduce(sbuf2, rbuf2, count, dtype, MPI_SUM, comm2, &reqs[1]);
      break:
    case 2:
      MPI_Iallreduce(sbuf2, rbuf2, count, dtype, MPI_SUM, comm2, &reqs[0]);
      MPI_Iallreduce(sbuf3, rbuf3, count, dtype, MPI_SUM, comm3, &reqs[1]);
      break;
MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);
     Advice to users. This method can be useful if overlapping neighboring regions (halo
     or ghost zones) are used in collective operations. The sequence of the two calls in
     each process is irrelevant because the two nonblocking operations are performed on
     different communicators. (End of advice to users.)
Example 6.36 The progress of multiple outstanding nonblocking collective operations is
completely independent.
MPI_Request reqs[2];
compute(buf1);
MPI_Ibcast(buf1, count, type, 0, comm, &reqs[0]);
compute(buf2);
MPI_Ibcast(buf2, count, type, 0, comm, &reqs[1]);
MPI_Wait(&reqs[1], MPI_STATUS_IGNORE);
/* nothing is known about the status of the first bcast here */
MPI_Wait(&reqs[0], MPI_STATUS_IGNORE);
```

Finishing the second MPI_IBCAST is completely independent of the first one. This means that it is not guaranteed that the first broadcast operation is finished or even started after the second one is completed via regs[1].

 24

 31

}

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 processes (potentially running unrelated code),
- Abstract process naming to allow libraries to describe their communication in terms suitable to their own data structures and algorithms,
- The ability to "adorn" a set of communicating processes with additional user-defined attributes, such as extra collective operations. This mechanism should provide a means for the user or library writer effectively to extend a message-passing notation.

In addition, a unified mechanism or object is needed for conveniently denoting communication context, the group of communicating processes, to house abstract process naming, and to store adornments.

7.1.2 MPI's Support for Libraries

The corresponding concepts that MPI provides, specifically to support robust libraries, are as follows:

- **Contexts** of communication,
- Groups of processes,
- Virtual topologies,
- Attribute caching,
- Communicators.

Communicators (see [22, 60, 64]) encapsulate all of these ideas in order to provide the appropriate scope for all communication operations in MPI. Communicators are divided into two kinds: intra-communicators for operations within a single group of processes and inter-communicators for operations between two groups of processes.

¹⁹ Caching. Communicators (see below) provide a "caching" mechanism that allows one to ²⁰ associate new attributes with communicators, on par with MPI built-in features. This can ²¹ be used by advanced users to adorn communicators further, and by MPI to implement ²² some communicator functions. For example, the virtual-topology functions described in ²³ Chapter 8 are likely to be supported this way.

24

1

2

3

4 5

6

7 8

9 10

11

12 13

14

15

16

17 18

Groups. Groups define an ordered collection of processes, each with a rank, and it is this group that defines the low-level names for inter-process communication (ranks are used for sending and receiving). Thus, groups define a scope for process names in point-to-point communication. In addition, groups define the scope of collective operations. Groups may be manipulated separately from communicators in MPI, but only communicators can be used in communication operations.

31 32

33 34

35 36

37

38

39

40

41

42

43

44

45 46

Intra-Communicators. The most commonly used means for message-passing in MPI is via intra-communicators. Intra-communicators contain an instance of a group, contexts of communication for both point-to-point and collective communication, and the ability to include virtual topology and other attributes. These features work as follows:

• **Contexts** provide the ability to have separate safe "universes" of message-passing in MPI. A context is akin to an additional tag that differentiates messages. The system manages this differentiation process. The use of separate communication contexts by distinct libraries (or distinct library invocations) insulates communication internal to the library execution from external communication. This allows the invocation of the library even if there are pending communications on "other" communicators, and avoids the need to synchronize entry or exit into library code. Pending point-to-point communications are also guaranteed not to interfere with collective communications within a single communicator.

- **Groups** define the participants in the communication (see above) of a communicator.
- 47 48

- A virtual topology defines a special mapping of the ranks in a group to and from a topology. Special constructors for communicators are defined in Chapter 8 to provide this feature. Intra-communicators as described in this chapter do not have topologies.
- Attributes define the local information that the user or library has added to a communicator for later reference.

Advice to users. The practice in many communication libraries is that there is a unique, predefined communication universe that includes all processes available when the parallel program is initiated; the processes are assigned consecutive ranks. Participants in a point-to-point communication are identified by their rank; a collective communication (such as broadcast) always involves all processes. When using the World Model (Section 11.2), this practice can be followed in MPI by using the predefined communicator MPI_COMM_WORLD. Users who are satisfied with this practice can plug in MPI_COMM_WORLD wherever a communicator argument is required, and can consequently disregard the rest of this chapter. (End of advice to users.)

Inter-Communicators. The discussion has dealt so far with intra-communication: communication within a group. MPI also supports inter-communication: communication between two non-overlapping groups. When an application is built by composing several parallel modules, it is convenient to allow one module to communicate with another using local ranks for addressing within the second module. This is especially convenient in a client-server computing paradigm, where either client or server are parallel. The support of inter-communication also provides a mechanism for the extension of MPI to a dynamic model where not all processes are preallocated at initialization time. In such a situation, it becomes necessary to support communication across "universes." Inter-communication is supported by objects called **inter-communicators**. These objects bind two groups together with communication contexts shared by both groups. For inter-communicators, these features work as follows:

- Contexts provide the ability to have a separate safe "universe" of message-passing between the two groups. A send in the local group is always a receive in the remote group, and vice versa. The system manages this differentiation process. The use of separate communication contexts by distinct libraries (or distinct library invocations) insulates communication internal to the library execution from external communication. This allows the invocation of the library even if there are pending communications on "other" communicators, and avoids the need to synchronize entry or exit into library code.
- A local and remote group specify the recipients and destinations for an inter-communicator.
- Virtual topology is undefined for an inter-communicator.
- As before, attributes cache defines the local information that the user or library has added to a communicator for later reference.

MPI provides mechanisms for creating and manipulating inter-communicators. They are used for point-to-point and collective communication in a related manner to intracommunicators. Users who do not need inter-communication in their applications can safely

Unofficial Draft for Comment Only

 24

 31

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

1

 $\mathbf{2}$

3 4

5 6

> 7 8

9

20

21

22

23

24 25

26

27

28

33 34

35

¹⁰ A **group** is an ordered set of process identifiers (henceforth processes); processes are imple-¹¹ mentation-dependent objects. Each process in a group is associated with an integer **rank**. ¹² Ranks are contiguous and start from zero. Groups are represented by opaque **group ob-**¹³ **jects**, and hence cannot be directly transferred from one process to another. A group is ¹⁴ used within a communicator to describe the participants in a communication "universe" ¹⁵ and to rank such participants (thus giving them unique names within that "universe" of ¹⁶ communication).

There is a special pre-defined group: MPI_GROUP_EMPTY, which is a group with no
 members. The predefined constant MPI_GROUP_NULL is the value used for invalid group
 handles.

Advice to users. MPI_GROUP_EMPTY, which is a valid handle to an empty group, should not be confused with MPI_GROUP_NULL, which in turn is an invalid handle. The former may be used as an argument to group operations; the latter, which is returned when a group is freed, is not a valid argument. (End of advice to users.)

Advice to implementors. A group may be represented by a virtual-to-real processaddress-translation table. Each communicator object (see below) would have a pointer to such a table.

Simple implementations of MPI will enumerate groups, such as in a table. However,
 more advanced data structures make sense in order to improve scalability and memory
 usage with large numbers of processes. Such implementations are possible with MPI.
 (End of advice to implementors.)

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-topoint operations. Contexts are not explicit MPI objects; they appear only as part of the realization of communicators (below).

 $41 \\ 42$

43

44

45

46

Advice to implementors. Distinct communicators in the same process have distinct contexts. A context is essentially a system-managed tag (or tags) needed to make a communicator safe for point-to-point and MPI-defined collective communication. Safety means that collective and point-to-point communication within one communicator do not interfere, and that communication over distinct communicators don't interfere.

A possible implementation for a context is as a supplemental tag attached to messages on send and matched on receive. Each intra-communicator stores the value of its two tags (one for point-to-point and one for collective communication). Communicatorgenerating 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 process. The source and destination of a message are identified by process ranks within that group.

For collective communication, the intra-communicator specifies the set of processes that participate in the collective operation (and their order, when significant). Thus, the communicator restricts the "spatial" scope of communication, and provides machine-independent process addressing through ranks.

Intra-communicators are represented by opaque **intra-communicator objects**, and hence cannot be directly transferred from one process to another.

7.2.4 Predefined Intra-Communicators

When using the World Model for MPI initialization, an initial intra-communicator MPI_COMM_WORLD of all processes the local process can communicate with after initialization (itself included) is defined once MPI_INIT or MPI_INIT_THREAD has been called. In addition, the communicator MPI_COMM_SELF is provided, which includes only the process itself. When using the Sessions Model (Section 11.3) for initialization of MPI resources, MPI_COMM_WORLD and MPI_COMM_SELF are not valid for use as a communicator. See the discussion concerning use of MPI named constants in 2.5.4 for valid uses of MPI_COMM_WORLD and MPI_COMM_SELF prior to initialization of MPI.

The predefined constant MPI_COMM_NULL is the value used for invalid communicator handles.

In a static-process-model implementation of MPI, all processes that participate in the computation are available after MPI is initialized. For this case, MPI_COMM_WORLD is a communicator of all processes available for the computation; this communicator has the same value in all processes. In an implementation of MPI where processes can dynamically join an MPI execution, it may be the case that a process starts an MPI computation without having access to all other processes. In such situations, MPI_COMM_WORLD is a communicator incorporating all processes with which the joining process can immediately

311

1

 $\mathbf{2}$

3

4

5

6

7

8

9

10

11 12

13 14

15

16

17

18

19

20

21

22

23

 24

25

26

27

28 29

30 31

32 33

34

35

36 37

38 39

40

48

```
1
     communicate. Therefore, MPI_COMM_WORLD may simultaneously represent disjoint groups
\mathbf{2}
      in different processes.
3
          All MPI implementations are required to provide the MPI_COMM_WORLD communi-
4
      cator. It cannot be deallocated during the life of a process. The group corresponding to
\mathbf{5}
      this communicator does not appear as a pre-defined constant, but it may be accessed using
6
      MPI_COMM_GROUP (see below). MPI does not specify the correspondence between the
7
      process rank in MPI_COMM_WORLD and its (machine-dependent) absolute address. Neither
8
      does MPI specify the function of the host process, if any. Other implementation-dependent,
9
     predefined communicators may also be provided.
10
11
      7.3
            Group Management
12
13
      This section describes the manipulation of process groups in MPI. These operations are
14
      local and their execution does not require interprocess communication.
15
16
     7.3.1 Group Accessors
17
18
19
      MPI_GROUP_SIZE(group, size)
20
21
       IN
                                              group (handle)
                 group
22
       OUT
                                              number of processes in the group (integer)
                 size
23
^{24}
      C binding
25
      int MPI_Group_size(MPI_Group group, int *size)
26
27
      Fortran 2008 binding
28
     MPI_Group_size(group, size, ierror)
29
          TYPE(MPI_Group), INTENT(IN) :: group
30
          INTEGER, INTENT(OUT) :: size
^{31}
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
      Fortran binding
33
     MPI_GROUP_SIZE(GROUP, SIZE, IERROR)
34
          INTEGER GROUP, SIZE, IERROR
35
36
37
     MPI_GROUP_RANK(group, rank)
38
39
       IN
                  group
                                              group (handle)
40
       OUT
                                              rank of the calling process in group, or
                 rank
41
                                              MPI_UNDEFINED if the process is not a member
42
                                              (integer)
43
44
      C binding
45
      int MPI_Group_rank(MPI_Group group, int *rank)
46
47
48
```

MPI_Group TYPE(INTEG	2008 binding p_rank(group, rank, ierro MPI_Group), INTENT(IN) : ER, INTENT(OUT) :: rank ER, OPTIONAL, INTENT(OUT	: group	1 2 3 4 5		
	Dinding P_RANK(GROUP, RANK, IERRO EER GROUP, RANK, IERROR	R)	6 7 8 9 10		
MPI GROU	JP TRANSLATE RANKS(gro	pup1, n, ranks1, group2, ranks2)	11 12		
IN	group1	group1 (handle)	13		
IN	n	number of ranks in ranks1 and ranks2 arrays (integer)	14		
	ranks1		15		
IN		array of zero or more valid ranks in group1	16 17		
IN	group2	group2 (handle)	18		
OUT	ranks2	array of corresponding ranks in group2,	19		
		MPI_UNDEFINED when no correspondence exists.	20		
C binding	ч г		21		
		_Group group1, int n, const int ranks1[],	22 23		
	MPI_Group group2, in		23 24		
Fortron 9	2008 binding		25		
	0	n, ranks1, group2, ranks2, ierror)	26		
-	<pre>MPI_Group), INTENT(IN) :</pre>		27		
	ER, INTENT(IN) :: n, ran		28		
INTEG	ER, INTENT(OUT) :: ranks	2(n)	29		
INTEG	ER, OPTIONAL, INTENT(OUT) :: ierror	30 31		
Fortran b	binding		32		
MPI_GROUP	P_TRANSLATE_RANKS(GROUP1,	N, RANKS1, GROUP2, RANKS2, IERROR)	33		
INTEG	ER GROUP1, N, RANKS1(*),	GROUP2, RANKS2(*), IERROR	34		
This f	unction is important for deter	mining the relative numbering of the same processes	35		
	-	one knows the ranks of certain processes in the group	36		
		to know their ranks in a subset of that group.	37		
		${\rm input} \ {\rm to} \ {\sf MPI_GROUP_TRANSLATE_RANKS}, {\rm which}$	38 39		
returns MF	PI_PROC_NULL as the translat	ed rank.	40		
			41		
			42		
			43		
			44		
	45				
			$46 \\ 47$		
	48				

1 MPI_GROUP_COMPARE(group1, group2, result) 2 IN group1 first group (handle) 3 IN group2 second group (handle) 4 5OUT result result (integer) 6 $\overline{7}$ C binding 8 int MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result) 9 Fortran 2008 binding 10 MPI_Group_compare(group1, group2, result, ierror) 11 TYPE(MPI_Group), INTENT(IN) :: group1, group2 12INTEGER, INTENT(OUT) :: result 13 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1415Fortran binding 16MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR) 17INTEGER GROUP1, GROUP2, RESULT, IERROR 18

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.

2324

7.3.2 Group Constructors

25MPI provides two approaches to constructing groups. In the first approach, MPI procedures 26are provided to subset and superset existing groups. These constructors construct new 27groups from existing groups. In the second approach, a group is created using a session 28handle and associated process set. This second approach is available when using the Sessions 29 Model. With both approaches, these are local operations, and distinct groups may be 30 defined on different processes; a process may also define a group that does not include itself. 31 Consistent definitions are required when groups are used as arguments in communicator 32 creation functions. When using the World Model for initializing MPI, the base group, upon 33 which all other groups are defined, is the group associated with the initial communicator 34 MPI_COMM_WORLD (accessible through the function MPI_COMM_GROUP). 35

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.)

46 47

36

37

38

39

40

41 42

43

44

45

IN	comm	communicator (handla)	
		communicator (handle)	
OUT	group	group corresponding to comm (handle)	
~			
C bindi			
nt MPI_	_Comm_group(MP1_Co	nm comm, MPI_Group *group)	
	2008 binding		
	_group(comm, grou		
	E(MPI_Comm), INTEN		
	E(MPI_Group), INTE	TENT(OUT) :: ierror	
	GER, OFIIONAL, IN		
	binding		
	1_GROUP(COMM, GROU		
TN.I.F	EGER COMM, GROUP,	IERROR	
MPI.	_COMM_GROUP retu	rns in group a handle to the group of comm.	
APL GR	OUP_UNION(group1,	group? newgroup)	
IN	group1	first group (handle)	
IN	group2	second group (handle)	
OUT	newgroup	union group (handle)	
C bindi	U		
nt MPI_	-	coup group1, MPI_Group group2,	
	MPI_Group *n	ewgroup)	
Fortran	2008 binding		
		coup2, newgroup, ierror)	
		NT(IN) :: group1, group2	
	-	VT(OUT) :: newgroup	
1111	GER, UPIIUNAL, IN	TENT(OUT) :: ierror	
	binding		
		ROUP2, NEWGROUP, IERROR)	
INTE	GER GROUP1, GROUP	2, NEWGROUP, IERROR	
		(group1 group2 powgroup)	
		(group1, group2, newgroup)	
IN	group1	first group (handle)	
IN	group2	second group (handle)	
OUT	newgroup	intersection group (handle)	
		,	

```
1
                    MPI_Group *newgroup)
\mathbf{2}
     Fortran 2008 binding
3
     MPI_Group_intersection(group1, group2, newgroup, ierror)
4
          TYPE(MPI_Group), INTENT(IN) :: group1, group2
5
          TYPE(MPI_Group), INTENT(OUT) :: newgroup
6
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
     Fortran binding
9
     MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR)
10
          INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
11
12
13
     MPI_GROUP_DIFFERENCE(group1, group2, newgroup)
14
       IN
                 group1
                                             first group (handle)
15
16
       IN
                 group2
                                             second group (handle)
17
       OUT
                 newgroup
                                             difference group (handle)
18
19
     C binding
20
     int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
21
                    MPI_Group *newgroup)
22
23
     Fortran 2008 binding
^{24}
     MPI_Group_difference(group1, group2, newgroup, ierror)
          TYPE(MPI_Group), INTENT(IN) :: group1, group2
25
26
          TYPE(MPI_Group), INTENT(OUT) :: newgroup
27
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
     Fortran binding
29
     MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)
30
          INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
^{31}
32
     The set-like operations are defined as follows:
33
     union All elements of the first group (group1), followed by all elements of second group
34
           (group2) not in the first group.
35
36
     intersect All elements of the first group that are also in the second group, ordered as in
37
          the first group.
38
39
     difference All elements of the first group that are not in the second group, ordered as in
40
           the first group.
41
     Note that for these operations the order of processes in the output group is determined
42
     primarily by order in the first group (if possible) and then, if necessary, by order in the
43
     second group. Neither union nor intersection are commutative, but both are associative.
44
     The new group can be empty, that is, equal to MPI_GROUP_EMPTY.
45
46
47
48
```

MPI_GROUP_INCL(group, n, ranks, newgroup)

	(= · · _ =		
IN	group	group (handle)	2
	0 - 1	0.1.1.1.1	3
IN	n	number of elements in array $ranks$ (and size of	4
		newgroup) (integer)	5
IN	ranks	ranks of processes in group to appear in newgroup	6
		(array of integers)	7
		where we can derive define a basis in the ender defined	8
OUT	newgroup	new group derived from above, in the order defined	9
		by ranks (handle)	10
			10

C binding

Fortran 2008 binding

<pre>MPI_Group_incl(group, n, ranks, newgroup, ierror)</pre>	
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 processes in group with ranks ranks[0],..., ranks[n-1]; the process with rank i in newgroup is the process with rank ranks[i] in group. Each of the n elements of ranks must be a valid rank in group and all elements must be distinct, or else the program is erroneous. If n = 0, then newgroup is MPI_GROUP_EMPTY. This function can, for instance, be used to reorder the elements of a group. See also MPI_GROUP_COMPARE.

MPI_GROUP_EXCL(group	, n, ranks, newgroup)
IN group	group (handle)
IN n	number of elements in array

IIN	n	number of elements in array ranks (integer)	36
IN	ranks	array of integer ranks of processes in group not to	37
		appear in newgroup	38
OUT	newgroup	new group derived from above, preserving the order	39
	0	defined by group (handle)	40
			41

C binding

Fortran 2008 binding MPI_Group_excl(group, n, ranks, newgroup, ierror) TYPE(MPI_Group), INTENT(IN) :: group

Unofficial Draft for Comment Only

1. (.

```
1
           INTEGER, INTENT(IN) :: n, ranks(n)
2
           TYPE(MPI_Group), INTENT(OUT) :: newgroup
3
           INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
      Fortran binding
5
      MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)
6
           INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
7
8
          The function MPI_GROUP_EXCL creates a group of processes newgroup that is obtained
9
      by deleting from group those processes with ranks ranks[0],..., ranks[n-1]. The ordering of
10
      processes in newgroup is identical to the ordering in group. Each of the n elements of ranks
11
      must be a valid rank in group and all elements must be distinct; otherwise, the program is
12
      erroneous. If n = 0, then newgroup is identical to group.
13
14
      MPI_GROUP_RANGE_INCL(group, n, ranges, newgroup)
15
16
        IN
                                                 group (handle)
                  group
17
                                                 number of triplets in array ranges (integer)
        IN
                  n
18
        IN
                                                 a one-dimensional array of integer triplets, of the
                   ranges
19
                                                 form (first rank, last rank, stride) indicating ranks in
20
                                                 group of processes to be included in newgroup
21
22
                                                 new group derived from above, in the order defined
        OUT
                   newgroup
23
                                                 by ranges (handle)
^{24}
25
      C binding
26
      int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],
27
                      MPI_Group *newgroup)
28
      Fortran 2008 binding
29
      MPI_Group_range_incl(group, n, ranges, newgroup, ierror)
30
           TYPE(MPI_Group), INTENT(IN) :: group
^{31}
           INTEGER, INTENT(IN) :: n, ranges(3, n)
32
           TYPE(MPI_Group), INTENT(OUT) :: newgroup
33
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
      Fortran binding
36
      MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)
37
           INTEGER GROUP, N, RANGES(3, *), NEWGROUP, IERROR
38
39
      If ranges consists of the triplets
40
            (first_1, last_1, stride_1), \ldots, (first_n, last_n, stride_n)
41
42
      then newgroup consists of the sequence of processes in group with ranks
43
           first_1, first_1 + stride_1, \dots, first_1 + \left| \frac{last_1 - first_1}{stride_1} \right| stride_1, \dots,
44
45
46
           first_n, first_n + stride_n, \dots, first_n + \left| \frac{last_n - first_n}{stride_n} \right| stride_n.
47
48
```

Each computed rank must be a valid rank in group and all computed ranks must be distinct, or else the program is erroneous. Note that we may have $first_i > last_i$, and $stride_i$ may be negative, but cannot be zero.

The functionality of this routine is specified to be equivalent to expanding the array of ranges to an array of the included ranks and passing the resulting array of ranks and other arguments to MPI_GROUP_INCL. A call to MPI_GROUP_INCL is equivalent to a call to MPI_GROUP_RANGE_INCL with each rank i in ranks replaced by the triplet (i,i,1) in the argument ranges.

MPI_GROUP_RANGE_EXCL(group, n, ranges, newgroup)

IN	group	group (handle)	12
IN	n	number of triplets in array ranges (integer)	13
		number of unpieus in array ranges (integer)	14
IN	ranges	a one-dimensional array of integer triplets, of the	15
		form (first rank, last rank, stride) indicating ranks in	16
		group of processes to be excluded from the output	17
		group newgroup (array of integers)	18
OUT	newgroup	new group derived from above, preserving the order	19
	2 .	in group (handle)	20
			21

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	
<pre>INTEGER, INTENT(IN) :: n, ranges(3, n)</pre>	
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 whenenever possible, as high-quality implementations will take advantage of this fact. (End of advice to users.)

Unofficial Draft for Comment Only

1

2

3

4

5

6 $\overline{7}$

8

9 10

11

22

23

 24

25

33

34

35

36

37

38

39

40

41

4243

44

45

320 CHAPTER 7. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

```
1
           Advice to implementors. The range operations should be implemented, if possible,
\mathbf{2}
           without enumerating the group members, in order to obtain better scalability (time
3
           and space). (End of advice to implementors.)
4
5
6
     MPI_GROUP_FROM_SESSION_PSET(session, pset_name, newgroup)
7
8
       IN
                 session
                                            session (handle)
9
       IN
                 pset_name
                                             name of process set to use to create the new group
10
                                             (string)
11
       OUT
                 newgroup
                                             new group derived from supplied session and process
12
                                             set (handle)
13
14
     C binding
15
     int MPI_Group_from_session_pset(MPI_Session session, const char *pset_name,
16
                    MPI_Group *newgroup)
17
18
     Fortran 2008 binding
19
     MPI_Group_from_session_pset(session, pset_name, newgroup, ierror)
20
         TYPE(MPI_Session), INTENT(IN) :: session
21
         CHARACTER(LEN=*), INTENT(IN) :: pset_name
22
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
23
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{24}
     Fortran binding
25
     MPI_GROUP_FROM_SESSION_PSET(SESSION, PSET_NAME, NEWGROUP, IERROR)
26
         INTEGER SESSION, NEWGROUP, IERROR
27
         CHARACTER*(*) PSET_NAME
28
29
         The function MPI_GROUP_FROM_SESSION_PSET creates a group newgroup using the
30
     provided session handle and process set. The process set name must be one returned from
^{31}
     an invocation of MPI_SESSION_GET_NTH_PSET using the supplied session handle. If the
32
     pset_name does not exist, MPI_GROUP_NULL will be returned in the newgroup argument.
33
     As with other group constructors, MPI_GROUP_FROM_SESSION_PSET is a local function.
34
     See Section 11.3 for more information on sessions and process sets.
35
36
            Group Destructors
     7.3.3
37
38
39
     MPI_GROUP_FREE(group)
40
41
       INOUT
                                             group (handle)
                 group
42
43
     C binding
44
     int MPI_Group_free(MPI_Group *group)
45
     Fortran 2008 binding
46
     MPI_Group_free(group, ierror)
47
         TYPE(MPI_Group), INTENT(INOUT) :: group
48
```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

MPI_GROUP_FREE(GROUP, IERROR) INTEGER GROUP, IERROR

This operation marks a group object for deallocation. The handle group is set to MPI_GROUP_NULL by the call. Any on-going operation using this group will complete normally.

Advice to implementors. One can keep a reference count that is incremented for each call to MPI_COMM_GROUP, MPI_COMM_CREATE, MPI_COMM_DUP, and MPI_COMM_IDUP, and decremented for each call to MPI_GROUP_FREE or MPI_COMM_FREE; the group object is ultimately deallocated when the reference count drops to zero. (*End of advice to implementors.*)

7.4 Communicator Management

This section describes the manipulation of communicators in MPI. Operations that access communicators are local and their execution does not require interprocess communication. Operations that create communicators are collective and may require interprocess communication.

Advice to implementors. High-quality implementations should amortize the overheads associated with the creation of communicators (for the same group, or subsets thereof) over several calls, by allocating multiple contexts with one collective communication. (*End of advice to implementors.*)

7.4.1 Communicator Accessors

The following are all local operations.

MPI_COMM_SIZE(comm, size)

IN	comm	communicator (handle)
OUT	size	number of processes in the group of $comm$ (integer)

C binding

int MPI_Comm_size(MPI_Comm comm, int *size)

Fortran 2008 binding

MPI_Comm_size(comm, size, ierror)
 TYPE(MPI_Comm), INTENT(IN) :: comm
 INTEGER, INTENT(OUT) :: size
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

```
MPI_COMM_SIZE(COMM, SIZE, IERROR)
INTEGER COMM, SIZE, IERROR
```

 24

	322	CHAPTER 7. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING	
1		<i>Rationale.</i> This function is equivalent to accessing the communicator's group with	
2		MPI_COMM_GROUP (see above), computing the size using MPI_GROUP_SIZE, and	
3		then freeing the temporary group via $MPI_GROUP_FREE.$ However, this function is	
4		so commonly used that this shortcut was introduced. (End of rationale.)	
5			
6		Advice to users. This function indicates the number of processes involved in a	
7		communicator. For MPI_COMM_WORLD, it indicates the total number of processes	
8		available unless the number of processes has been changed by using the functions	
9 10		described in Chapter 11; note that the number of processes in MPI_COMM_WORLD does not change during the life of an MPI program.	
11			
12		This call is often used with the next call to determine the amount of concurrency	
13	available for a specific library or program. The following call, MPI_COMM_RANK		
14		indicates the rank of the process that calls it in the range from $0, \ldots$, size-1, where	
15		size is the return value of MPI_COMM_SIZE.(<i>End of advice to users.</i>)	
16			
17			
18	MPI_	_COMM_RANK(comm, rank)	
19	IN	comm communicator (handle)	
20 21	OU	Trankrank of the calling process in group of comm (integer)	
22			
23	C bi	inding	
24	int	MPI_Comm_rank(MPI_Comm comm, int *rank)	
25	Fort	ran 2008 binding	
26		Comm_rank(comm, rank, ierror)	
27		TYPE(MPI_Comm), INTENT(IN) :: comm	
28		INTEGER, INTENT(OUT) :: rank	
29		INTEGER, OPTIONAL, INTENT(OUT) :: ierror	
30 31	Fort	ran binding	
32		COMM_RANK(COMM, RANK, IERROR)	
33		INTEGER COMM, RANK, IERROR	
34			
35		Patienale. This function is equivalent to accessing the communicator's group with	
36		<i>Rationale.</i> This function is equivalent to accessing the communicator's group with MPI_COMM_GROUP (see above), computing the rank using MPI_GROUP_RANK,	
37		and then freeing the temporary group via MPI_GROUP_FREE. However, this function	
38		is so commonly used that this shortcut was introduced. (<i>End of rationale.</i>)	
39		is so commonly about this shorteat was introduced. (2000 of 100000000)	
40		Advice to users. This function gives the rank of the process in the particular commu-	
41		nicator's group. It is useful, as noted above, in conjunction with MPI_COMM_SIZE.	
42		Many programs will be written with the supervisor/executor or manager/worker	
43		model, where one process (such as the rank-zero process) will play a supervisory	
44		role, and the other processes will serve as compute nodes. In this framework, the	
45		two preceding calls are useful for determining the roles of the various processes of a	
46 47		communicator. (End of advice to users.)	
48			

MPL COMM_COMPARE(comm1, comm2, result)

	(,
IN	comm1	first communicator (handle)
IN	comm2	second communicator (handle)
OUT	result	result (integer)

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 processes in the group or groups associated with comm, with the exception of MPI_COMM_CREATE_GROUP, MPI_COMM_CREATE_FROM_GROUP, and MPI_INTERCOMM_CREATE_FROM_GROUPS. MPI_COMM_CREATE_GROUP and MPI_COMM_CREATE_FROM_GROUP are invoked only by the processes in the group of the new communicator being constructed. MPI_INTERCOMM_CREATE_FROM_GROUPS is invoked by all the processes in the local and remote groups of the new communicator being constructed. See the discussion below for the definition of local and remote groups.

Rationale. Note that, when using the World Model, there is a chicken-and-egg aspect to MPI in that a communicator is needed to create a new communicator. In the World Model, the base communicator for all MPI communicators is predefined outside of MPI, and is MPI_COMM_WORLD. The World Model was arrived at after considerable debate, and was chosen to increase "safety" of programs written in MPI. (*End of rationale.*)

This chapter presents the following communicator construction routines: MPI_COMM_CREATE, MPI_COMM_DUP, MPI_COMM_IDUP, MPI_COMM_DUP_WITH_INFO, MPI_COMM_IDUP_WITH_INFO, MPI_COMM_SPLIT and MPI_COMM_SPLIT_TYPE can be used to create both intra-communicators and intercommunicators; MPI_COMM_CREATE_GROUP, MPI_COMM_CREATE_FROM_GROUP and MPI_INTERCOMM_MERGE (see Section 7.6.2) can be used to create intra-communicators;

```
1
     MPI_INTERCOMM_CREATE and MPI_INTERCOMM_CREATE_FROM_GROUPS (see Sec-
\mathbf{2}
     tion 7.6.2) can be used to create inter-communicators.
3
          An intra-communicator involves a single group while an inter-communicator involves
4
     two groups. Where the following discussions address inter-communicator semantics, the
\mathbf{5}
     two groups in an inter-communicator are called the left and right groups. A process in an
6
     inter-communicator is a member of either the left or the right group. From the point of
7
     view of that process, the group that the process is a member of is called the local group; the
8
     other group (relative to that process) is the remote group. The left and right group labels
9
     give us a way to describe the two groups in an inter-communicator that is not relative to
10
     any particular process (as the local and remote groups are).
11
12
     MPI_COMM_DUP(comm, newcomm)
13
14
       IN
                                              communicator (handle)
                 comm
15
       OUT
                                              copy of comm (handle)
                 newcomm
16
17
     C binding
18
     int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
19
20
     Fortran 2008 binding
21
     MPI_Comm_dup(comm, newcomm, ierror)
22
          TYPE(MPI_Comm), INTENT(IN) :: comm
23
          TYPE(MPI_Comm), INTENT(OUT) :: newcomm
^{24}
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
     Fortran binding
26
     MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
27
          INTEGER COMM, NEWCOMM, IERROR
28
29
          MPI_COMM_DUP duplicates the existing communicator comm with associated key
30
     values and topology information. For each key value, the respective copy callback function
^{31}
     determines the attribute value associated with this key in the new communicator; one
32
     particular action that a copy callback may take is to delete the attribute from the new
33
     communicator. MPI_COMM_DUP returns in newcomm a new communicator with the same
34
     group or groups, same topology, and any copied cached information, but a new context (see
35
     Section 7.7.1).
36
37
           Advice to users. This operation is used to provide a parallel library with a duplicate
38
           communication space that has the same properties as the original communicator. This
           includes any attributes (see below) and topologies (see Chapter 8). This call is valid
39
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
           of the call. Other models of communicator management are also possible.
43
44
           This call applies to both intra- and inter-communicators. (End of advice to users.)
45
46
           Advice to implementors. One need not actually copy the group information, but only
47
           add a new reference and increment the reference count. Copy on write can be used
48
           for the cached information. (End of advice to implementors.)
```

Unofficial Draft for Comment Only

<pre>MPI_COMM_DUP_WITH_INFO(comm, info, newcomm) N comm communicator (handle) N info info object (handle) OUT newcomm copy of comm (handle) C binding int MPI_Comm_dup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm) Fortran 2008 binding WPI_Comm_dup_with_info(comm, info, newcomm, ierror) TYPE(MPI_Comm), INTENT(IN) :: info TYPE(MPI_Comm), INTENT(IN) :: info TYPE(MPI_Comm), INTENT(IN) :: ierror Fortran binding MPI_COMM_DUP_WITH_INFO(COMM, INFO, NEWCOMM, IERROR) INTEGER, OPTIONAL, INTENT(OUT) :: newcomm 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 recation time. However, for legacy reasons, most communicator relate of any communicator a nifo argument. One may associate info hints with a duplicate of any communicator a creation time through a call to MPI_COMM_DUP_WITH_INFO. (End of rationale.) MPI_COMM_IDUP(comm, newcomm, request) N C comm communicator (handle) OUT request communication request (handle) CUT request communication request (handle) CC binding MPI_Comm_idup(MPI_Comm comm, MPI_Comm *newcomm, MPI_Request *request) Fortran 2008 binding MPI_Comm_idup(MPI_Comm, request, ierror) TYPE(MPI_Comm), INTENT(INT) :: comm TYPE(MPI_Comm), INTENT(INT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_COMM, IDUP(COMM, NEWCOMM, REQUEST, IERROR) INTEGER COMM, NEWCOMM, REQUEST, IERROR</pre>			
N info info object (handle) OUT newcomm copy of comm (handle) C binding int MPI_Comm_dup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm) Fortran 2008 binding WPI_Comm_dup_with_info(comm, info, newcomm, ierror) TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Comm), INTENT(UDT) :: newcomm INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding WPI_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 receation time. However, for legacy reasons, most communicator creation calls do not provide an info argument. One may associate into bints 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) N NUT comm communicator (handle) OUT request communicator (handle) OUT request communicator (handle) OUT request communicator (handle) OUT request communicator (handle)			
OUT newcomm copy of comm (handle) 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_Comm), INTENT(OUT) :: newcomm TYPE(MPI_Comm), INTENT(OUT) :: ierror Fortran binding MPI_COMM_DUP_WITH_INFO(COMM, INFO, NEWCOMM, IERROR) INTEGER COMM, INFO, NEWCOMM, IERROR) INTEGER 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 reaction an info argument. One may associate info bints will only be valid, at communicator reaction and info argument. One may associate info bints will only be valid, at communicator reaction at creation time through a call to MPI_COMM_DUP_WITH_INFO. (End of rationale.) MPI_COMM_IDUP(comm, newcomm, request) IN N comm communicator (handle) OUT newcomm copy of comm (handle) OUT newcomm, newcomm, request, ierror) TYPE(MPI_Comm], INTENT(IN) :: comm TYPE(MPI_Comm, INTENT(OUT), ASYNCHRONOUS :: newcomm TYPE(MPI_Comm), INTENT(OUT), SYNCHRONOUS :: newcomm TYPE(MPI_Comm), INTENT(OUT) :: request TYPE(MPI_Comm), INTENT(OUT) :: request TYPE(MPI_Request), INTENT(OUT) :: request TYPE(MPI_Request), INTENT(OUT) :: request TY	IN	comm	communicator (handle)
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) :: ioomm TYPE(MPI_Comm), INTENT(UN) :: info TYPE(MPI_Comm), INTENT(UT) :: newcomm INTEGER, OPTIONAL, 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 reaction time. However, for legacy reasons, most communicator eration calls do not provide an info argument. One may associate info hints will aduplicate 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 copy of comm (handle) OUT newcomm copy of comm (handle) OUT request communicator request (handle) C binding 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) :: nequest INTEGER, OPTIONAL, INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_COMM_IDUP(COMM, NEWCOMM, REQUEST, IERROR)	IN	info	info object (handle)
<pre>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_Comm), INTENT(UT) :: newcomm INTEGER, OPTIONAL, INTENT(UUT) :: newcomm INTEGER, OPTIONAL, INTENT(UUT) :: 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 into 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</pre>	OUT	newcomm	copy of comm (handle)
<pre>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_Comm), INTENT(UT) :: newcomm INTEGER, OPTIONAL, INTENT(UUT) :: newcomm INTEGER, OPTIONAL, INTENT(UUT) :: 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 into 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</pre>			
Fortran 2008 biding MPI_Comm_dup_with_info(comm, info, newcomm, ierror) TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Lomm), INTENT(IN) :: info TYPE(MPI_Comm), INTENT(UT) :: ierror Fortran binding MPI_COMM_DUP_WITH_INFO(COMM, INFO, NEWCOMM, IERROR) INTEGER COMM, INFO, NEWCOMM, IERROR 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 revocomm. Rationale. It is expected that some hints will only be valid at communicator receation 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 NMPI_COMM_input communicator (handle) OUT newcomm copy of comm (handle) OUT request communication request (handle) Fortran 2008 binding MPI_Comm_idup(Comm_newcomm, request, ierror) TYPE(MPI_Comm), INTENT(UT) :: comm TYPE(MPI_Comm), INTENT(UT) :: comm TYPE(MPI_Comm_idup(comm, newcomm, request, ierror) TYPE(MPI_Request), INTENT(OUT) :: ierror	C bindiı	ng	
<pre>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) :: newcomm, IERROR) INTEGER COMM, INFO, NEWCOMM, IERROR 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</pre>	int MPI_	Comm_dup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm)
<pre>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(UT) :: inewcomm INTEGER, OPTIONAL, INTENT(OUT) :: newcomm INTEGER COMM, INFO, NEWCOMM, IERROR) INTEGER COMM, INFO, NEWCOMM, IERROR 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</pre>	Fortran	2008 binding	
TYPE(MPI_Info), INTENT(IN) :: info TYPE(MPI_Comm), INTENT(CUT) :: newcomm INTEGER, OPTIONAL, INTENT(CUT) :: 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. <i>Rationale.</i> It is expected that some hints will only be valid at communicator reaction 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. (<i>End of rationale.</i>) MPI_COMM_IDUP(comm, newcomm, request) IN comm communicator (handle) OUT newcomm copy of comm (handle) OUT request communication request (handle) OUT request streagest, ierror) 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_Comm), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_COMM_IDUP(COMM, NEWCOMM, REQUEST, IERROR)		•	, info, newcomm, ierror)
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. <i>Rationale.</i> 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. (<i>End of rationale.</i>) MPI_COMM_IDUP(comm, newcomm, request) IN comm communicator (handle) OUT newcomm copy of comm (handle) OUT request communication request (handle) OUT request communication request (handle) 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_Comm), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_COMM_IDUP(COMM, NEWCOMM, REQUEST, IERROR)			
<pre>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 computed (handle) OUT newcomm copy of comm (handle) OUT request communication request (handle) OUT request Fortran 2008 binding MPI_COMM_INTENT(IN) :: comm TYPE(MPI_Comm, newcomm, request, ierror) TYPE(MPI_Comm, INTENT(IN) :: comm TYPE(MPI_Comm, INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_COMM_IDUP(COMM, NEWCOMM, REQUEST, IERROR)</pre>			
<pre>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.)</pre>			
<pre>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, INTENT(IN) :: comm TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Comm), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_COMM_IDUP(COMM, NEWCOMM, REQUEST, IERROR)</pre>	TNJF	GER, UPTIONAL, INTE	NI(UUI) :: lerror
<pre>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.)</pre>		0	
<pre>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.)</pre> MPI_COMM_IDUP(comm, newcomm, request) IN comm communicator (handle) OUT newcomm copy of comm (handle) OUT request communication request (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_Comm), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror	_		
<pre>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)</pre>	INTE	GER COMM, INFO, NEW	COMM, IERROR
<pre>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.)</pre> MPI_COMM_IDUP(comm, newcomm, request) IN comm communicator (handle) OUT newcomm copy of comm (handle) OUT request communication request (handle) OUT request communication request (handle) Fortran 2008 binding 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	MPI_	_COMM_DUP_WITH_II	NFO behaves exactly as MPI_COMM_DUP except that the
<pre>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) 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)</pre>	hints prov	vided by the argument i	nfo are associated with the output communicator newcomm.
<pre>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) 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)</pre>	D	······	
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. (<i>End of rationale.</i>) 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)		-	
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)			
<pre>MPI_COMM_IDUP(comm, newcomm, request) IN comm communicator (handle) OUT newcomm copy of comm (handle) OUT request communication request (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)</pre>			· · · · ·
<pre>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)</pre>			
<pre>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)</pre>			
<pre>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)</pre>	MPI CON	MM IDUP(comm. newco	omm. request)
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)		· ·	
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)			
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)		newcomm	copy of comm (handle)
<pre>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)</pre>	OUT	request	communication request (handle)
<pre>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)</pre>			
<pre>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)</pre>		U	
<pre>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)</pre>	int MPI_	Comm_idup(MP1_Comm	comm, MP1_Comm *newcomm, MP1_Request *request)
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)			
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)		-	-
TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_COMM_IDUP(COMM, NEWCOMM, REQUEST, IERROR)		-	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_COMM_IDUP(COMM, NEWCOMM, REQUEST, IERROR)			
Fortran binding MPI_COMM_IDUP(COMM, NEWCOMM, REQUEST, IERROR)		-	-
MPI_COMM_IDUP(COMM, NEWCOMM, REQUEST, IERROR)			
		0	
INIEGER OUMM, NEWOUMM, REQUEDI, IERRUK		-	
		GER CUMM, NEWCUMM,	REQUEDI, LERRUR

1	MPI_	COMM_IDUP is a	nonblocking vari	iant of MPI_CON	MM_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	-	ons for nonblockin			ion 6.12) apply	to
6		MM_IDUP and the	-			
7		erroneous to use t				to other MPI
8 9	functions	before the MPI_C	OMM_IDUP oper	ation completes.		
9 10						
11	MPI_CON	MM_IDUP_WITH_	INFO(comm, info,	newcomm, requ	est)	
12	IN	comm	con	nmunicator (hand	le)	
13 14	IN	info	info	o object (handle)		
15	OUT	newcomm	cop	y of comm (handl	le)	
16	OUT	request	con	nmunication reque	est (handle)	
17						
18	C bindir	ıg				
19	int MPI_	Comm_idup_with_	info(MPI_Comm	comm, MPI_Info) info,	
20 21		MPI_Comm *	newcomm, MPI_R	equest *reque	st)	
21	Fortran	2008 binding				
23		_idup_with_info	(comm, info, n	ewcomm, reques	st, ierror)	
24	TYPE(MPI_Comm), INTENT(IN) :: comm					
25	TYPE(MPI_Info), INTENT(IN) :: info					
26	TYPE	(MPI_Comm), INT	ENT(OUT), ASYN	CHRONOUS :: ne	ewcomm	
27	TYPE	(MPI_Request),	INTENT(OUT) ::	request		
28	INTE	GER, OPTIONAL,	INTENT(OUT) ::	ierror		
29	Fortran	binding)*		
30		_IDUP_WITH_INFO	(COMM, INFO, N	EWCOMM, REQUES	ST, IERROR)	
31 32	INTE	GER COMM, INFO,	NEWCOMM, REQU	EST, IERROR		
33	MPI	.COMM_IDUP_WI	TH INFO is a n	onblocking varia	ant of	
34		M_DUP_WITH_I				navior, the se-
35		f MPI_COMM_ID				
36	executed a	at the time that M	PI_COMM_IDUP_	_WITH_INFO is a	called. For exam	ple, attributes
37	or info hi	nts changed after	MPI_COMM_IDU	IP_WITH_INFO	will not be copi	ed to the new
38		cator. All restricti	*			- (
39		12) apply to MPI			-	•
40		erroneous to use t				to other MPI
41	functions	before the MPI_C	OMM_IDUP_WIT	H_INFO operati	ion completes.	
42	Dat	ionale. The MP	_COMM_IDUP a	nd MPL COMM		NEO functions
43		crucial for the de				
44 45		onale.)	velopment of pu	iony nonoioeking	, потапер (рес [(Dina O)
45	1 000					
40						
48						

IN	comm	communicator (handle)		
IN	group	group, which is a subset of the group of comm (handle)		
OUT	newcomm	new communicator (handle)		

MPI_COMM_CREATE(comm, group, newcomm)

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 process must call MPI_COMM_CREATE with a group argument that is a subgroup of the group associated with comm; this could be MPI_GROUP_EMPTY. The processes may specify different values for the group argument. If a process calls with a non-empty group then all processes in that group must call the function with the same group as argument, that is the same processes in the same order. Otherwise, the call is erroneous. This implies that the set of groups specified across the processes must be disjoint. If the calling process is a member of the group given as group argument, then newcomm is a communicator with group as its associated group. In the case that a process calls with a group to which it does not belong, e.g., MPI_GROUP_EMPTY, then MPI_COMM_NULL is returned as newcomm. The function is collective and must be called by all processes in the group of comm.

Rationale. The interface supports the original mechanism from MPI-1.1, which required the same group in all processes of comm. It was extended in MPI-2.2 to allow the use of disjoint subgroups in order to allow implementations to eliminate unnecessary communication that MPI_COMM_SPLIT would incur when the user already knows the membership of the disjoint subgroups. (*End of rationale.*)

Rationale. The requirement that the entire group of **comm** participate in the call stems from the following considerations:

- It allows the implementation to layer MPI_COMM_CREATE on top of regular collective communications.
- It provides additional safety, in particular in the case where partially overlapping groups are used to create new communicators.

Unofficial Draft for Comment Only

 24

 31

1

2

3

4 5

6

7

8

9

10

11

15

16

17

18

19

20

25

• It permits implementations to sometimes avoid communication related to context creation.

(End of rationale.)

Advice to users. MPI_COMM_CREATE provides a means to subset a group of processes for the purpose of separate MIMD computation, with separate communication space. newcomm, which emerges from MPI_COMM_CREATE, can be used in subsequent calls to MPI_COMM_CREATE (or other communicator constructors) to further subdivide a computation into parallel sub-computations. A more general service is provided by MPI_COMM_SPLIT, below. (*End of advice to users.*)

Advice to implementors. When calling MPI_COMM_DUP, all processes call with the same group (the group associated with the communicator). When calling

MPI_COMM_CREATE, the processes provide the same group or disjoint subgroups. For both calls, it is theoretically possible to agree on a group-wide unique context with no communication. However, local execution of these functions requires use of a larger context name space and reduces error checking. Implementations may strike various compromises between these conflicting goals, such as bulk allocation of multiple contexts in one collective operation.

Important: If new communicators are created without synchronizing the processes involved then the communication system must be able to cope with messages arriving in a context that has not yet been allocated at the receiving process. (*End of advice to implementors.*)

If comm is an inter-communicator, then the output communicator is also an inter-com-26municator where the local group consists only of those processes contained in group (see 27Figure 7.1). The group argument should only contain those processes in the local group of 28the input inter-communicator that are to be a part of newcomm. All processes in the same 29 local group of comm must specify the same value for group, i.e., the same members in the 30 same order. If either group does not specify at least one process in the local group of the 31 inter-communicator, or if the calling process is not included in the group, MPI_COMM_NULL 32 is returned. 33

Rationale. In the case where either the left or right group is empty, a null communicator is returned instead of an inter-communicator with MPI_GROUP_EMPTY because the side with the empty group must return MPI_COMM_NULL. (*End of rationale.*)

37 38

34

35

36

³⁹ **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.

- 43
- 45
- 46
- 47
- 48

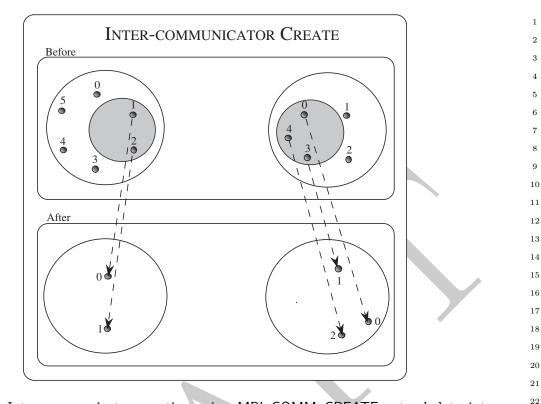


Figure 7.1: Inter-communicator creation using MPI_COMM_CREATE extended to intercommunicators. The input groups are those in the grey circle.

```
MPI_Comm inter_comm, new_inter_comm;
MPI_Group local_group, group;
          rank = 0; /* rank on left side to include in
int
                       new inter-comm */
/* Construct the original inter-communicator: "inter_comm" */
. . .
/* Construct the group of processes to be in new
   inter-communicator */
if (/* I'm on the left side of the inter-communicator */) {
  MPI_Comm_group(inter_comm, &local_group);
  MPI_Group_incl(local_group, 1, &rank, &group);
  MPI_Group_free(&local_group);
}
else
  MPI_Comm_group(inter_comm, &group);
MPI_Comm_create(inter_comm, group, &new_inter_comm);
MPI_Group_free(&group);
```

23

 $\frac{24}{25}$

26

27

28

29 30

 31

32 33

34

35

36

37

38

39

40

41

42 43

44

330 CHAPTER 7. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

1 MPI_COMM_CREATE_GROUP(comm, group, tag, newcomm) 2 IN intra-communicator (handle) comm 3 IN group, which is a subset of the group of comm group 4 (handle) 56 IN tag (integer) tag 7 OUT newcomm new communicator (handle) 8 9 C binding 10 int MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag, 11 MPI_Comm *newcomm) 1213Fortran 2008 binding 14MPI_Comm_create_group(comm, group, tag, newcomm, ierror) 15TYPE(MPI_Comm), INTENT(IN) :: comm 16TYPE(MPI_Group), INTENT(IN) :: group 17INTEGER, INTENT(IN) :: tag 18 TYPE(MPI_Comm), INTENT(OUT) :: newcomm 19INTEGER, OPTIONAL, INTENT(OUT) :: ierror 20Fortran binding 21MPI_COMM_CREATE_GROUP(COMM, GROUP, TAG, NEWCOMM, IERROR) 22 INTEGER COMM, GROUP, TAG, NEWCOMM, IERROR 23 24 MPI_COMM_CREATE_GROUP is similar to MPI_COMM_CREATE; however, 25MPI_COMM_CREATE must be called by all processes in the group of comm, whereas 26MPI_COMM_CREATE_GROUP must be called by all processes in group, which is a subgroup 27of the group of comm. In addition, MPI_COMM_CREATE_GROUP requires that comm is 28an intra-communicator. MPI_COMM_CREATE_GROUP returns a new intra-communicator, 29newcomm, for which the group argument defines the communication group. No cached 30 information propagates from comm to newcomm and no virtual topology information is 31 added to the created communicator. Each process must provide a group argument that is a 32 subgroup of the group associated with comm; this could be MPI_GROUP_EMPTY. If a non-33 empty group is specified, then all processes in that group must call the function, and each of 34these processes must provide the same arguments, including a group that contains the same 35 members with the same ordering. Otherwise the call is erroneous. If the calling process is a 36 member of the group given as the group argument, then newcomm is a communicator with 37 group as its associated group. If the calling process is not a member of group, e.g., group is 38 MPI_GROUP_EMPTY, then the call is a local operation and MPI_COMM_NULL is returned as 39 newcomm. 40 41 Functionality similar to MPI_COMM_CREATE_GROUP can be imple-Rationale. 42mented through repeated MPI_INTERCOMM_CREATE and MPI_INTERCOMM_MERGE calls that start with the MPI_COMM_SELF communicators 43 44 at each process in group and build up an intra-communicator with group group [17]. 45Such an algorithm requires the creation of many intermediate communicators;

⁴⁶ MPI_COMM_CREATE_GROUP can provide a more efficient implementation that avoids ⁴⁷ this overhead. (*End of rationale.*)

Advice to users. An inter-communicator can be created collectively over processes in the union of the local and remote groups by creating the local communicator using MPI_COMM_CREATE_GROUP and using that communicator as the local communicator argument to MPI_INTERCOMM_CREATE. (*End of advice to users.*)

The tag argument does not conflict with tags used in point-to-point communication and is not permitted to be a wildcard. If multiple threads at a given process perform concurrent MPI_COMM_CREATE_GROUP operations, the user must distinguish these operations by providing different tag or comm arguments.

Advice to users. MPI_COMM_CREATE may provide lower overhead than MPI_COMM_CREATE_GROUP because it can take advantage of collective communication on comm when constructing newcomm. (*End of advice to users.*)

MPI_COMM_SPLIT(comm, color, key, newcomm)

IN	comm	communicator (handle)
IN	color	control of subset assignment (integer)
IN	key	control of rank assignment (integer)
OUT	newcomm	new communicator (handle)

C binding

int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)

Fortran 2008 binding

<pre>MPI_Comm_split(comm, color, key, newcomm, ierror)</pre>
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 processes of the same color. Within each subgroup, the processes are ranked in the order defined by the value of the argument key, with ties broken according to their rank in the old group. A new communicator is created for each subgroup and returned in newcomm. A process may supply the color value MPI_UNDEFINED, in which case newcomm returns MPI_COMM_NULL. This is a collective call, but each process is permitted to provide different values for color and key. No cached information propagates from comm to newcomm and no virtual topology information is added to the created communicators.

With an intra-communicator comm, a call to MPI_COMM_CREATE(comm, group, newcomm) is equivalent to a call to MPI_COMM_SPLIT(comm, color, key, newcomm), where processes that are members of their group argument provide color = number of the group 45 46 47 48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4 5

6

 $\overline{7}$

8

9 10

11

12

 24

25 26

27

33

34

35 36

37

38

39

40

41

42

43

1 (based on a unique numbering of all disjoint groups) and key = rank in group, and all $\mathbf{2}$ processes that are not members of their group argument provide color = MPI_UNDEFINED. 3 The value of color must be non-negative or MPI_UNDEFINED.

Advice to users. This is an extremely powerful mechanism for dividing a single communicating group of processes into k subgroups, with k chosen implicitly by the user (by the number of colors asserted over all the processes). Each resulting communicator will be non-overlapping. Such a division could be useful for defining a hierarchy of computations, such as for multigrid, or linear algebra. For intra-communicators, MPI_COMM_SPLIT provides similar capability as MPI_COMM_CREATE to split a 10 communicating group into disjoint subgroups. MPI_COMM_SPLIT is useful when some processes do not have complete information of the other members in their 12group, but all processes know (the color of) the group to which they belong. In 13 this case, the MPI implementation discovers the other group members via communi-14cation. MPI_COMM_CREATE is useful when all processes have complete information 15of the members of their group. In this case, MPI can avoid the extra communica-16tion required to discover group membership. MPI_COMM_CREATE_GROUP is useful 17 when all processes in a given group have complete information of the members of their 18 19 group and synchronization with processes outside the group can be avoided.

20Multiple calls to MPI_COMM_SPLIT can be used to overcome the requirement that 21any call have no overlap of the resulting communicators (each process is of only one 22color per call). In this way, multiple overlapping communication structures can be 23created. Creative use of the color and key in such splitting operations is encouraged.

24Note that, for a fixed color, the keys need not be unique. It is MPI_COMM_SPLIT's 25responsibility to sort processes in ascending order according to this key, and to break 26ties in a consistent way. If all the keys are specified in the same way, then all the 27processes in a given color will have the relative rank order as they did in their parent 28 group. 29

Essentially, making the key value zero for all processes of a given color means that one 30 does not really care about the rank-order of the processes in the new communicator. 31(End of advice to users.) 32

Rationale. color is restricted to be non-negative, so as not to conflict with the value assigned to MPI_UNDEFINED. (End of rationale.)

The result of MPI_COMM_SPLIT on an inter-communicator is that those processes on the 37 left with the same color as those processes on the right combine to create a new inter-38 communicator. The key argument describes the relative rank of processes on each side of 39 the inter-communicator (see Figure 7.2). For those colors that are specified only on one 40 side of the inter-communicator, MPI_COMM_NULL is returned. MPI_COMM_NULL is also 41 returned to those processes that specify MPI_UNDEFINED as the color. 42

Advice to users. For inter-communicators, MPI_COMM_SPLIT is more general than MPI_COMM_CREATE. A single call to MPI_COMM_SPLIT can create a set of disjoint inter-communicators, while a call to MPI_COMM_CREATE creates only one. (End of advice to users.)

47 48

43

44

45

46

33

34

35 36

4

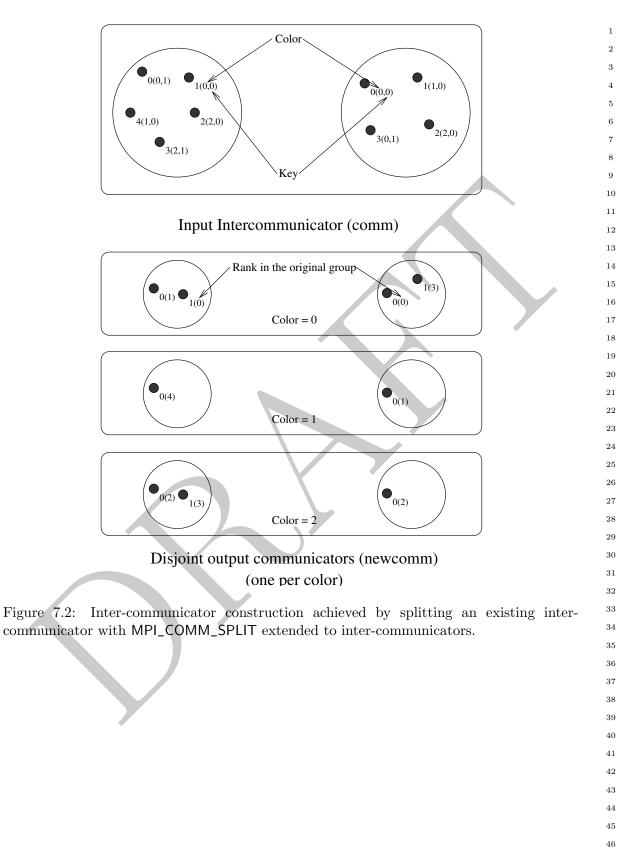
5

6

7

8

9



Unofficial Draft for Comment Only

 $47 \\ 48$

334 CHAPTER 7. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

```
1
     Example 7.2 Parallel client-server model.
\mathbf{2}
     The following client code illustrates how clients on the left side of an inter-communicator
3
     could be assigned to a single server from a pool of servers on the right side of an inter-
4
     communicator.
5
              /* Client code */
6
              MPI_Comm multiple_server_comm;
7
              MPI_Comm
                         single_server_comm;
8
              int
                         color, rank, num_servers;
9
10
              /* Create inter-communicator with clients and servers:
11
                 multiple_server_comm */
12
              . . .
13
14
              /* Find out the number of servers available */
15
              MPI_Comm_remote_size(multiple_server_comm, &num_servers);
16
17
              /* Determine my color */
18
              MPI_Comm_rank(multiple_server_comm, &rank);
19
              color = rank % num_servers;
20
21
              /* Split the inter-communicator */
22
              MPI_Comm_split(multiple_server_comm, color, rank,
23
                               &single_server_comm);
24
25
     The following is the corresponding server code:
26
27
              /* Server code */
28
              MPI_Comm multiple_client_comm;
29
                         single_server_comm;
              MPI_Comm
30
              int
                         rank;
31
32
              /* Create inter-communicator with clients and servers:
33
                 multiple_client_comm */
34
                . .
35
36
              /* Split the inter-communicator for a single server per group
37
                 of clients */
38
              MPI_Comm_rank(multiple_client_comm, &rank);
39
              MPI_Comm_split(multiple_client_comm, rank, 0,
40
                               &single_server_comm);
41
42
43
44
45
46
47
48
```

N	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

Fortran 2008 binding

MPI_Comm_split_type(comm, split_type, key, info, newcomm, ierror)
 TYPE(MPI_Comm), INTENT(IN) :: comm
 INTEGER, INTENT(IN) :: split_type, key
 TYPE(MPI_Info), INTENT(IN) :: info
 TYPE(MPI_Comm), INTENT(OUT) :: newcomm
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

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

Unofficial Draft for Comment Only

 31

1 cache) specified by the "mpi_hw_resource_type" info key. Each output communicator 2 newcomm corresponds to a single instance of the specified hardware resource type. 3 The MPI processes in the group associated with the output communicator newcomm 4 utilize that specific hardware resource type instance, and no other instance of the 5same hardware resource type. 6 If an MPI process does not meet the above criteria, then MPI_COMM_NULL is returned 7 in newcomm for such process. 8 MPI_COMM_NULL is also returned in **newcomm** in the following cases: 9 10 • No info key is provided. 11 • The info handle does not include the key "mpi_hw_resource_type". 12• The MPI implementation neither recognizes nor supports the info key 13 14"mpi_hw_resource_type". 15• The MPI implementation does not recognize the value associated with the info 16key "mpi_hw_resource_type". 17 The MPI implementation will return in the group of the output communicator 18 newcomm the largest subset of MPI processes that match the splitting criterion. 19 20The processes in the group associated with newcomm are ranked in the order defined 21by the value of the argument key with ties broken according to their rank in the group 22 associated with comm. 2324Advice to users. The set of hardware resources that an MPI process is able to 25utilize may change during the application execution (e.g., because of the reloca-26tion of an MPI process), in which case the communicators created with the value 27MPI_COMM_TYPE_HW_GUIDED before this change may not reflect the utiliza-28 tion of hardware resources of such process at any time after the communicator 29 creation. (End of advice to users.) 30 31The user explicitly constrains with the info argument the splitting of the input com-32 municator comm. To this end, the info key "mpi_hw_resource_type" is reserved and 33 its associated value is an implementation-defined string designating the type of the 34 requested hardware resource (e.g., "NUMANode", "Package" or "L3Cache"). 35The value "mpi_shared_memory" is reserved and its use is equivalent to using 36 MPI_COMM_TYPE_SHARED for the split_type parameter. 37 38 *Rationale.* The value "mpi_shared_memory" is defined in order to ensure consis-39 tency between the use of MPI_COMM_TYPE_SHARED and the use of 40 MPI_COMM_TYPE_HW_GUIDED. (*End of rationale.*) 41 All MPI processes must provide the same value for the info key "mpi_hw_resource_type". 4243 MPI_COMM_TYPE_HW_UNGUIDED—the group of MPI processes associated with newcomm 44must be a *strict* subset of the group associated with comm and each 45newcomm corresponds to a single instance of a hardware resource type (for example 46a computing core or an L3 cache). 47 48

Example 7.3 Splitting MPI_COMM_WORLD into NUMANode subcommunicators.

All MPI processes in the group associated with comm which utilize that specific hardware resource type instance—and no other instance of the same hardware resource type—are included in the group of newcomm.

If a given MPI process cannot be a member of a communicator that forms such a strict subset, or does not meet the above criteria, then MPI_COMM_NULL is returned in newcomm for this process.

Advice to implementors. In a high-quality MPI implementation, the number of different new valid communicators **newcomm** produced by this splitting operation should be minimal unless the user provides a key/value pair that modifies this behavior. The sets of hardware resource types used for the splitting operation are implementation-dependent, but should reflect the hardware of the actual system on which the application is currently executing. (*End of advice to implementors.*)

Rationale. If the hardware resources are hierarchically organized, calling this routine several times using as its input communicator **comm** the output communicator **newcomm** of the previous call creates a sequence of **newcomm** communicators in each MPI process, which exposes a hierarchical view of the hardware platform, as shown in Example 7.4. This sequence of returned **newcomm** communicators may differ from the sets of hardware resource types, as shown in the second splitting operation in Figure 7.3. (*End of rationale.*)

Advice to users. Each output communicator newcomm can represent a different hardware resource type (see Figure 7.3 for an example). The set of hardware resources an MPI process utilizes may change during the application execution (e.g., because of process relocation), in which case the communicators created with the value MPI_COMM_TYPE_HW_UNGUIDED before this change may not reflect the utilization of hardware resources for such process at any time after the communicator creation. (*End of advice to users.*)

If a valid info handle is provided as an argument, the MPI implementation sets the ⁴⁶ info key "mpi_hw_resource_type" for each MPI process in the group associated with a ⁴⁷ returned newcomm communicator and the info key value is an implementation-defined ⁴⁸

Unofficial Draft for Comment Only

 24

 31

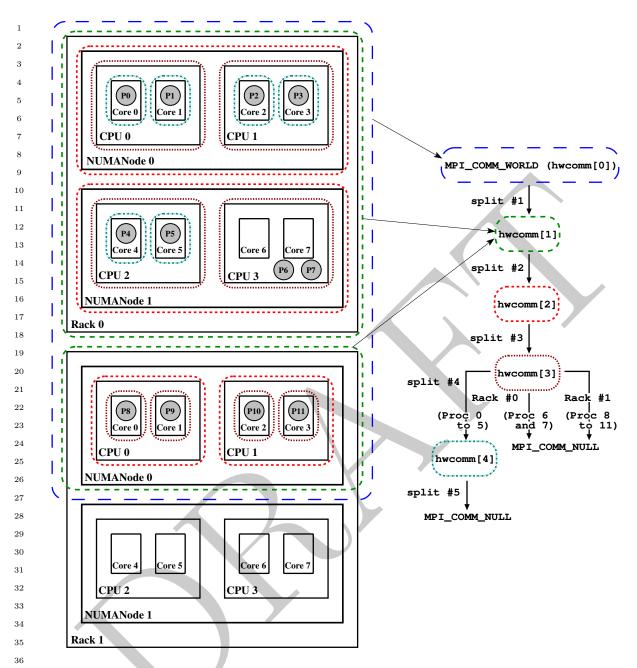


Figure 7.3: Recursive splitting of MPI_COMM_WORLD with MPI_COMM_SPLIT_TYPE and 37 MPI_COMM_TYPE_HW_UNGUIDED. Dashed lines represent communicators whilst solid lines 38 represent hardware resources. MPI processes (P0 to P11) utilize exclusively their respective 39 core, except for P6 and P7 which utilize CPU #3 of Rack #0 and can therefore use Cores 40 #6 and #7 indifferently. The second splitting operation yields two subcommunicators 41 corresponding to NUMANodes in Rack #0 and to CPUs in Rack #1 because Rack #1 42features only one NUMANode, which corresponds to the whole portion of the Rack that 43 is included in MPI_COMM_WORLD and hwcomm[1]. For the first splitting operation, the 44hardware resource type returned in the info argument is "Rack" on the processes on Rack 45#0, whereas on Rack #1, it can be either "Rack" or "NUMANode". 46

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.

Advice to implementors. Implementations can define their own split_type values, or use the info argument, to assist in creating communicators that help expose platformspecific information to the application. The concept of hardware-based communicators was first described by Träff [67] for SMP systems. Guided and unguided modes description as well as an implementation path are introduced by Goglin *et al.* [27]. (*End of advice to implementors.*)

			34
MPI_COM	M_CREATE_FROM_GROUP(group, stringtag, info, errhandler, newcomm)	35
IN	group	group (handle)	36
IN	stringtag	unique identifier for this operation (string)	37
IN	info	info object (handle)	38 39
IN	errhandler	error handler to be attached to new	40
IIN	ermanuler	intra-communicator (handle)	41
OUT	newcomm	new communicator (handle)	42
001			43
C binding			45

 $\mathbf{2}$

1 Fortran 2008 binding $\mathbf{2}$ MPI_Comm_create_from_group(group, stringtag, info, errhandler, newcomm, 3 ierror) 4 TYPE(MPI_Group), INTENT(IN) :: group 5CHARACTER(LEN=*), INTENT(IN) :: stringtag 6 TYPE(MPI_Info), INTENT(IN) :: info 7 TYPE(MPI_Errhandler), INTENT(IN) :: errhandler 8 TYPE(MPI_Comm), INTENT(OUT) :: newcomm 9 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 10 Fortran binding 11 MPI_COMM_CREATE_FROM_GROUP(GROUP, STRINGTAG, INFO, ERRHANDLER, NEWCOMM, 12IERROR) 13 INTEGER GROUP, INFO, ERRHANDLER, NEWCOMM, IERROR 14CHARACTER*(*) STRINGTAG 1516MPI_COMM_CREATE_FROM_GROUP is similar to MPI_COMM_CREATE_GROUP, ex-17cept that the set of MPI processes involved in the creation of the new intra-communicator 18is specified by a group argument, rather than the group associated with a pre-existing com-19municator. If a non-empty group is specified, then all MPI processes in that group must call 20the function and each of these MPI processes must provide the same arguments, including 21a group that contains the same members with the same ordering, and identical stringtag 22value. In the event that MPI_GROUP_EMPTY is supplied as the group argument, then the 23call is a local operation and MPI_COMM_NULL is returned as newcomm. The stringtag argu- 24 ment is analogous to the tag used for MPI_COMM_CREATE_GROUP. If multiple threads at 25a given MPI process perform concurrent MPI_COMM_CREATE_FROM_GROUP operations, 26the user must distinguish these operations by providing different stringtag arguments. The 27stringtag shall not exceed MPI_MAX_FROM_GROUP_TAG characters in length. For C, this 28includes space for a null terminating character. The errhandler argument specifies an error 29handler to be attached to the new intra-communicator. This error handler will also be in-30 voked if the MPI_COMM_CREATE_FROM_GROUP function encounters an error. The info

31 32

33 34

> 35 36

> 37

38

39

Advice to users. The stringtag argument is used to distinguish concurrent communicator construction operations issued by different entities. As such, it is important to ensure that this argument is unique for each concurrent call to

argument provides hints and assertions, possibly MPI implementation dependent, which

indicate desired characteristics and guide communicator creation.

MPI_COMM_CREATE_FROM_GROUP. Reverse domain name notation convention [1] is one approach to constructing unique stringtag arguments. See also example 11.8. (*End of advice to users.*)

7.4.3 Communicator Destructors

MPI_COMM_FREE(comm)

INOUT comm communicator to be destroyed (handle)

C binding

int MPI_Comm_free(MPI_Comm *comm)

Fortran 2008 binding

```
MPI_Comm_free(comm, ierror)
    TYPE(MPI_Comm), INTENT(INOUT) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

MPI_COMM_FREE(COMM, IERROR) INTEGER COMM, IERROR

This collective operation marks the communication object for deallocation. The handle is set to MPI_COMM_NULL. Any pending operations that use this communicator will complete normally; the object is actually deallocated only if there are no other active references to it. This call applies to intra- and inter-communicators. The delete callback functions for all cached attributes (see Section 7.7) are called in arbitrary order.

Advice to implementors. Though collective, it is anticipated that this operation will normally be implemented to be local, though a debugging version of an MPI library might choose to synchronize. (*End of advice to implementors.*)

7.4.4 Communicator Info

Hints specified via info (see Chapter 10) allow a user to provide information to direct optimization. Providing hints may enable an implementation to deliver increased performance or minimize use of system resources. An implementation is free to ignore all hints; however, applications must comply with any info hints they provide that are used by the MPI implementation (i.e., are returned by a call to MPI_COMM_GET_INFO) and that place a restriction on the behavior of the application. Hints are specified on a per communicator basis, in MPI_COMM_DUP_WITH_INFO, MPI_COMM_IDUP_WITH_INFO, MPI_COMM_SET_INFO, MPI_COMM_SET_INFO, MPI_COMM_SET_INFO, MPI_COMM_SET_ADJACENT, via the opaque info object. When an info object that specifies a subset of valid hints is passed to MPI_COMM_SET_INFO, there will be no effect on previously set or defaulted hints that the info does not specify.

Advice to implementors. It may happen that a program is coded with hints for one system, and later executes on another system that does not support these hints. In general, unsupported hints should simply be ignored. Needless to say, no hint can be mandatory. However, for each hint used by a specific implementation, a default value must be provided when the user does not specify a value for this hint. (*End of advice to implementors.*)

Unofficial Draft for Comment Only

 31

Info hints are not propagated by MPI from one communicator to another. The following info keys are valid for all communicators.

"mpi_assert_no_any_tag" (boolean, default: "false"): If set to "true", then the implementation may assume that the process will not use the MPI_ANY_TAG wildcard on the given communicator.

- "mpi_assert_no_any_source" (boolean, default: "false"): If set to "true", then the implementation may assume that the process will not use the MPI_ANY_SOURCE wildcard on the given communicator.
- "mpi_assert_exact_length" (boolean, default: "false"): If set to "true", then the implementation may assume that the lengths of messages received by the process are equal to the lengths of the corresponding receive buffers, for point-to-point communication operations on the given communicator.

"mpi_assert_allow_overtaking" (boolean, default: "false"): If set to "true", then the implementation may assume that point-to-point communications on the given communicator do not rely on the non-overtaking rule specified in Section 3.5. In other words, the application asserts that send operations are not required to be matched at the receiver in the order in which the send operations were posted by the sender, and receive operations are not required to be matched in the order in which they were posted by the receiver.

Advice to users. Use of the "mpi_assert_allow_overtaking" info key can result in nondeterminism in the message matching order. (End of advice to users.)

Advice to users. Some optimizations may only be possible when all processes in the group of the communicator provide a given info key with the same value. (End of advice to users.)

MPI_COMM_SET_INFO(comm, info)

1

 $\mathbf{2}$

3 4

5 6

7

8

9

10 11

12

13

14

15

23

24

25 26

27

28

33	
34	INOUT comm communicator (handle)
35	IN info info object (handle)
36	
37	C binding
38	<pre>int MPI_Comm_set_info(MPI_Comm comm, MPI_Info info)</pre>
39	
40	Fortran 2008 binding
41	<pre>MPI_Comm_set_info(comm, info, ierror)</pre>
42	TYPE(MPI_Comm), INTENT(IN) :: comm
43	TYPE(MPI_Info), INTENT(IN) :: info
44	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45	Fortran binding
10	Forman Shang

```
    <sup>46</sup> MPI_COMM_SET_INFO(COMM, INFO, IERROR)
    <sup>47</sup> INTEGER COMM, INFO, IERROR
```

MPI_COMM_SET_INFO updates the hints of the communicator associated with comm using the hints provided in info. This operation has no effect on previously set or defaulted hints that are not specified by info. It also has no effect on previously set or defaulted hints that are specified by info, but are ignored by the MPI implementation in this call to MPI_COMM_SET_INFO. MPI_COMM_SET_INFO is a collective routine. The info object may be different on each process, but any info entries that an implementation requires to be the same on all processes must appear with the same value in each process's info object.

Advice to users. Some info items that an implementation can use when it creates a communicator cannot easily be changed once the communicator has been created. Thus, an implementation may ignore hints issued in this call that it would have accepted in a creation call. An implementation may also be unable to update certain info hints in a call to MPI_COMM_SET_INFO. MPI_COMM_GET_INFO can be used to determine whether updates to existing info hints were ignored by the implementation. (*End of advice to users.*)

Advice to users. Setting info hints on the predefined communicators MPI_COMM_WORLD and MPI_COMM_SELF may have unintended effects, as changes to these global objects may affect all components of the application, including libraries and tools. Users must ensure that all components of the application that use a given communicator, including libraries and tools, can comply with any info hints associated with that communicator. (*End of advice to users.*)

MPI_COMM_GET_INFO(comm, info_used)

IN	comm	communicator object (handle)				
OUT	info_used	new info object (handle)				
C binding	;					
int MPI_C	<pre>int MPI_Comm_get_info(MPI_Comm comm, MPI_Info *info_used)</pre>					
Fortran 2008 binding						
MPI_Comm_	MPI_Comm_get_info(comm, info_used, ierror)					
TYPE(I	MPI_Comm), INTENT(IN) ::	comm				
TYPE(MPI_Info), INTENT(OUT) ::	info_used				

Fortran binding

MPI_COMM_GET_INFO(COMM, INFO_USED, IERROR) INTEGER COMM, INFO_USED, IERROR

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_COMM_GET_INFO returns a new info object containing the hints of the commu-nicator associated with comm. The current setting of all hints related to this communicator is returned in info_used. An MPI implementation is required to return all hints that are supported by the implementation and have default values specified; any user-supplied hints that were not ignored by the implementation; and any additional hints that were set by the implementation. If no such hints exist, a handle to a newly created info object is re-turned that contains no key/value pair. The user is responsible for freeing info_used via MPI_INFO_FREE.

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\overline{7}$

```
7.5
            Motivating Examples
1
\mathbf{2}
     7.5.1 Current Practice #1
3
4
     Example #1a:
5
         int main(int argc, char *argv[])
6
         {
7
           int me, size;
8
9
           . . .
           MPI_Init(&argc, &argv);
10
           MPI_Comm_rank(MPI_COMM_WORLD, &me);
11
           MPI_Comm_size(MPI_COMM_WORLD, &size);
12
13
           (void)printf("Process %d size %d\n", me, size);
14
15
           MPI_Finalize();
16
           return 0;
17
         }
18
19
     Example \#1a is a do-nothing program that initializes itself, and refers to the "all" commu-
20
     nicator, and prints a message. It terminates itself too. This example does not imply that
21
     MPI supports printf-like communication itself.
22
     Example #1b: Message exchange (supposing that size is even)
23
          int main(int argc, char *argv[])
24
          {
25
             int me, size;
26
             int SOME_TAG = 0;
27
             . . .
28
             MPI_Init(&argc, &argv);
29
30
             MPI_Comm_rank(MPI_COMM_WORLD, &me);
                                                         /* local */
31
             MPI_Comm_size(MPI_COMM_WORLD, &size); /* local */
32
33
             if((me % 2) == 0)
34
             ſ
35
                 /* send unless highest-numbered process */
36
                 if((me + 1) < size)
37
                    MPI_Send(..., me + 1, SOME_TAG, MPI_COMM_WORLD);
38
             }
39
             else
40
                 MPI_Recv(..., me - 1, SOME_TAG, MPI_COMM_WORLD, &status);
41
42
             . . .
43
             MPI_Finalize();
44
             return 0;
45
          }
46
47
```

Example #1b schematically illustrates message exchanges between "even" and "odd" pro cesses in the "all" communicator.

Unofficial Draft for Comment Only

. . .

```
7.5.2 Current Practice #2
                                                                                         1
                                                                                         2
   int main(int argc, char *argv[])
                                                                                         3
   {
                                                                                         4
     int me, count;
                                                                                         5
     void *data;
                                                                                         6
     . . .
                                                                                         7
                                                                                         8
     MPI_Init(&argc, &argv);
                                                                                         9
     MPI_Comm_rank(MPI_COMM_WORLD, &me);
                                                                                        10
                                                                                        11
     if(me == 0)
                                                                                        12
     {
                                                                                        13
          /* get input, create buffer ''data'' */
                                                                                        14
          . . .
                                                                                        15
     }
                                                                                        16
                                                                                        17
     MPI_Bcast(data, count, MPI_BYTE, 0, MPI_COMM_WORLD);
                                                                                        18
                                                                                        19
      . . .
                                                                                        20
     MPI_Finalize();
                                                                                        21
     return 0;
                                                                                        22
   }
                                                                                        23
                                                                                        ^{24}
This example illustrates the use of a collective communication.
                                                                                        25
                                                                                        26
7.5.3 (Approximate) Current Practice #3
                                                                                        27
  int main(int argc, char *argv[])
                                                                                        28
                                                                                        29
  ſ
                                                                                        30
    int me, count, count2;
                                                                                        31
    void *send_buf, *recv_buf, *send_buf2, *recv_buf2;
                                                                                        32
    MPI_Group group_world, grprem;
                                                                                        33
    MPI_Comm commWorker;
                                                                                        34
   static int ranks[] = {0};
                                                                                        35
    . . .
    MPI_Init(&argc, &argv);
                                                                                        36
                                                                                        37
    MPI_Comm_group(MPI_COMM_WORLD, &group_world);
    MPI_Comm_rank(MPI_COMM_WORLD, &me); /* local */
                                                                                        38
                                                                                        39
    MPI_Group_excl(group_world, 1, ranks, &grprem); /* local */
                                                                                        40
                                                                                        41
    MPI_Comm_create(MPI_COMM_WORLD, grprem, &commWorker);
                                                                                        42
    if(me != 0)
                                                                                        43
                                                                                        44
    ſ
      /* compute on worker */
                                                                                        45
                                                                                        46
                                                                                        47
      MPI_Reduce(send_buf,recv_buf,count, MPI_INT, MPI_SUM, 1, commWorker);
```

```
1
            MPI_Comm_free(&commWorker);
\mathbf{2}
          }
3
          /* zero falls through immediately to this reduce, others do later... */
4
          MPI_Reduce(send_buf2, recv_buf2, count2,
5
                      MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
6
7
          MPI_Group_free(&group_world);
8
          MPI_Group_free(&grprem);
9
          MPI_Finalize();
10
          return 0;
11
       }
12
     This example illustrates how a group consisting of all but the zeroth process of the "all"
13
     group is created, and then how a communicator is formed (commWorker) for that new group.
14
     The new communicator is used in a collective call, and all processes execute a collective call
15
     in the MPI_COMM_WORLD context. This example illustrates how the two communicators
16
     (that inherently possess distinct contexts) protect communication. That is, communication
17
     in MPI_COMM_WORLD is insulated from communication in commWorker, and vice versa.
18
          In summary, "group safety" is achieved via communicators because distinct contexts
19
     within communicators are enforced to be unique on any process.
20
21
     7.5.4
            Example #4
22
23
     The following example is meant to illustrate "safety" between point-to-point and collective
24
     communication. MPI guarantees that a single communicator can do safe point-to-point and
25
     collective communication.
26
27
         #define TAG_ARBITRARY 12345
28
         #define SOME_COUNT
                                      50
29
30
         int main(int argc, char *argv[])
31
         {
32
           int me;
33
           MPI_Request request[2];
34
           MPI_Status status[2];
35
           MPI_Group group_world, subgroup;
36
           int ranks[] = {2, 4, 6, 8};
37
           MPI_Comm the_comm;
38
           . . .
39
           MPI_Init(&argc, &argv);
40
           MPI_Comm_group(MPI_COMM_WORLD, &group_world);
41
42
           MPI_Group_incl(group_world, 4, ranks, &subgroup); /* local */
43
           MPI_Group_rank(subgroup, &me);
                                                   /* local */
44
45
           MPI_Comm_create(MPI_COMM_WORLD, subgroup, &the_comm);
46
47
           if(me != MPI_UNDEFINED)
48
           {
```

```
1
          MPI_Irecv(buff1, count, MPI_DOUBLE, MPI_ANY_SOURCE, TAG_ARBITRARY,
                                                                                           \mathbf{2}
                               the_comm, request);
                                                                                           3
          MPI_Isend(buff2, count, MPI_DOUBLE, (me+1)%4, TAG_ARBITRARY,
                               the_comm, request+1);
                                                                                           4
          for(i = 0; i < SOME_COUNT; i++)</pre>
                                                                                           5
                                                                                           6
            MPI_Reduce(..., the_comm);
          MPI_Waitall(2, request, status);
                                                                                           7
                                                                                           8
          MPI_Comm_free(&the_comm);
                                                                                           9
     }
                                                                                           10
                                                                                           11
     MPI_Group_free(&group_world);
                                                                                           12
     MPI_Group_free(&subgroup);
                                                                                           13
                                                                                           14
     MPI_Finalize();
                                                                                           15
     return 0;
                                                                                           16
   }
                                                                                           17
                                                                                           18
7.5.5
      Library Example #1
                                                                                           19
The main program:
                                                                                           20
                                                                                          21
   int main(int argc, char *argv[])
                                                                                          22
   {
                                                                                          23
     int done = 0;
                                                                                           ^{24}
     user_lib_t *libh_a, *libh_b;
                                                                                           25
     void *dataset1, *dataset2;
                                                                                           26
     . . .
                                                                                           27
     MPI_Init(&argc, &argv);
                                                                                           28
     . . .
                                                                                           29
     init_user_lib(MPI_COMM_WORLD, &libh_a);
                                                                                           30
     init_user_lib(MPI_COMM_WORLD, &libh_b);
                                                                                           ^{31}
      . . .
                                                                                           32
     user_start_op(libh_a, dataset1);
                                                                                           33
     user_start_op(libh_b, dataset2);
                                                                                          34
      . . .
                                                                                          35
     while(!done)
                                                                                          36
     {
                                                                                          37
         /* work */
                                                                                           38
         . . .
                                                                                           39
         MPI_Reduce(..., MPI_COMM_WORLD);
                                                                                           40
                                                                                           41
         /* see if done */
                                                                                           42
         . . .
                                                                                           43
     }
                                                                                           44
     user_end_op(libh_a);
                                                                                           45
     user_end_op(libh_b);
                                                                                           46
                                                                                           47
     uninit_user_lib(libh_a);
                                                                                           48
```

```
1
           uninit_user_lib(libh_b);
\mathbf{2}
           MPI_Finalize();
3
           return 0;
4
        }
5
     The user library initialization code:
6
7
        void init_user_lib(MPI_Comm comm, user_lib_t **handle)
8
         {
9
           user_lib_t *save;
10
11
           user_lib_initsave(&save); /* local */
12
           MPI_Comm_dup(comm, &(save->comm));
13
14
           /* other inits */
15
           . . .
16
17
           *handle = save;
18
        }
19
20
     User start-up code:
21
        void user_start_op(user_lib_t *handle, void *data)
22
        {
23
           MPI_Irecv( ..., handle->comm, &(handle->irecv_handle) );
24
           MPI_Isend( ..., handle->comm, &(handle->isend_handle) );
25
        }
26
27
     User communication clean-up code:
28
29
        void user_end_op(user_lib_t *handle)
30
         {
^{31}
           MPI_Status status;
32
           MPI_Wait(&handle->isend_handle, &status);
33
           MPI_Wait(&handle->irecv_handle, &status);
34
        7
35
     User object clean-up code:
36
37
        void uninit_user_lib(user_lib_t *handle)
38
         {
39
           MPI_Comm_free(&(handle->comm));
40
           free(handle);
41
        }
42
43
     7.5.6 Library Example \#2
44
45
     The main program:
46
         int main(int argc, char *argv[])
47
         {
48
```

```
int ma, mb;
     MPI_Group group_world, group_a, group_b;
    MPI_Comm comm_a, comm_b;
     static int list_a[] = \{0, 1\};
#if defined(EXAMPLE_2B) || defined(EXAMPLE_2C)
     static int list_b[] = {0, 2, 3};
#else/* EXAMPLE_2A */
     static int list_b[] = \{0, 2\};
#endif
     int size_list_a = sizeof(list_a)/sizeof(int);
     int size_list_b = sizeof(list_b)/sizeof(int);
    MPI_Init(&argc, &argv);
     MPI_Comm_group(MPI_COMM_WORLD, &group_world);
     MPI_Group_incl(group_world, size_list_a, list_a, &group_a);
    MPI_Group_incl(group_world, size_list_b, list_b, &group_b);
     MPI_Comm_create(MPI_COMM_WORLD, group_a, &comm_a);
     MPI_Comm_create(MPI_COMM_WORLD, group_b, &comm_b);
     if(comm_a != MPI_COMM_NULL)
        MPI_Comm_rank(comm_a, &ma);
     if(comm_b != MPI_COMM_NULL)
        MPI_Comm_rank(comm_b, &mb);
     if(comm_a != MPI_COMM_NULL)
        lib_call(comm_a);
     if(comm_b != MPI_COMM_NULL)
       lib_call(comm_b);
       lib_call(comm_b);
     }
     if(comm_a != MPI_COMM_NULL)
       MPI_Comm_free(&comm_a);
     if(comm_b != MPI_COMM_NULL)
       MPI_Comm_free(&comm_b);
     MPI_Group_free(&group_a);
     MPI_Group_free(&group_b);
    MPI_Group_free(&group_world);
    MPI_Finalize();
     return 0;
  }
```

1

 $\mathbf{2}$

3

4

5

6

7

9 10

11

16

17

18 19

20 21

22 23 24

25

26

27 28 29

30 31

32 33 34

35

36 37

38 39

40 41

42

43 44

45 46

47

```
1
     The library:
2
         void lib_call(MPI_Comm comm)
3
         {
4
           int me, done = 0:
5
           MPI_Status status;
6
           MPI_Comm_rank(comm, &me);
7
           if(me == 0)
8
              while(!done)
9
              {
10
                  MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
11
12
              }
13
           else
14
           {
15
              /* work */
16
             MPI_Send(..., 0, ARBITRARY_TAG, comm);
17
18
              . . .
           }
19
     #ifdef EXAMPLE_2C
20
           /* include (resp, exclude) for safety (resp, no safety): */
21
           MPI_Barrier(comm);
22
     #endif
23
         }
^{24}
```

The above example is really three examples, depending on whether or not one includes rank in list_b, and whether or not a synchronize is included in lib_call. This example illustrates that, despite contexts, subsequent calls to lib_call with the same context need not be safe from one another (colloquially, "back-masking"). Safety is realized if the MPI_Barrier is added. What this demonstrates is that libraries have to be written carefully, even with contexts. When rank 3 is excluded, then the synchronize is not needed to get safety from back-masking.

Algorithms like "reduce" and "allreduce" have strong enough source selectivity properties so that they are inherently okay (no back-masking), provided that MPI provides basic guarantees. So are multiple calls to a typical tree-broadcast algorithm with the same root or different roots (see [64]). Here we rely on two guarantees of MPI: pairwise ordering of messages between processes in the same context, and source selectivity—deleting either feature removes the guarantee that back-masking cannot be required.

Algorithms that try to do non-deterministic broadcasts or other calls that include wildcard operations will not generally have the good properties of the deterministic implementations of "reduce," "allreduce," and "broadcast." Such algorithms would have to utilize the monotonically increasing tags (within a communicator scope) to keep things straight.

All of the foregoing is a supposition of "collective calls" implemented with point-topoint operations. MPI implementations may or may not implement collective calls using point-to-point operations. These algorithms are used to illustrate the issues of correctness and safety, independent of how MPI implements its collective calls. See also Section 7.9.

- 47
- 48

Inter-Communication 7.6

This section introduces the concept of inter-communication and describes the portions of MPI that support it. It describes support for writing programs that contain user-level servers.

All communication described thus far has involved communication between processes that are members of the same group. This type of communication is called "intra-communication" and the communicator used is called an "intra-communicator," as we have noted earlier in the chapter.

In modular and multi-disciplinary applications, different process groups execute distinct 10 modules and processes within different modules communicate with one another in a pipeline 11 or a more general module graph. In these applications, the most natural way for a process 12to specify a target process is by the rank of the target process within the target group. In 13 applications that contain internal user-level servers, each server may be a process group that 14 provides services to one or more clients, and each client may be a process group that uses the 15services of one or more servers. It is again most natural to specify the target process by rank 16within the target group in these applications. This type of communication is called "inter 17 -communication" and the communicator used is called an "inter-communicator," as 18 introduced earlier. 19

An inter-communication is a point-to-point communication between processes in different groups. The group containing a process that initiates an inter-communication 21operation is called the "local group," that is, the sender in a send and the receiver in a receive. The group containing the target process is called the "remote group," that is, the receiver in a send and the sender in a receive. As in intra-communication, the target process is specified using a (communicator, rank) pair. Unlike intra-communication, the rank is relative to a second, remote group.

All inter-communicator constructors are blocking except for MPI_COMM_IDUP and require that the local and remote groups be disjoint.

Advice to users. The groups must be disjoint for several reasons. Primarily, this is the intent of the inter-communicators—to provide a communicator for communication between disjoint groups. This is reflected in the definition of

MPI_INTERCOMM_MERGE, which allows the user to control the ranking of the processes in the created intra-communicator; this ranking makes little sense if the groups are not disjoint. In addition, the natural extension of collective operations to intercommunicators makes the most sense when the groups are disjoint. (End of advice to users.)

Here is a summary of the properties of inter-communication and inter-communicators:

- The syntax of point-to-point and collective communication is the same for both interand intra-communication. The same communicator can be used both for send and for receive operations.
- A target process is addressed by its rank in the remote group, both for sends and for receives.
- Communications using an inter-communicator are guaranteed not to conflict with any communications that use a different communicator.
- A communicator will provide either intra- or inter-communication, never both.

Unofficial Draft for Comment Only

1 2

3

4

5

6

 $\overline{7}$

8

9

20

22

23

 24

25

26

27

2829

30

 31

32

33

34

35

36

37

38 39

40

41

42

43

44 45

46

1	The routine MPI_COMM_TEST_INTER may be used to determine if a communicator is an
2	inter- or intra-communicator. Inter-communicators can be used as arguments to some of the
3	other communicator access routines. Inter-communicators cannot be used as input to some
4	of the constructor routines for intra-communicators (for instance, MPI_CART_CREATE).
5	
6	Advice to implementors. For the purpose of point-to-point communication, commu-
7	nicators can be represented in each process by a tuple consisting of:
8	
9	group
10	send_context
11	receive_context
12 13	source
14	For inter-communicators, group describes the remote group, and source is the rank of
15	the process in the local group. For intra-communicators, group is the communicator
16	group (remote=local), source is the rank of the process in this group, and send context
17	and <i>receive context</i> are identical. A group can be represented by a rank-to-absolute-
18	
19	address translation table.
20	The inter-communicator cannot be discussed sensibly without considering processes in
21	both the local and remote groups. Imagine a process \mathbf{P} in group \mathcal{P} , which has an inter-
22	communicator $\mathbf{C}_{\mathcal{P}}$, and a process \mathbf{Q} in group \mathcal{Q} , which has an inter-communicator
23	$\mathbf{C}_{\mathcal{Q}}$. Then
24	\mathbf{C} mound accritication mound \mathbf{C} and \mathbf{C} shows described the moun \mathcal{D}
25	• $\mathbf{C}_{\mathcal{P}}$.group describes the group \mathcal{Q} and $\mathbf{C}_{\mathcal{Q}}$.group describes the group \mathcal{P} .
26	• $C_{\mathcal{P}}$.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} .
27 28	• $\mathbf{C}_{\mathcal{P}}$.source is rank of P in \mathcal{P} and $\mathbf{C}_{\mathcal{Q}}$.source is rank of Q in \mathcal{Q} .
29	Assume that \mathbf{P} could a massage to \mathbf{Q} using the inter communicator. Then \mathbf{P} uses
30	Assume that \mathbf{P} sends a message to \mathbf{Q} using the inter-communicator. Then \mathbf{P} uses the group table to find the absolute address of \mathbf{Q} , gourge and good context are
31	the group table to find the absolute address of Q ; source and send_context are
32	appended to the message.
33	Assume that \mathbf{Q} posts a receive with an explicit source argument using the inter-
34	communicator. Then \mathbf{Q} matches receive_context to the message context and source
35	argument to the message source.
36	The same algorithm is appropriate for intra-communicators as well.
37	In order to support inter-communicator accessors and constructors, it is necessary to
38	supplement this model with additional structures, that store information about the
39	local communication group, and additional safe contexts. (<i>End of advice to imple-</i>
40	mentors.)
41	116/16013.)
42	
43	
44	
45	
46	
47	
48	

7.6.1 Inter-Communicator Accessors			
			2
			3
MPI_COM	MM_TEST_INTER(comm, flag	<u>z</u>)	4
	,	-,	5
IN	comm	communicator (handle)	6
OUT	flag	true if comm is an inter-communicator (logical)	7
			8
C bindi	ng		9
int MPI_	_Comm_test_inter(MPI_Comm	a comm, int *flag)	10
Fortron	2008 binding		11
	<pre></pre>	ierror)	12
	E(MPI_Comm), INTENT(IN) :		13
	CAL, INTENT(OUT) :: flag		14
	EGER, OPTIONAL, INTENT(OU		15 16
			10
Fortran	8		18
	1_TEST_INTER(COMM, FLAG,	IERROR)	19
	EGER COMM, IERROR		20
LUGI	ICAL FLAG		21
This local routine allows the calling process to determine if a communicator is an inter-			22
communi	cator or an intra-communica	tor. It returns true if it is an inter-communicator,	23
otherwise	e false.		24
Whe	n an inter-communicator is u	ised as an input argument to the communicator ac-	25
cessors de	escribed above under intra-co	mmunication, the following table describes behavior.	26
			27
		urns the size of the local group.	28
		Irns the local group.	29
	MPI_COMM_RANK retu	rns the rank in the local group	30
			31
Ta	ble 7.1: MPI_COMM_* Funct	ion Behavior (in Inter-Communication Mode)	32
		````	33

Furthermore, the operation MPI_COMM_COMPARE is valid for inter-communicators. Both communicators must be either intra- or inter-communicators, or else MPI_UNEQUAL results. Both corresponding local and remote groups must compare correctly to get the results MPI_CONGRUENT or MPI_SIMILAR. In particular, it is possible for MPI_SIMILAR to result because either the local or remote groups were similar but not identical.

The following accessors provide consistent access to the remote group of an intercommunicator. The following are all local operations.

Unofficial Draft for Comment Only

 $\frac{34}{35}$ 

36

37

38 39

40

```
1
 MPI_COMM_REMOTE_SIZE(comm, size)
2
 IN
 inter-communicator (handle)
 comm
3
 OUT
 size
 number of processes in the remote group of comm
4
 (integer)
5
6
\overline{7}
 C binding
8
 int MPI_Comm_remote_size(MPI_Comm comm, int *size)
9
 Fortran 2008 binding
10
 MPI_Comm_remote_size(comm, size, ierror)
11
 TYPE(MPI_Comm), INTENT(IN) :: comm
12
 INTEGER, INTENT(OUT) :: size
13
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
 Fortran binding
16
 MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR)
17
 INTEGER COMM, SIZE, IERROR
18
19
20
 MPI_COMM_REMOTE_GROUP(comm, group)
21
 IN
 inter-communicator (handle)
 comm
22
 OUT
 remote group corresponding to comm (handle)
23
 group
^{24}
25
 C binding
26
 int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
27
 Fortran 2008 binding
28
 MPI_Comm_remote_group(comm, group, ierror)
29
 TYPE(MPI_Comm), INTENT(IN) :: comm
30
 TYPE(MPI_Group), INTENT(OUT) :: group
^{31}
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33
 Fortran binding
34
 MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR)
35
 INTEGER COMM, GROUP, IERROR
36
37
 Rationale.
 Symmetric access to both the local and remote groups of an inter-
38
 communicator is important, so this function, as well as MPI_COMM_REMOTE_SIZE
39
 have been provided. (End of rationale.)
40
41
 7.6.2 Inter-Communicator Operations
42
43
 This section introduces five blocking inter-communicator operations.
44
 MPI_INTERCOMM_CREATE is used to bind two intra-communicators into an inter-com-
45
 municator; the function MPI_INTERCOMM_CREATE_FROM_GROUPS constructs an inter-
46
 communicator from two previously defined disjoint groups; the function
47
 MPI_INTERCOMM_MERGE creates an intra-communicator by merging the local and remote
48
```

groups of an inter-communicator. The functions MPI_COMM_DUP and MPI_COMM_FREE, introduced previously, duplicate and free an inter-communicator, respectively.

Overlap of local and remote groups that are bound into an inter-communicator is prohibited. If there is overlap, then the program is erroneous and is likely to deadlock.

The function MPI_INTERCOMM_CREATE can be used to create an inter-communicator from two existing intra-communicators, in the following situation: At least one selected member from each group (the "group leader") has the ability to communicate with the selected member from the other group; that is, a "peer" communicator exists to which both leaders belong, and each leader knows the rank of the other leader in this peer communicator. Furthermore, members of each group know the rank of their leader.

Construction of an inter-communicator from two intra-communicators requires separate collective operations in the local group and in the remote group, as well as a point-to-point communication between a process in the local group and a process in the remote group.

When using the World Model, the MPI_COMM_WORLD communicator (or preferably a dedicated duplicate thereof) can be this peer communicator. For applications that use the Sessions Model, or the spawn or join operations, it may be necessary to first create an intra-communicator to be used as the peer communicator.

The application topology functions described in Chapter 8 do not apply to intercommunicators. Users that require this capability should utilize MPI_INTERCOMM_MERGE to build an intra-communicator, then apply the graph or cartesian topology capabilities to that intra-communicator, creating an appropriate topologyoriented intra-communicator. Alternatively, it may be reasonable to devise one's own application topology mechanisms for this case, without loss of generality.

#### MPI_INTERCOMM_CREATE(local_comm, local_leader, peer_comm, remote_leader, tag, newintercomm)

IN	local_comm	local intra-communicator (handle)	28			
IN	local_leader	rank of local group leader in local_comm (integer)	29			
IN	peer_comm	"peer" communicator; significant only at the	30 31			
		local_leader (handle)	32			
IN	remote_leader	rank of remote group leader in peer_comm;	33			
	_	significant only at the local_leader (integer)	34			
IN	tag	tag (integer)	35			
			36			
OUT	newintercomm	new inter-communicator (handle)	37			
			38			
C binding						
nt MPI Intercomm create(MPI Comm local comm, int local leader,						

### C

```
int MPI_Inter
 comm_create(MPI_Comm local_comm, int local_leader,
 MPI_Comm peer_comm, int remote_leader, int tag,
 MPI_Comm *newintercomm)
```

### Fortran 2008 binding

```
MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader,
 45
 tag, newintercomm, ierror)
 46
 TYPE(MPI_Comm), INTENT(IN) :: local_comm, peer_comm
 47
 INTEGER, INTENT(IN) :: local_leader, remote_leader, tag
 48
```

1  $\mathbf{2}$ 

3

4

5

6  $\overline{7}$ 

8 9

10 11

12

13

1415

16

17

18

19 20

21

22

23 24 25

26

27

41

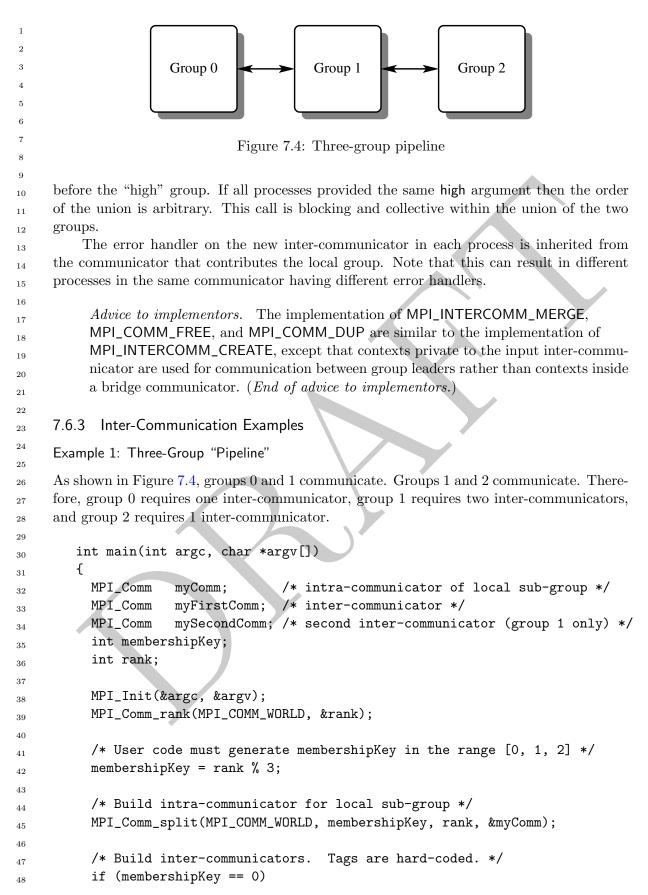
4243

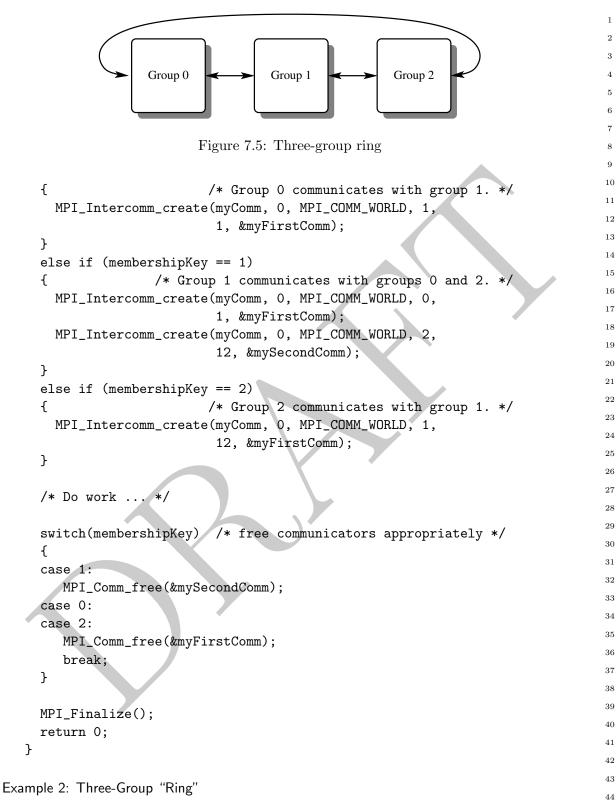
12		MPI_Comm), INTENT(OUT) : ER, OPTIONAL, INTENT(OUT				
3						
4	Fortran b	6				
5	MP1_INTER		LOCAL_LEADER, PEER_COMM, REMOTE_LEADER,			
6	тытго	TAG, NEWINTERCOMM, I				
7	INTEG		DER, PEER_COMM, REMOTE_LEADER, TAG,			
8	NEWINTERCOMM, IERROR					
9	This call creates an inter-communicator. It is collective over the union of the local and					
10	remote groups. MPI processes should provide identical local_comm and					
11 12			p. Wildcards are not permitted for remote_leader,			
12	local_leade	r, and tag.				
14						
15	MPI_INTE	RCOMM_CREATE_FROM_G	ROUPS(local_group, local_leader, remote_group,			
16		remote_leader, stringtag,	info, errhandler, newintercomm)			
17 18	IN	local_group	local group (handle)			
19	IN	local_leader	rank of local group leader in local_group (integer)			
20	IN	remote_group	remote group, significant only at $local_leader$ (handle)			
21	IN	remote_leader	rank of remote group leader in remote_group,			
22			significant only at local_leader (integer)			
23 24	IN	stringtag	unique idenitifier for this operation (string)			
25	IN	info	info object (handle)			
26	IN	errhandler	error handler to be attached to new			
27			inter-communicator (handle)			
28	OUT	newintercomm	new inter-communicator (handle)			
29 30						
31	C binding	g				
32	int MPI_I		ups(MPI_Group local_group,			
33			I_Group remote_group, int remote_leader,			
34		const char *stringta	-			
35		MPI_Errnandler errna	ndler, MPI_Comm *newintercomm)			
36	Fortran 2008 binding					
37 38	MPI_Intercomm_create_from_groups(local_group, local_leader, remote_group,					
39	remote_leader, stringtag, info, errhandler, newintercomm,					
40	ierror)					
41	TYPE(MPI_Group), INTENT(IN) :: local_group, remote_group INTEGER, INTENT(IN) :: local_leader, remote_leader					
42	CHARACTER(LEN=*), INTENT(IN) :: stringtag					
43	TYPE(MPI_Info), INTENT(IN) :: info					
44	TYPE(MPI_Errhandler), INTENT(IN) :: errhandler					
45	TYPE(MPI_Comm), INTENT(OUT) :: newintercomm					
46	INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
47						
48						

Fortran binding					
MPI_INTERCOMM_CREATE_FROM_GROUPS(LOCAL_GROUP, LOCAL_LEADER, REMOTE_GROUP,					
REMOTE_LEADER, STRINGTAG, INFO, ERRHANDLER, NEWINTERCOMM,					
IERROR)					
INTEGER LOCAL_GROUP, LOCAL_LEADER, REMOTE_GROUP, REMOTE_LEADER, INFO,					
ERRHANDLER, NEWINTERCOMM, IERROR					
CHARACTER*(*) STRINGTAG					
This call are	This call exectes on inter communicator. Unlike MDL INTERCOMM CREATE this function				
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					
			12 13		
stringtag argument. Within each group, all MPI processes shall provide identical local_group, local_leader arguments. Wildcards are not permitted for the					
• • • •	8		14 15		
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					
0	concurrent calls in a multithreaded environment. The stringtag shall not exceed				
		aracters in length. For C, this includes space for	17 18		
a null terminating character. In the event that MPI_GROUP_EMPTY is supplied as the					
		the call is a local operation and MPI_COMM_NULL	19		
	as the newintercomm.		20		
15 100411104			21		
			22		
MPI_INTER	COMM_MERGE(intercomm,	high, newintracomm)	23		
IN	intercomm	inter-communicator (handle)	24 25		
IN	high	ordering of the local and remote groups in the new	26		
		intra-communicator (logical)	27		
OUT	newintracomm	new intra-communicator (handle)	28		
	· ·		29		
C binding			30		
0	tercomm_merge(MPI_Comm i	ntercomm, int high,	31 32		
	MPI_Comm *newintracomm)				
			33 34		
Fortran 2008 binding					
	MPI_Intercomm_merge(intercomm, high, newintracomm, ierror)				
TYPE(MPI_Comm), INTENT(IN) :: intercomm					
	L, INTENT(IN) :: high		37 38		
TYPE(MPI_Comm), INTENT(OUT) :: newintracomm					
INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
Fortran binding					
MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR)					
INTEGER INTERCOMM, NEWINTRACOMM, IERROR					
LOGICA	L HIGH		43 44		
This function creates an intra-communicator from the union of the two groups that are					
The remember of the communication from the anon of the two groups that are					

This function creates an intra-communicator from the union of the two groups that are associated with intercomm. All processes should provide the same high value within each of the two groups. If processes in one group provided the value high = false and processes in the other group provided the value high = true then the union orders the "low" group 48

### Unofficial Draft for Comment Only





As shown in Figure 7.5, groups 0 and 1 communicate. Groups 1 and 2 communicate. Groups 0 and 2 communicate. Therefore, each requires two inter-communicators.

```
int main(int argc, char *argv[])
```

45

46 47

```
1
 {
\mathbf{2}
 MPI_Comm
 myComm;
 /* intra-communicator of local sub-group */
3
 MPI_Comm
 myFirstComm; /* inter-communicators */
4
 MPI_Comm
 mySecondComm;
5
 int membershipKey;
6
 int rank;
7
8
 MPI_Init(&argc, &argv);
9
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10
 . . .
11
12
 /* User code must generate membershipKey in the range [0, 1, 2] */
13
 membershipKey = rank % 3;
14
15
 /* Build intra-communicator for local sub-group */
16
 MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);
17
18
 /* Build inter-communicators. Tags are hard-coded. */
19
 if (membershipKey == 0)
20
 ſ
 /* Group 0 communicates with groups 1 and 2. */
21
 MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1,
22
 1, &myFirstComm);
23
 MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 2,
24
 2, &mySecondComm);
25
 }
26
 else if (membershipKey == 1)
 /* Group 1 communicates with groups 0 and 2. */
27
 {
28
 MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 0,
29
 1, &myFirstComm);
30
 MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 2,
31
 12, &mySecondComm);
32
 }
33
 else if (membershipKey == 2)
34
 /* Group 2 communicates with groups 0 and 1. */
 {
35
 MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 0,
36
 2, &myFirstComm);
37
 MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1,
38
 12, &mySecondComm);
39
 }
40
41
 /* Do some work ... */
42
43
 /* Then free communicators before terminating... */
44
 MPI_Comm_free(&myFirstComm);
45
 MPI_Comm_free(&mySecondComm);
46
 MPI_Comm_free(&myComm);
47
 MPI_Finalize();
48
 return 0;
```

}

# 7.7 Caching

MPI provides a "caching" facility that allows an application to attach arbitrary pieces of information, called **attributes**, to three kinds of MPI objects: communicators, windows, and datatypes. More precisely, the caching facility allows a portable library to do the following:

- pass information between calls by associating it with an MPI intra- or inter-communicator, window, or datatype,
- quickly retrieve that information, and
- be guaranteed that out-of-date information is never retrieved, even if the object is freed and its handle subsequently reused by MPI.

The caching capabilities, in some form, are required by built-in MPI routines such as collective communication and application topology. Defining an interface to these capabilities as part of the MPI standard is valuable because it permits routines like collective communication and application topologies to be implemented as portable code, and also because it makes MPI more extensible by allowing user-written routines to use standard MPI calling sequences.

Advice to users. The communicator MPI_COMM_SELF is a suitable choice for posting process-local attributes, via this attribute-caching mechanism. (*End of advice to* users.)

*Rationale.* In one extreme one can allow caching on all opaque handles. The other extreme is to only allow it on communicators. Caching has a cost associated with it and should only be allowed when it is clearly needed and the increased cost is modest. This is the reason that windows and datatypes were added but not other handles. (*End of rationale.*)

One difficulty is the potential for size differences between Fortran integers and C pointers. For this reason, the Fortran versions of these routines use integers of kind MPI_ADDRESS_KIND.

Advice to implementors. High-quality implementations should raise an error when a keyval that was created by a call to MPI_XXX_CREATE_KEYVAL is used with an object of the wrong type with a call to MPI_YYY_GET_ATTR, MPI_YYY_SET_ATTR, MPI_YYY_DELETE_ATTR, or MPI_YYY_FREE_KEYVAL. To do so, it is necessary to maintain, with each keyval, information on the type of the associated user function. (*End of advice to implementors.*)

## 7.7.1 Functionality

Attributes can be attached to communicators, windows, and datatypes. Attributes are local ⁴⁴ to the process and specific to the communicator to which they are attached. Attributes are ⁴⁵ not propagated by MPI from one communicator to another except when the communicator ⁴⁶ is duplicated using MPI_COMM_DUP or MPI_COMM_IDUP (and even then the application ⁴⁷ must give specific permission through callback functions for the attribute to be copied). ⁴⁸

## Unofficial Draft for Comment Only

 24 

 31 

# 362 CHAPTER 7. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

be a obje	a pointer to a structure that ect. In Fortran, attributes a	C are of type void*. Typically, such an attribute will t contains further information, or a handle to an MPI re of type INTEGER. Such attribute can be a handle to er-valued attribute. ( <i>End of advice to users.</i> )	
a C-	_	butes are scalar values, equal in size to, or larger than es can always hold an MPI handle. ( <i>End of advice to</i>	
		re requires that attributes be stored by MPI opaquely latatype. Accessor functions include the following:	
	s by which $MPI$ informs the	ntify an attribute); the user specifies "callback" func- e application when the communicator is destroyed or	
• stor	e and retrieve the value of a	an attribute;	
in re peat	esponse to explicit application	ng and callback functions are only called synchronously, on requests. This avoids problems that result from re- and system space. (This synchronous calling rule is a	
imp	general property of MPL) The choice of key values is under control of MPL. This allows MPL to optimize its implementation of attribute sets. It also avoids conflict between independent modules caching information on the same communicators.		
cach bacl com acce tabl effic	ing facility to be implement in the interface, some form of tab municators. In contrast, th ess to attributes through the e) and cleverly chosen key v iency "hit" inherent in the r	sting of just a callback facility, would allow the entire ted by portable code. However, with the minimal call- ble searching is implied by the need to handle arbitrary e more complete interface defined here permits rapid use of pointers in communicators (to find the attribute alues (to retrieve individual attributes). In light of the minimal interface, the more complete interface defined and of advice to implementors.)	
MPI provi	des the following services re	elated to caching. They are all process local.	
	ommunicators		
Functions	for caching on communicat	fors are:	
MPI_COM	IM_CREATE_KEYVAL(com extra_state)	m_copy_attr_fn, comm_delete_attr_fn, comm_keyval,	
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)	
	extra_state	extra state for callback function	

<pre>MPI_Comm_delete_attr_function *comm_delete_attr_fn, int *comm_keyval, void *extra_state)</pre>	1 2
·	3
Fortran 2008 binding	4
<pre>MPI_Comm_create_keyval(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval,</pre>	5
PROCEDURE(MPI_Comm_copy_attr_function), INTENT(IN) :: comm_copy_attr_fn	6
PROCEDURE(MPI_Comm_delete_attr_function), INTENT(IN) ::	7
comm_delete_attr_fn	8 9
INTEGER, INTENT(OUT) :: comm_keyval	9 10
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state	11
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	12
Fortran binding	13
MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL,	14
EXTRA_STATE, IERROR)	15
EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN	16
INTEGER COMM_KEYVAL, IERROR	17
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	18
Generates a new attribute key. Keys are locally unique in a process, and opaque to	19
user, though they are explicitly stored in integers. Once allocated, the key value can be	20 21
used to associate attributes and access them on any locally defined communicator.	21
The C callback functions are:	23
<pre>typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm, int comm_keyval,</pre>	24
<pre>void *extra_state, void *attribute_val_in,</pre>	25
<pre>void *attribute_val_out, int *flag);</pre>	26
and	27
typedef int MPI_Comm_delete_attr_function(MPI_Comm comm, int comm_keyval,	28
<pre>void *attribute_val, void *extra_state);</pre>	29
which are the same as the MPI-1.1 calls but with a new name. The old names are deprecated.	30
With the mpi_f08 module, the Fortran callback functions are:	31
ABSTRACT INTERFACE	32 33
SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,	34
attribute_val_in, attribute_val_out, flag, ierror)	35
TYPE(MPI_Comm) :: oldcomm	36
INTEGER :: comm_keyval, ierror	37
<pre>INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,</pre>	38
attribute_val_out	39
LOGICAL :: flag	40
and	41
ABSTRACT INTERFACE	42
SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval,	43
attribute_val, extra_state, ierror)	44 45
TYPE(MPI_Comm) :: comm	46
INTEGER :: comm_keyval, ierror	47
INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state	48

1	With the mpi module and mpif.h, the Fortran callback functions are:
2	SUBROUTINE COMM_COPY_ATTR_FUNCTION(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
3	ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
4	INTEGER OLDCOMM, COMM_KEYVAL, IERROR
5	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
6	ATTRIBUTE_VAL_OUT
7	LOGICAL FLAG
8	and
9	and
10	SUBROUTINE COMM_DELETE_ATTR_FUNCTION(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
11	EXTRA_STATE, IERROR)
12	INTEGER COMM, COMM_KEYVAL, IERROR
13	INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
14	The comm_copy_attr_fn function is invoked when a communicator is duplicated by
15	MPI_COMM_DUP or MPI_COMM_IDUP. comm_copy_attr_fn should be of type
16	MPI_Comm_copy_attr_function. The copy callback function is invoked for each key value in
17	oldcomm in arbitrary order. Each call to the copy callback is made with a key value and its
18	corresponding attribute. If it returns $flag = 0$ or .FALSE., then the attribute is deleted in
19	the duplicated communicator. Otherwise ( $flag = 1$ or .TRUE.), the new attribute value is
20	set to the value returned in attribute_val_out. The function returns MPI_SUCCESS on success
21	and an error code on failure (in which case MPI_COMM_DUP or MPI_COMM_IDUP will
22	fail).
23	The argument comm_copy_attr_fn may be specified as MPI_COMM_NULL_COPY_FN
24	or MPI_COMM_DUP_FN from either C or Fortran. MPI_COMM_NULL_COPY_FN is a
25	function that does nothing other than returning $flag = 0$ or .FALSE. (depending on whether
26	the keyval was created with a C or Fortran binding to MPI_COMM_CREATE_KEYVAL) and
27	MPI_SUCCESS. MPI_COMM_DUP_FN is a simple copy function that sets flag = 1 or .TRUE.
28	returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS. These
29	replace the MPI-1 predefined callbacks MPI_NULL_COPY_FN and MPI_DUP_FN, whose
30	use is deprecated.
31	
32	Advice to users. Even though both formal arguments attribute_val_in and
33	attribute_val_out are of type void*, their usage differs. The C copy function is passed
34	by MPI in attribute_val_in the value of the attribute, and in attribute_val_out the
35	address of the attribute, so as to allow the function to return the (new) attribute
36	value. The use of type void* for both is to avoid messy type casts.
37	A valid copy function is one that completely duplicates the information by making
38	a full duplicate copy of the data structures implied by an attribute; another might
39	just make another reference to that data structure, while using a reference-count
40	mechanism. Other types of attributes might not copy at all (they might be specific
41	to oldcomm only). (End of advice to users.)
42	
43	Advice to implementors. A C interface should be assumed for copy and delete
44	functions associated with key values created in C; a Fortran calling interface should
45	be assumed for key values created in Fortran. (End of advice to implementors.)
46	
47	Analogous to comm_copy_attr_fn is a callback deletion function, defined as follows.
48	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.	1
comm_delete_attr_fn should be of type MPI_Comm_delete_attr_function.	2
This function is called by MPI_COMM_FREE, MPI_COMM_DELETE_ATTR, and	3
MPI_COMM_SET_ATTR to do whatever is needed to remove an attribute. The function	4
returns MPI_SUCCESS on success and an error code on failure (in which case	5
	6
MPI_COMM_FREE will fail).	
The argument comm_delete_attr_fn may be specified as	7
MPI_COMM_NULL_DELETE_FN from either C or Fortran.	8
MPI_COMM_NULL_DELETE_FN is a function that does nothing, other than returning	9
MPI_SUCCESS. MPI_COMM_NULL_DELETE_FN replaces MPI_NULL_DELETE_FN, whose	10
use is deprecated.	11
If an attribute copy function or attribute delete function returns other than	12
MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_COMM_FREE),	13
is erroneous.	14
The special key value MPI_KEYVAL_INVALID is never returned by	15
MPI_COMM_CREATE_KEYVAL. Therefore, it can be used for static initialization of key	16
	17
values.	18
Advice to implementors. The predefined Fortran functions	
MPI_COMM_NULL_COPY_FN, MPI_COMM_DUP_FN, and	19
MPI_COMM_NULL_DELETE_FN are defined in the mpi module (and mpif.h) and	20
the mpi_f08 module with the same name, but with different interfaces. Each function	21
-	22
can coexist twice with the same name in the same MPI library, one routine as an	23
implicit interface outside of the mpi module, i.e., declared as EXTERNAL, and the other	24
routine within mpi_f08 declared with CONTAINS. These routines have different link	25
names, which are also different to the link names used for the routines used in C.	26
(End of advice to implementors.)	27
Advice to users. Callbacks, including the predefined Fortran functions	28
MPI_COMM_NULL_COPY_FN, MPI_COMM_DUP_FN, and	29
MPI_COMM_NULL_DELETE_FN should not be passed from one application routine	30
	31
that uses the mpi_f08 module to another application routine that uses the mpi module	32
or mpif.h, and vice versa; see also the advice to users on page 840. (End of advice to	33
users.)	34
	35
MPI_COMM_FREE_KEYVAL(comm_keyval)	36
	37
INOUT comm_keyval key value (integer)	38
	39
C binding	40
int MPI_Comm_free_keyval(int *comm_keyval)	41
Fortron 2008 hinding	42
Fortran 2008 binding	43
MPI_Comm_free_keyval(comm_keyval, ierror)	44
INTEGER, INTENT(INOUT) :: comm_keyval	45
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	46
Fortran binding	47
MPI_COMM_FREE_KEYVAL(COMM_KEYVAL, IERROR)	48

366 CHAPTER 7. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

```
1
 INTEGER COMM_KEYVAL, IERROR
\mathbf{2}
 Frees an extant attribute key. This function sets the value of keyval to
3
 MPI_KEYVAL_INVALID. Note that it is not erroneous to free an attribute key that is in use,
4
 because the actual free does not transpire until after all references (in other communicators
5
 on the process) to the key have been freed. These references need to be explicitly freed by the
6
 program, either via calls to MPI_COMM_DELETE_ATTR that free one attribute instance,
7
 or by calls to MPI_COMM_FREE that free all attribute instances associated with the freed
8
 communicator.
9
10
11
 MPI_COMM_SET_ATTR(comm, comm_keyval, attribute_val)
12
 INOUT
 communicator to which attribute will be attached
 comm
13
 (handle)
14
 key value (integer)
15
 IN
 comm_keyval
16
 attribute_val
 IN
 attribute value
17
18
 C binding
19
 int MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void *attribute_val)
20
21
 Fortran 2008 binding
22
 MPI_Comm_set_attr(comm, comm_keyval, attribute_val, ierror)
23
 TYPE(MPI_Comm), INTENT(IN) :: comm
^{24}
 INTEGER, INTENT(IN) :: comm_keyval
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
25
26
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
 Fortran binding
28
 MPI_COMM_SET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, IERROR)
29
 INTEGER COMM, COMM_KEYVAL, IERROR
30
 INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
^{31}
32
 This function stores the stipulated attribute value attribute_val for subsequent retrieval
33
 by MPI_COMM_GET_ATTR. If the value is already present, then the outcome is as if
34
 MPI_COMM_DELETE_ATTR was first called to delete the previous value (and the callback
35
 function comm_delete_attr_fn was executed), and a new value was next stored. The call
36
 is erroneous if there is no key with value keyval; in particular MPI_KEYVAL_INVALID is an
37
 erroneous key value. The call will fail if the comm_delete_attr_fn function returned an error
38
 code other than MPI_SUCCESS.
39
40
41
42
43
44
45
46
47
48
```

MPI_COMM_GET_ATTR(comm, comm_keyval, attribute_val, flag) ¹			
IN	comm	communicator to which the attribute is attached	2
		(handle)	3
IN	comm_keyval	key value (integer)	4
	5		5
OUT	attribute_val	attribute value, unless $flag = false$	6
OUT	flag	false if no attribute is associated with the key	7
		(logical)	8 9
			9 10
C bindi	ng		10
int MPI	_Comm_get_attr(MPI_Co	omm comm, int comm_keyval, void *attribute_val,	12
	int *flag)		13
Fortron	2008 hinding		14
Fortran 2008 binding			15
<pre>MPI_Comm_get_attr(comm, comm_keyval, attribute_val, flag, ierror)     TYPE(MPI_Comm), INTENT(IN) :: comm</pre>			16
INTEGER, INTENT(IN) :: comm_keyval			17
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val			18
LOGICAL, INTENT(OUT) :: flag			19
INTEGER, OPTIONAL, INTENT(OUT) :: ierror			20
			21
	Fortran binding		
		4_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)	23
	EGER COMM, COMM_KEYV		24
		S_KIND) ATTRIBUTE_VAL	25
LUG	ICAL FLAG		26
Ret	rieves attribute value by	key. The call is erroneous if there is no key with value	27
keyval. (	On the other hand, the	call is correct if the key value exists, but no attribute is	28
attached	on comm for that key;	in such case, the call returns $flag = false$ . In particular	29
			30

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_get_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.)

MPI_KEYVAL_INVALID is an erroneous key value.

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.)

1 MPI_COMM_DELETE_ATTR(comm, comm_keyval) 2 INOUT communicator from which the attribute is deleted comm 3 (handle) 4 IN comm_keyval key value (integer) 56 C binding 7int MPI_Comm_delete_attr(MPI_Comm comm, int comm_keyval) 8 9 Fortran 2008 binding 10 MPI_Comm_delete_attr(comm, comm_keyval, ierror) 11 TYPE(MPI_Comm), INTENT(IN) :: comm 12INTEGER, INTENT(IN) :: comm_keyval 13 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1415Fortran binding 16MPI_COMM_DELETE_ATTR(COMM, COMM_KEYVAL, IERROR) 17INTEGER COMM, COMM_KEYVAL, IERROR 18 Delete attribute from cache by key. This function invokes the attribute delete function 19comm_delete_attr_fn specified when the keyval was created. The call will fail if the 20comm_delete_attr_fn function returns an error code other than MPI_SUCCESS. 21Whenever a communicator is replicated using the function MPI_COMM_DUP or 22 MPI_COMM_IDUP, all call-back copy functions for attributes that are currently set are 23invoked (in arbitrary order). Whenever a communicator is deleted using the function 24MPI_COMM_FREE all callback delete functions for attributes that are currently set are 25invoked. 26277.7.3 Windows 28 29The functions for caching on windows are: 30  31 MPI_WIN_CREATE_KEYVAL(win_copy_attr_fn, win_delete_attr_fn, win_keyval, 32 extra_state) 33 34 IN win_copy_attr_fn copy callback function for win_keyval (function) 35IN win_delete_attr_fn delete callback function for win_keyval (function) 36 win_keyval OUT 37 key value for future access (integer) 38 IN extra_state extra state for callback function 39 40 C binding 41 int MPI_Win_create_keyval(MPI_Win_copy_attr_function *win_copy_attr_fn, 42MPI_Win_delete_attr_function *win_delete_attr_fn, 43 int *win_keyval, void *extra_state) 4445Fortran 2008 binding MPI_Win_create_keyval(win_copy_attr_fn, win_delete_attr_fn, win_keyval, 4647extra_state, ierror) 48 PROCEDURE(MPI_Win_copy_attr_function), INTENT(IN) :: win_copy_attr_fn

```
PROCEDURE(MPI_Win_delete_attr_function), INTENT(IN) ::
 1
 2
 win_delete_attr_fn
 INTEGER, INTENT(OUT) :: win_keyval
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 5
Fortran binding
MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,
 EXTRA_STATE, IERROR)
 EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN
 10
 INTEGER WIN_KEYVAL, IERROR
 11
 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
 12
 13
 The argument win_copy_attr_fn may be specified as MPI_WIN_NULL_COPY_FN or
 14
MPI_WIN_DUP_FN from either C or Fortran. MPI_WIN_NULL_COPY_FN is a function
 15
that does nothing other than returning flag = 0 and MPL_SUCCESS. MPL_WIN_DUP_FN is
 16
a simple copy function that sets flag = 1, returns the value of attribute_val_in in
 17
attribute_val_out, and returns MPI_SUCCESS.
 18
 The argument win_delete_attr_fn may be specified as MPI_WIN_NULL_DELETE_FN
 19
from either C or Fortran. MPI_WIN_NULL_DELETE_FN is a function that does nothing,
other than returning MPI_SUCCESS.
 20
 21
The C callback functions are:
 22
typedef int MPI_Win_copy_attr_function(MPI_Win oldwin, int win_keyval,
 23
 void *extra_state, void *attribute_val_in,
 ^{24}
 void *attribute_val_out, int *flag);
 25
and
 26
typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval,
 27
 void *attribute_val, void *extra_state);
 28
 29
With the mpi_f08 module, the Fortran callback functions are:
 30
ABSTRACT INTERFACE
 31
 SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state,
 32
 attribute_val_in, attribute_val_out, flag, ierror)
 33
 TYPE(MPI_Win) :: oldwin
 34
 INTEGER :: win_keyval, ierror
 35
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
 36
 attribute_val_out
 37
 LOGICAL :: flag
 38
and
 39
ABSTRACT INTERFACE
 40
 SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,
 41
 extra_state, ierror)
 42
 TYPE(MPI_Win) :: win
 43
 INTEGER :: win_keyval, ierror
 44
 INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
 45
 46
With the mpi module and mpif.h, the Fortran callback functions are:
 47
```

```
1
 SUBROUTINE WIN_COPY_ATTR_FUNCTION(OLDWIN, WIN_KEYVAL, EXTRA_STATE,
\mathbf{2}
 ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
3
 INTEGER OLDWIN, WIN_KEYVAL, IERROR
4
 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
5
 ATTRIBUTE_VAL_OUT
6
 LOGICAL FLAG
7
 and
8
 SUBROUTINE WIN_DELETE_ATTR_FUNCTION(WIN, WIN_KEYVAL, ATTRIBUTE_VAL,
9
 EXTRA_STATE, IERROR)
10
 INTEGER WIN, WIN_KEYVAL, IERROR
11
 INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
12
13
 If an attribute copy function or attribute delete function returns other than
14
 MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_WIN_FREE), is
15
 erroneous.
16
17
 MPI_WIN_FREE_KEYVAL(win_keyval)
18
19
 INOUT
 win_keyval
 key value (integer)
20
21
 C binding
22
 int MPI_Win_free_keyval(int *win_keyval)
23
 Fortran 2008 binding
^{24}
 MPI_Win_free_keyval(win_keyval, ierror)
25
 INTEGER, INTENT(INOUT) :: win_keyval
26
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
 Fortran binding
29
 MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)
30
 INTEGER WIN_KEYVAL, IERROR
^{31}
32
33
 MPI_WIN_SET_ATTR(win, win_keyval, attribute_val)
34
 INOUT
35
 win
 window to which attribute will be attached (handle)
36
 IN
 win_keyval
 key value (integer)
37
 IN
 attribute_val
 attribute value
38
39
 C binding
40
 int MPI_Win_set_attr(MPI_Win win, int win_keyval, void *attribute_val)
41
42
 Fortran 2008 binding
43
 MPI_Win_set_attr(win, win_keyval, attribute_val, ierror)
44
 TYPE(MPI_Win), INTENT(IN) :: win
45
 INTEGER, INTENT(IN) :: win_keyval
46
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
47
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

Fortran binding MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR) INTEGER WIN, WIN_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL			1 2 3 4 5 6	
MPI WIN	GET_ATTR(win, win_keyval, a	attribute val. flag)	7	
	IN win window to which the attribute is attached (handle)			
IN	win_keyval	key value (integer)	9 10	
			11	
	OUTattribute_valattribute value, unless flag = false			
OUT	flag	false if no attribute is associated with the key $(1 + 1)$	13	
		(logical)	14	
C binding	r		15 16	
-		int win_keyval, void *attribute_val,	17	
	int *flag)		18	
Fortran 2	008 binding		19	
	6	attribute_val, flag, ierror)	20	
TYPE(	MPI_Win), INTENT(IN) :: (	win	21 22	
	ER, INTENT(IN) :: win_ke		22	
		), INTENT(OUT) :: attribute_val	24	
LOGICAL, INTENT(OUT) :: flag			25	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror			26	
	Fortran binding MPI_WIN_GET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)			
	EI_AIIR(WIN, WIN_KEYVAL, ER WIN, WIN_KEYVAL, IERR(		28 29	
	ER(KIND=MPI_ADDRESS_KIND)		30	
	AL FLAG		31	
			32	
			33	
MPI_WIN_	DELETE_ATTR(win, win_key	val)	34	
INOUT	win	window from which the attribute is deleted (handle)	35 36	
IN	win_keyval	key value (integer)	37	
	·		38	
C binding	g		39	
int MPI_W	in_delete_attr(MPI_Win w:	in, int win_keyval)	40	
Fortran 2	Fortran 2008 binding			
MPI_Win_d	elete_attr(win, win_keyva	al, ierror)	42 43	
	MPI_Win), INTENT(IN) :: v		44	
	ER, INTENT(IN) :: win_key	•	45	
INIEG	ER, OPTIONAL, INTENT(OUT)	) :: lerror	46	
Fortran b	0		47	
MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR) 48			48	

```
1
 INTEGER WIN, WIN_KEYVAL, IERROR
\mathbf{2}
3
 7.7.4 Datatypes
4
\mathbf{5}
 The new functions for caching on datatypes are:
6
7
 MPI_TYPE_CREATE_KEYVAL(type_copy_attr_fn, type_delete_attr_fn, type_keyval,
8
 extra_state)
9
10
 IN
 type_copy_attr_fn
 copy callback function for type_keyval (function)
11
 IN
 type_delete_attr_fn
 delete callback function for type_keyval (function)
12
 OUT
 type_keyval
 key value for future access (integer)
13
14
 IN
 extra_state
 extra state for callback function
15
16
 C binding
17
 int MPI_Type_create_keyval(MPI_Type_copy_attr_function *type_copy_attr_fn,
18
 MPI_Type_delete_attr_function *type_delete_attr_fn,
19
 int *type_keyval, void *extra_state)
20
 Fortran 2008 binding
21
 MPI_Type_create_keyval(type_copy_attr_fn, type_delete_attr_fn, type_keyval,
22
23
 extra_state, ierror)
24
 PROCEDURE(MPI_Type_copy_attr_function), INTENT(IN) :: type_copy_attr_fn
25
 PROCEDURE(MPI_Type_delete_attr_function), INTENT(IN) ::
26
 type_delete_attr_fn
27
 INTEGER, INTENT(OUT) :: type_keyval
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
28
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
 Fortran binding
^{31}
 MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL,
32
 EXTRA_STATE, IERROR)
33
 EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN
34
 INTEGER TYPE_KEYVAL, IERROR
35
 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
36
37
 The argument type_copy_attr_fn may be specified as MPI_TYPE_NULL_COPY_FN or
38
 MPI_TYPE_DUP_FN from either C or Fortran. MPI_TYPE_NULL_COPY_FN is a function
39
 that does nothing other than returning flag = 0 and MPI_SUCCESS. MPI_TYPE_DUP_FN
40
 is a simple copy function that sets flag = 1, returns the value of attribute_val_in in
41
 attribute_val_out, and returns MPI_SUCCESS.
42
 The argument type_delete_attr_fn may be specified as MPI_TYPE_NULL_DELETE_FN
43
 from either C or Fortran. MPI_TYPE_NULL_DELETE_FN is a function that does nothing,
44
 other than returning MPI_SUCCESS.
45
 The C callback functions are:
46
 typedef int MPI_Type_copy_attr_function(MPI_Datatype oldtype,
47
 int type_keyval, void *extra_state, void *attribute_val_in,
48
 void *attribute_val_out, int *flag);
```

```
1
and
 2
typedef int MPI_Type_delete_attr_function(MPI_Datatype datatype,
 int type_keyval, void *attribute_val, void *extra_state);
 3
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
 a
 INTEGER :: type_keyval, ierror
 10
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
 11
 attribute_val_out
 12
 LOGICAL :: flag
 13
 14
and
 15
ABSTRACT INTERFACE
 16
 SUBROUTINE MPI_Type_delete_attr_function(datatype, type_keyval,
 17
 attribute_val, extra_state, ierror)
 18
 TYPE(MPI_Datatype) :: datatype
 19
 INTEGER :: type_keyval, ierror
 INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
 20
 21
With the mpi module and mpif.h, the Fortran callback functions are:
 22
SUBROUTINE TYPE_COPY_ATTR_FUNCTION(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE,
 23
 ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
 24
 INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
 25
 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
 26
 ATTRIBUTE_VAL_OUT
 27
 LOGICAL FLAG
 28
 29
and
 30
SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,
 31
 EXTRA_STATE, IERROR)
 32
 INTEGER DATATYPE, TYPE_KEYVAL, IERROR
 33
 INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
 34
 If an attribute copy function or attribute delete function returns other than
 35
MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_TYPE_FREE),
 36
is erroneous.
 37
 38
 39
MPI_TYPE_FREE_KEYVAL(type_keyval)
 40
 type_keyval
 INOUT
 key value (integer)
 41
 42
C binding
 43
int MPI_Type_free_keyval(int *type_keyval)
 44
 45
Fortran 2008 binding
 46
MPI_Type_free_keyval(type_keyval, ierror)
 47
 INTEGER, INTENT(INOUT) :: type_keyval
 48
```

```
1
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
 Fortran binding
3
 MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)
4
 INTEGER TYPE_KEYVAL, IERROR
5
6
7
 MPI_TYPE_SET_ATTR(datatype, type_keyval, attribute_val)
8
9
 INOUT
 datatype
 datatype to which attribute will be attached (handle)
10
 type_keyval
 IN
 key value (integer)
11
 IN
 attribute_val
 attribute value
12
13
14
 C binding
15
 int MPI_Type_set_attr(MPI_Datatype datatype, int type_keyval,
16
 void *attribute_val)
17
 Fortran 2008 binding
18
 MPI_Type_set_attr(datatype, type_keyval, attribute_val, ierror)
19
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
20
 INTEGER, INTENT(IN) :: type_keyval
21
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
22
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
^{24}
 Fortran binding
25
 MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)
26
 INTEGER DATATYPE, TYPE_KEYVAL, IERROR
27
 INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
28
29
30
 MPI_TYPE_GET_ATTR(datatype, type_keyval, attribute_val, flag)
^{31}
 IN
 datatype to which the attribute is attached (handle)
 datatype
32
33
 IN
 type_keyval
 key value (integer)
34
 OUT
 attribute_val
 attribute value, unless flag = false
35
 OUT
 false if no attribute is associated with the key
 flag
36
 (logical)
37
38
 C binding
39
 int MPI_Type_get_attr(MPI_Datatype datatype, int type_keyval,
40
 void *attribute_val, int *flag)
41
42
 Fortran 2008 binding
43
 MPI_Type_get_attr(datatype, type_keyval, attribute_val, flag, ierror)
44
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
45
 INTEGER, INTENT(IN) :: type_keyval
46
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
47
 LOGICAL, INTENT(OUT) :: flag
48
```

IN'	TEGER, OPTIONAL, INTENT(OUT	) :: ierror	1
Fortra	n binding		2 3
		KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)	4
	TEGER DATATYPE, TYPE_KEYVAL		5
	TEGER(KIND=MPI_ADDRESS_KIND	) ATTRIBUTE_VAL	6
LU	GICAL FLAG		7
			8
MPI TY	/PE_DELETE_ATTR(datatype, t		9 10
	<b>x - - -</b>	,	10
INOU [.]	51	datatype from which the attribute is deleted (handle)	12
IN	type_keyval	key value (integer)	13
~ • • •			14
C bind	0		15
int MP	I_Iype_delete_attr(MPI_Data	type datatype, int type_keyval)	16
	n 2008 binding		17 18
	pe_delete_attr(datatype, ty		19
	PE(MPI_Datatype), INTENT(IN		20
	<pre>FEGER, INTENT(IN) :: type_k FEGER, OPTIONAL, INTENT(OUT</pre>		21
			22
	n binding		23
	PE_DELETE_ATTR(DATATYPE, TY FEGER DATATYPE, TYPE_KEYVAL		24
T 1V	IEGEN DATATIFE, IFFE_REIVAL	, IERROR	25 26
			20
7.7.5	Error Class for Invalid Keyval		28
Key val	ues for attributes are system-all	ocated, by	29
		uch values can be passed to the functions that use	30
0	, e	er to signal that an erroneous key value has been	31
-		is a new MPI error class: MPI_ERR_KEYVAL. It can	32
	rned_by_MPI_ATTR_PUT,_MPI_ EYVAL_FREE,	_ATTR_GET, MPI_ATTR_DELETE,	33 34
	XXX}_DELETE_ATTR,		35
C.	$XX$ }_SET_ATTR,		36
C C	XX}_GET_ATTR,		37
MPI_{X	XXX}_FREE_KEYVAL, MPI_COM	MM_DUP, MPI_COMM_IDUP,	38
	÷ ,	COMM_FREE. The last four are included because	39
keyval i	s an argument to the copy and c	lelete functions for attributes.	40
770			41 42
7.7.6	Attributes Example		42
A	dvice to users. This example	e shows how to write a collective communication	44
_	. –	more efficient after the first call. (End of advice to	45
u	sers.)		46
			47

```
1
 /* key for this module's stuff: */
\mathbf{2}
 static int gop_key = MPI_KEYVAL_INVALID;
3
4
 typedef struct
5
 ſ
6
 int ref_count;
 /* reference count */
7
 /* other stuff, whatever else we want */
8
 } gop_stuff_type;
9
10
 void Efficient_Collective_Op(MPI_Comm comm, ...)
11
 {
12
 gop_stuff_type *gop_stuff;
13
 MPI_Group
 group;
14
 int
 foundflag;
15
16
 MPI_Comm_group(comm, &group);
17
18
 if (gop_key == MPI_KEYVAL_INVALID) /* get a key on first call ever */
19
 ſ
20
 if (! MPI_Comm_create_keyval(gop_stuff_copier,
21
 gop_stuff_destructor,
22
 &gop_key, NULL) {
23
 /* get the key while assigning its copy and delete callback
24
 behavior. */
25
 } else
26
 MPI_Abort(comm, 99);
27
 }
28
29
 MPI_Comm_get_attr(comm, gop_key, &gop_stuff, &foundflag);
30
 if (foundflag)
31
 { /* This module has executed in this group before.
32
 We will use the cached information */
33
 ł
34
 else
35
 { /* This is a group that we have not yet cached anything in.
36
 We will now do so.
37
 */
38
39
 /* First, allocate storage for the stuff we want,
40
 and initialize the reference count */
41
42
 gop_stuff = (gop_stuff_type *) malloc(sizeof(gop_stuff_type));
43
 if (gop_stuff == NULL) { /* abort on out-of-memory error */ }
44
45
 gop_stuff->ref_count = 1;
46
47
 /* Second, fill in *gop_stuff with whatever we want.
48
 This part isn't shown here */
```

```
/* Third, store gop_stuff as the attribute value */
 MPI_Comm_set_attr(comm, gop_key, gop_stuff);
 }
 /* Then, in any case, use contents of *gop_stuff
 to do the global op ... */
}
/* The following routine is called by MPI when a group is freed */
int gop_stuff_destructor(MPI_Comm comm, int keyval, void *gop_stuffP,
 void *extra)
{
 gop_stuff_type *gop_stuff = (gop_stuff_type *)gop_stuffP;
 if (keyval != gop_key) { /* abort -- programming error */ }
 /* The group's being freed removes one reference to gop_stuff */
 gop_stuff->ref_count -= 1;
 /* If no references remain, then free the storage */
 if (gop_stuff->ref_count == 0) {
 free((void *)gop_stuff);
 }
 return MPI_SUCCESS;
}
/* The following routine is called by MPI when a group is copied */
int gop_stuff_copier(MPI_Comm comm, int keyval, void *extra,
 void *gop_stuff_inP, void *gop_stuff_outP, int *flag)
{
 gop_stuff_type *gop_stuff_in = (gop_stuff_type *)gop_stuff_inP;
 gop_stuff_type **gop_stuff_out = (gop_stuff_type **)gop_stuff_outP;
 if (keyval != gop_key) { /* abort -- programming error */ }
 /* The new group adds one reference to this gop_stuff */
 gop_stuff_in->ref_count += 1;
 *gop_stuff_out = gop_stuff_in;
 return MPI_SUCCESS;
}
```

# 7.8 Naming Objects

There are many occasions on which it would be useful to allow a user to associate a printable identifier with an MPI communicator, window, or datatype, for instance error reporting, debugging, and profiling. The names attached to opaque objects do not propagate when the object is duplicated or copied by MPI routines. For communicators this can be achieved using the following two functions.

## Unofficial Draft for Comment Only

1 2

5 6

9

10 11

12

13 14

15

16 17

18

19

20 21

22

23 24

25

26

27

28 29

30 31

32

33

34

35 36

37 38

39 40 41

42 43

44

45

46

47

1 MPI_COMM_SET_NAME(comm, comm_name) 2 INOUT comm communicator whose identifier is to be set (handle) 3 IN the character string which is remembered as the comm_name 4 name (string) 56  $\overline{7}$ C binding 8 int MPI_Comm_set_name(MPI_Comm comm, const char *comm_name) 9 Fortran 2008 binding 10 MPI_Comm_set_name(comm, comm_name, ierror) 11 TYPE(MPI_Comm), INTENT(IN) :: comm 12CHARACTER(LEN=*), INTENT(IN) :: comm_name 13 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1415Fortran binding 16MPI_COMM_SET_NAME(COMM, COMM_NAME, IERROR) 17 INTEGER COMM, IERROR 18 CHARACTER*(*) COMM_NAME 19MPI_COMM_SET_NAME allows a user to associate a name string with a communicator. 20The character string which is passed to MPI_COMM_SET_NAME will be saved inside the 21MPI library (so it can be freed by the caller immediately after the call, or allocated on the 22 stack). Leading spaces in name are significant but trailing ones are not. 23MPI_COMM_SET_NAME is a local (non-collective) operation, which only affects the  24 name of the communicator as seen in the process which made the MPI_COMM_SET_NAME 25call. There is no requirement that the same (or any) name be assigned to a communicator 26in every process where it exists. 2728Advice to users. Since MPI_COMM_SET_NAME is provided to help debug code, it 29 is sensible to give the same name to a communicator in all of the processes where it 30 exists, to avoid confusion. (End of advice to users.)  31 32 The length of the name which can be stored is limited to the value of 33 MPI_MAX_OBJECT_NAME in Fortran and MPI_MAX_OBJECT_NAME-1 in C to allow for the 34null terminator. Attempts to put names longer than this will result in truncation of the 35 name. MPI_MAX_OBJECT_NAME must have a value of at least 64. 36 37 Advice to users. Under circumstances of store exhaustion an attempt to put a name 38 of any length could fail, therefore the value of MPI_MAX_OBJECT_NAME should be 39 viewed only as a strict upper bound on the name length, not a guarantee that setting 40 names of less than this length will always succeed. (End of advice to users.) 41 42Advice to implementors. Implementations which pre-allocate a fixed size space for a 43 name should use the length of that allocation as the value of MPI_MAX_OBJECT_NAME. 44 Implementations which allocate space for the name from the heap should still define 45MPI_MAX_OBJECT_NAME to be a relatively small value, since the user has to allocate 46space for a string of up to this size when calling MPI_COMM_GET_NAME. (End of 47 advice to implementors.) 48

MPI_COMM_GET_NAME(comm, comm_name, resultlen)				
IN	comm	communicator whose name is to be returned (handle)		
OUT	comm_name	the name previously stored on the communicator, or an empty string if no such name exists (string)		
OUT	resultlen	length of returned name (integer)		
	Comm_get_name(MPI_Comm con	nm, char *comm_name, int *resultlen)		
	2008 binding			
	get_name(comm, comm_name, MPI_Comm), INTENT(IN) ::			
	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				
CHARA	CHARACTER*(*) COMM_NAME			
		he last name which has previously been associated		
with the given communicator. The name may be set and retrieved from any language.				

wit 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 resulten cannot be larger than MPI_MAX_OBJECT_NAME.

If the user has not associated a name with a communicator, or an error occurs, MPI_COMM_GET_NAME will return an empty string (all spaces in Fortran, "" in C). The three predefined communicators will have predefined names associated with them. Thus, the names of MPI_COMM_WORLD, MPI_COMM_SELF, and the communicator returned by MPI_COMM_GET_PARENT (if not MPI_COMM_NULL) will have the default of MPI_COMM_WORLD, MPI_COMM_SELF, and MPI_COMM_PARENT. The fact that the system may have chosen to give a default name to a communicator does not prevent the user from setting a name on the same communicator; doing this removes the old name and assigns the new one.

*Rationale.* We provide separate functions for setting and getting the name of a communicator, rather than simply providing a predefined attribute key for the following reasons:

- It is not, in general, possible to store a string as an attribute from Fortran.
- It is not easy to set up the delete function for a string attribute unless it is known to have been allocated from the heap.

### 4546 47

48

1 2

3

4 5 6

7

9

10

11

18

23

 24 

25

26

27

28

29

30

 31 

32

33

34

35

36

37

38

39 40

41

42

```
1
 • To make the attribute key useful additional code to call strdup is necessary. If
2
 this is not standardized then users have to write it. This is extra unneeded work
3
 which we can easily eliminate.
4
 • The Fortran binding is not trivial to write (it will depend on details of the
5
 Fortran compilation system), and will not be portable. Therefore it should be in
6
 the library rather than in user code.
7
8
 (End of rationale.)
9
 Advice to users. The above definition means that it is safe simply to print the string
10
 returned by MPI_COMM_GET_NAME, as it is always a valid string even if there was
11
 no name.
12
13
 Note that associating a name with a communicator has no effect on the semantics of
14
 an MPI program, and will (necessarily) increase the store requirement of the program,
15
 since the names must be saved. Therefore there is no requirement that users use these
16
 functions to associate names with communicators. However debugging and profiling
17
 MPI applications may be made easier if names are associated with communicators,
18
 since the debugger or profiler should then be able to present information in a less
19
 cryptic manner. (End of advice to users.)
20
21
 The following functions are used for setting and getting names of datatypes. The
22
 constant MPI_MAX_OBJECT_NAME also applies to these names.
23
^{24}
 MPI_TYPE_SET_NAME(datatype, type_name)
25
26
 INOUT
 datatype
 datatype whose identifier is to be set (handle)
27
 IN
 the character string which is remembered as the
 type_name
28
 name (string)
29
30
 C binding
^{31}
 int MPI_Type_set_name(MPI_Datatype datatype, const char *type_name)
32
33
 Fortran 2008 binding
34
 MPI_Type_set_name(datatype, type_name, ierror)
35
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
36
 CHARACTER(LEN=*), INTENT(IN) :: type_name
37
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
 Fortran binding
39
 MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR)
40
 INTEGER DATATYPE, IERROR
41
 CHARACTER*(*) TYPE_NAME
42
43
44
45
46
47
48
```

IN	datatura	datatume mbaga name : t - 1
	datatype	datatype whose name is to be returned (handle)
OUT	type_name	the name previously stored on the datatype, or an empty string if no such name exists (string)
OUT	resultlen	length of returned name (integer)
C bindiı	ng	
nt MPI_	Type_get_name(MPI_Da int *resultlen)	atatype datatype, char *type_name, )
ortran	2008 binding	
	6	type_name, resultlen, ierror)
	C(MPI_Datatype), INTE	
		BJECT_NAME), INTENT(OUT) :: type_name
	GER, INTENT(OUT) ::	
TNIF	GER, OPTIONAL, INTEN	NI(UUI) :: lerror
	binding	
		TYPE_NAME, RESULTLEN, IERROR)
	GER DATATYPE, RESULT	LEN, IERRUR
CHAP	ACTER*(*) TYPE_NAME	
Nam	ed predefined datatypes	have the default names of the datatype name. For exam-
le, MPI_	WCHAR has the default	name of "MPI_WCHAR".
le, MPI_ The	WCHAR has the default is following functions are u	name of "MPI_WCHAR". used for setting and getting names of windows. The con-
le, MPI_ The	WCHAR has the default is following functions are u	name of "MPI_WCHAR".
le, MPI_ The tant MP	WCHAR has the default is following functions are u _MAX_OBJECT_NAME a	name of "MPI_WCHAR". used for setting and getting names of windows. The con- lso applies to these names.
le, MPI_ The tant MP	WCHAR has the default is following functions are u	name of "MPI_WCHAR". used for setting and getting names of windows. The con- lso applies to these names.
le, MPI_ The tant MP	WCHAR has the default is following functions are u _MAX_OBJECT_NAME a	name of "MPI_WCHAR". used for setting and getting names of windows. The con- lso applies to these names.
le, MPI_ The tant MP MPI_WIN	WCHAR has the default is following functions are u _MAX_OBJECT_NAME a I_SET_NAME(win, win_r	name of "MPI_WCHAR". used for setting and getting names of windows. The con- lso applies to these names.
le, MPI_ The tant MP MPI_WIN INOUT	WCHAR has the default is following functions are u _MAX_OBJECT_NAME a I_SET_NAME(win, win_u win	name of "MPI_WCHAR". used for setting and getting names of windows. The con- lso applies to these names. name) window whose identifier is to be set (handle)
le, MPI_ The tant MP 1PI_WIN INOUT	WCHAR has the default is following functions are u _MAX_OBJECT_NAME a I_SET_NAME(win, win_u win	name of "MPI_WCHAR". used for setting and getting names of windows. The con- lso applies to these names. name) window whose identifier is to be set (handle) the character string which is remembered as the
le, MPI_ The tant MP IPI_WIN INOUT IN C bindin	WCHAR has the default is following functions are u _MAX_OBJECT_NAME a I_SET_NAME(win, win_r win win_name	name of "MPI_WCHAR". used for setting and getting names of windows. The con- lso applies to these names. name) window whose identifier is to be set (handle) the character string which is remembered as the name (string)
le, MPI_ The tant MP IPI_WIN INOUT IN C bindin	WCHAR has the default is following functions are u _MAX_OBJECT_NAME a I_SET_NAME(win, win_r win win_name	name of "MPI_WCHAR". used for setting and getting names of windows. The con- lso applies to these names. name) window whose identifier is to be set (handle) the character string which is remembered as the
le, MPI_ The tant MP IPI_WIN INOUT IN C bindin nt MPI_	WCHAR has the default is following functions are u _MAX_OBJECT_NAME a I_SET_NAME(win, win_r win win_name	name of "MPI_WCHAR". used for setting and getting names of windows. The con- lso applies to these names. name) window whose identifier is to be set (handle) the character string which is remembered as the name (string)
le, MPI_ The cant MP IPI_WIN INOUT IN C bindin nt MPI_ Fortran PI_Win_	WCHAR has the default is following functions are u _MAX_OBJECT_NAME a I_SET_NAME(win, win_n win_win_name Min_set_name(MPI_Win 2008 binding set_name(win, win_name)	<pre>name of "MPI_WCHAR". used for setting and getting names of windows. The con- lso applies to these names. name) window whose identifier is to be set (handle) the character string which is remembered as the name (string) n win, const char *win_name) ame, ierror)</pre>
le, MPI_ The tant MP MPI_WIN INOUT IN C bindin nt MPI_ Cortran PI_Win_ TYPE	WCHAR has the default is following functions are u _MAX_OBJECT_NAME a L_SET_NAME(win, win_r win win_name Min_set_name(MPI_Win 2008 binding set_name(win, win_na C(MPI_Win), INTENT(IN	<pre>name of "MPI_WCHAR". used for setting and getting names of windows. The con- lso applies to these names. mame) window whose identifier is to be set (handle) the character string which is remembered as the name (string) me, ierror) l) :: win</pre>
le, MPI_ The tant MP MPI_WIN INOUT IN C bindin N C bindin TN C bindin TYPE CHAF	WCHAR has the default is following functions are u _MAX_OBJECT_NAME a L_SET_NAME(win, win_r win win_name Min_set_name(MPI_Win 2008 binding set_name(win, win_na (MPI_Win), INTENT(IN ACTER(LEN=*), INTENT	<pre>name of "MPI_WCHAR". used for setting and getting names of windows. The con- lso applies to these names. mame)</pre>
le, MPI_ The tant MP MPI_WIN INOUT IN C bindin nt MPI_ Cortran PI_Win_ TYPE CHAF	WCHAR has the default is following functions are u _MAX_OBJECT_NAME a L_SET_NAME(win, win_r win win_name Min_set_name(MPI_Win 2008 binding set_name(win, win_na C(MPI_Win), INTENT(IN	<pre>name of "MPI_WCHAR". used for setting and getting names of windows. The con- lso applies to these names. mame)</pre>
ele, MPI_ The tant MP MPI_WIN INOUT IN C bindin nt MPI_ Fortran IPI_Win_ TYPE CHAF INTE	WCHAR has the default is following functions are u _MAX_OBJECT_NAME a L_SET_NAME(win, win_r win win_name Min_set_name(MPI_Win 2008 binding set_name(win, win_na (MPI_Win), INTENT(IN ACTER(LEN=*), INTENT	<pre>name of "MPI_WCHAR". used for setting and getting names of windows. The con- lso applies to these names. mame)</pre>
ole, MPI_ The The tant MP MPI_WIN INOUT IN IN C bindin nt MPI_ Fortran PI_WIN_ Fortran INTE	WCHAR has the default is following functions are used _MAX_OBJECT_NAME a U_SET_NAME(win, win_n win_name Min_set_name(MPI_Win 2008 binding set_name(win, win_na C(MPI_Win), INTENT(IN EACTER(LEN=*), INTENT GER, OPTIONAL, INTENT binding SET_NAME(WIN, WIN_NA	<pre>name of "MPI_WCHAR". used for setting and getting names of windows. The con- lso applies to these names. name) window whose identifier is to be set (handle) the character string which is remembered as the name (string) n win, const char *win_name) ame, ierror) I) :: win C(IN) :: win_name UT(OUT) :: ierror</pre>
ole, MPI_ The The tant MP MPI_WIN INOUT IN INOUT IN C bindin IN TYPE CHAF INTE Fortran IPI_WIN_ INTE	WCHAR has the default is following functions are u _MAX_OBJECT_NAME a U_SET_NAME(win, win_r win_win_name Mg Win_set_name(MPI_Win 2008 binding set_name(win, win_na C(MPI_Win), INTENT(IN ACTER(LEN=*), INTENT GER, OPTIONAL, INTEN binding SET_NAME(WIN, WIN_NA GER WIN, IERROR	<pre>name of "MPI_WCHAR". used for setting and getting names of windows. The con- lso applies to these names. name) window whose identifier is to be set (handle) the character string which is remembered as the name (string) n win, const char *win_name) ame, ierror) I) :: win C(IN) :: win_name UT(OUT) :: ierror</pre>
ole, MPI_ The The tant MP MPI_WIN INOUT IN INOUT IN C bindin IN TYPE CHAF INTE Fortran IPI_WIN_ INTE	WCHAR has the default is following functions are used _MAX_OBJECT_NAME a U_SET_NAME(win, win_n win_name Min_set_name(MPI_Win 2008 binding set_name(win, win_na C(MPI_Win), INTENT(IN EACTER(LEN=*), INTENT GER, OPTIONAL, INTENT binding SET_NAME(WIN, WIN_NA	<pre>name of "MPI_WCHAR". used for setting and getting names of windows. The con- lso applies to these names. name) window whose identifier is to be set (handle) the character string which is remembered as the name (string) n win, const char *win_name) ame, ierror) I) :: win C(IN) :: win_name UT(OUT) :: ierror</pre>

MPI_WIN_GET_NAME(win, win_name, resultlen)

-		,	
2 3	IN	win	window whose name is to be returned (handle)
4 5	OUT	win_name	the name previously stored on the window, or an empty string if no such name exists (string)
6 7	OUT	resultlen	length of returned name (integer)
8 9	C binding		char *win_name, int *resultlen)
10 11 12		008 binding et_name(win, win_name, re	esultlen, ierror)
13	TYPE(	MPI_Win), INTENT(IN) :: v	vin
14 15		ER, INTENT(OUT) :: result	NAME), INTENT(OUT) :: win_name tlen
16 17		ER, OPTIONAL, INTENT(OUT)	) :: ierror
18	Fortran b	U	
19		ET_NAME(WIN, WIN_NAME, RI ER WIN, RESULTLEN, IERROI	
20		CTER*(*) WIN_NAME	·
21	OIMIN	OIDN: (') WIN_NAID	
22			

7.9 Formalizing the Loosely Synchronous Model

In this section, we make further statements about the loosely synchronous model, with particular attention to intra-communication.

7.9.1 Basic Statements

When a caller passes a communicator (that contains a context and group) to a callee, that 30 communicator must be free of side effects throughout execution of the subprogram: there  31 32 should be no active operations on that communicator that might involve the process. This provides one model in which libraries can be written, and work "safely." For libraries 33 34so designated, the callee has permission to do whatever communication it likes with the communicator, and under the above guarantee knows that no other communications will 35 interfere. Since we permit good implementations to create new communicators without 36 synchronization (such as by preallocated contexts on communicators), this does not impose 37 a significant overhead. 38

This form of safety is analogous to other common computer-science usages, such as passing a descriptor of an array to a library routine. The library routine has every right to expect such a descriptor to be valid and modifiable.

42 43

23

24 25

26

27 28

29

1

7.9.2 Models of Execution

⁴⁴ ⁴⁵ In the loosely synchronous model, transfer of control to a **parallel procedure** is effected by ⁴⁶ having each executing process invoke the procedure. The invocation is a collective operation: ⁴⁷ it is executed by all processes in the execution group, and invocations are similarly ordered ⁴⁸ at all processes. However, the invocation need not be synchronized.

We say that a parallel procedure is *active* in a process if the process belongs to a group that may collectively execute the procedure, and some member of that group is currently executing the procedure code. If a parallel procedure is active in a process, then this process may be receiving messages pertaining to this procedure, even if it does not currently execute the code of this procedure.

### Static Communicator Allocation

This covers the case where, at any point in time, at most one invocation of a parallel procedure can be active at any process, and the group of executing processes is fixed. For example, all invocations of parallel procedures involve all processes, processes are single-threaded, and there are no recursive invocations.

In such a case, a communicator can be statically allocated to each procedure. The static allocation can be done in a preamble, as part of initialization code. If the parallel procedures can be organized into libraries, so that only one procedure of each library can be concurrently active in each processor, then it is sufficient to allocate one communicator per library.

#### Dynamic Communicator Allocation

Calls of parallel procedures are well-nested if a new parallel procedure is always invoked in a subset of a group executing the same parallel procedure. Thus, processes that execute the same parallel procedure have the same execution stack.

In such a case, a new communicator needs to be dynamically allocated for each new invocation of a parallel procedure. The allocation is done by the caller. A new communicator can be generated by a call to MPI_COMM_DUP, if the callee execution group is identical to the caller execution group, or by a call to MPI_COMM_SPLIT if the caller execution group is split into several subgroups executing distinct parallel routines. The new communicator is passed as an argument to the invoked routine.

The need for generating a new communicator at each invocation can be alleviated or avoided altogether in some cases: If the execution group is not split, then one can allocate a stack of communicators in a preamble, and next manage the stack in a way that mimics the stack of recursive calls.

One can also take advantage of the well-ordering property of communication to avoid confusing caller and callee communication, even if both use the same communicator. To do so, one needs to abide by the following two rules:

- messages sent before a procedure call (or before a return from the procedure) are also received before the matching call (or return) at the receiving end;
- messages are always selected by source (no use is made of MPI_ANY_SOURCE).

### The General Case

In the general case, there may be multiple concurrently active invocations of the same parallel procedure within the same group; invocations may not be well-nested. A new communicator needs to be created for each invocation. It is the user's responsibility to make sure that, should two distinct parallel procedures be invoked concurrently on overlapping sets of processes, communicator creation is properly coordinated.

### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

5

6 7

8

9

10

11

12

13

14

15

16

17 18

19 20

21

22

23

 24 

25

26

27

28

29

30

 31 

32

33

34

35

36

37

38 39

40 41

42 43

44

45

46

47



# Chapter 8

# **Process Topologies**

### 8.1 Introduction

This chapter discusses the MPI topology mechanism. A topology is an extra, optional attribute that one can give to an intra-communicator; topologies cannot be added to intercommunicators. A topology can provide a convenient naming mechanism for the processes of a group (within a communicator), and additionally, may assist the runtime system in mapping the processes onto hardware.

As stated in Chapter 7, a process group in MPI is a collection of n processes. Each process in the group is assigned a rank between 0 and n-1. In many parallel applications a linear ranking of processes does not adequately reflect the logical communication pattern of the processes (which is usually determined by the underlying problem geometry and the numerical algorithm used). Often the processes are arranged in topological patterns such as two- or three-dimensional grids. More generally, the logical process arrangement is described by a graph. In this chapter we will refer to this logical process arrangement as the "virtual topology."

A clear distinction must be made between the virtual process topology and the topology of the underlying, physical hardware. The virtual topology can be exploited by the system in the assignment of processes to physical processors, if this helps to improve the communication performance on a given machine. How this mapping is done, however, is outside the scope of MPI. The description of the virtual topology, on the other hand, depends only on the application, and is machine-independent. The functions that are described in this chapter deal with machine-independent mapping and communication on virtual process topologies.

Rationale. Though physical mapping is not discussed, the existence of the virtual topology information may be used as advice by the runtime system. There are well-known techniques for mapping grid/torus structures to hardware topologies such as hypercubes or grids. For more complicated graph structures good heuristics often yield nearly optimal results [49]. On the other hand, if there is no way for the user to specify the logical process arrangement as a "virtual topology," a random mapping is most likely to result. On some machines, this will lead to unnecessary contention in the interconnection network. Some details about predicted and measured performance improvements that result from good process-to-processor mapping on modern wormhole-routing architectures can be found in [12, 13].

 24 

 31 

2

3

4 5

> 6 7

> > 8

9

10

11

12

13

14

15

25

26

27

35 36

37

38

39

Besides possible performance benefits, the virtual topology can function as a convenient, process-naming structure, with significant benefits for program readability and notational power in message-passing programming. (*End of rationale.*)

# 8.2 Virtual Topologies

The communication pattern of a set of processes can be represented by a graph. The nodes represent processes, and the edges connect processes that communicate with each other. MPI provides message-passing between any pair of processes in a group. There is no requirement for opening a channel explicitly. Therefore, a "missing link" in the user-defined process graph does not prevent the corresponding processes from exchanging messages. It means rather that this connection is neglected in the virtual topology. This strategy implies that the topology gives no convenient way of naming this pathway of communication. Another possible consequence is that an automatic mapping tool (if one exists for the runtime environment) will not take account of this edge when mapping.

16Specifying the virtual topology in terms of a graph is sufficient for all applications. 17However, in many applications the graph structure is regular, and the detailed set-up of the 18 graph would be inconvenient for the user and might be less efficient at run time. A large frac-19 tion of all parallel applications use process topologies like rings, two- or higher-dimensional 20grids, or tori. These structures are completely defined by the number of dimensions and 21the numbers of processes in each coordinate direction. Also, the mapping of grids and tori 22 is generally an easier problem than that of general graphs. Thus, it is desirable to address 23these cases explicitly. 24

Process coordinates in a Cartesian structure begin their numbering at 0. Row-major numbering is always used for the processes in a Cartesian structure. This means that, for example, the relation between group rank and coordinates for four processes in a  $(2 \times 2)$  grid is as follows.

rank 0
rank 1
rank $2$
$\operatorname{rank} 3$

# 8.3 Embedding in MPI

The support for virtual topologies as defined in this chapter is consistent with other parts of MPI, and, whenever possible, makes use of functions that are defined elsewhere. Topology information is associated with communicators. It is added to communicators using the caching mechanism described in Chapter 7.

40 Information representing an MPI virtual topology may be added to a communicator at 41 the time of its creation. If a communicator creation function adds information representing 42an MPI virtual topology to the output communicator it creates, then it either propagates 43 the topology representation from the input communicator to the output communicator, or 44adds a new topology representation generated from the input parameters that describe a 45virtual topology. The description of every MPI communicator creation function explicitly 46states how topology information is handled. Communicator creation functions that create 47new topology representations are described in Section 8.5. 48

## 8.4 Overview of the Functions

MPI supports three topology types: **Cartesian**, **graph**, and **distributed graph**. The function MPI_CART_CREATE can be used to create Cartesian topologies, the function MPI_GRAPH_CREATE can be used to create graph topologies, and the functions MPI_DIST_GRAPH_CREATE_ADJACENT and MPI_DIST_GRAPH_CREATE can be used to create distributed graph topologies. These topology creation functions are collective. As with other collective calls, the program must be written to work correctly, whether the call synchronizes or not.

The above topology creation functions take as input an existing communicator comm_old, which defines the set of processes on which the topology is to be mapped. For MPI_GRAPH_CREATE and MPI_CART_CREATE, all input arguments must have identical values on all processes of the group of comm_old. When calling MPI_GRAPH_CREATE, each process specifies all nodes and edges in the graph. In contrast, the functions MPI_DIST_GRAPH_CREATE_ADJACENT or MPI_DIST_GRAPH_CREATE are used to specify the graph in a distributed fashion, whereby each process only specifies a subset of the edges in the graph such that the entire graph structure is defined collectively across the set of processes. Therefore the processes provide different values for the arguments specifying the graph. However, all processes must give the same value for reorder and the info argument. In all cases, a new communicator comm_topol is created that carries the topological structure as cached information (see Chapter 7). In analogy to function MPI_COMM_CREATE, no cached information propagates from comm_old to comm_topol.

MPI_CART_CREATE can be used to describe Cartesian structures of arbitrary dimension. For each coordinate direction one specifies whether the process structure is periodic or not. Note that an *n*-dimensional hypercube is an *n*-dimensional torus with 2 processes per coordinate direction. Thus, special support for hypercube structures is not necessary. The local auxiliary function MPI_DIMS_CREATE can be used to compute a balanced distribution of processes among a given number of dimensions.

MPI defines functions to query a communicator for topology information. The function 29 MPI_TOPO_TEST is used to query for the type of topology associated with a communicator. 30 Depending on the topology type, different information can be extracted. For a graph 31topology, the functions MPI_GRAPHDIMS_GET and MPI_GRAPH_GET retrieve the graph-32 topology information that is associated with the communicator. Additionally, the functions 33 MPL GRAPH_NEIGHBORS_COUNT and MPL_GRAPH_NEIGHBORS can be used to obtain 34 the neighbors of an arbitrary node in the graph. For a distributed graph topology, the 35 functions MPI_DIST_GRAPH_NEIGHBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS 36 can be used to obtain the neighbors of the calling process. For a Cartesian topology, the 37 function MPI_CARTDIM_GET returns the number of dimensions and 38 MPI_CART_GET returns the numbers of MPI processes in each dimension and periodicity 39 of the associated Cartesian topology. Additionally, the functions MPI_CART_RANK and 40 MPI_CART_COORDS translate Cartesian coordinates into a group rank, and vice-versa. 41

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 Carte-

sian subspace (analogous to MPI_COMM_SPLIT). This function is collective over the input communicator's group.

The two additional functions, MPI_GRAPH_MAP and MPI_CART_MAP, are, in general, not called by the user directly. However, together with the communicator manipulation 48

### Unofficial Draft for Comment Only

1 2

3

4

 $\mathbf{5}$ 

6

 $\overline{7}$ 

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

 24 

25

26

27

28

42

43

44

45

12	functions presented in Chapter 7, they are sufficient to implement all other topology func- tions. Section 8.5.8 outlines such an implementation.				
3	The neighborhood collective communication routines MPI_NEIGHBOR_ALLGATHER,				
4	MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL,				
5	$MPI_NEIGHBOR_ALLTOALLV, \ \mathrm{and} \ MPI_NEIGHBOR_ALLTOALLW \ \mathrm{communicate} \ \mathrm{with} \ \mathrm{the}$				
6	nearest neighbors on the topology associated with the communicator. The nonblocking				
7 8	variants are MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and				
9		GHBOR_ALLTOALL, MFT_INE	IGHDOR_ALLIOALLY, and		
10					
11 12	8.5 Topology Constructors				
13 14	8.5.1 Ca	rtesian Constructor			
15					
16		C C DEATE (comm old ndime	dime pariode rearder comm cart)		
17		,	dims, periods, reorder, comm_cart)		
18 19	IN	comm_old	input communicator (handle)		
20	IN	ndims	number of dimensions of Cartesian grid (integer)		
21 22	IN	dims	integer array of size ndims specifying the number of processes in each dimension		
23 24	IN	periods	logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension		
25 26	IN	reorder	ranking may be reordered (true) or not (false) (logical)		
27 28	OUT	comm_cart	communicator with new Cartesian topology (handle)		
29 30	C binding	r i i i i i i i i i i i i i i i i i i i			
30 31 32		Cart_create(MPI_Comm comm	_old, int ndims, const int dims[], int reorder, MPI_Comm *comm_cart)		
33	Fortron 2	2008 binding			
34		J J J J J J J J J J J J J J J J J J J	dims, periods, reorder, comm_cart, ierror)		
35		(MPI_Comm), INTENT(IN) ::	-		
36 27	INTEC	ER, INTENT(IN) :: ndims,	dims(ndims)		
37 38		CAL, INTENT(IN) :: period			
39		(MPI_Comm), INTENT(OUT) :			
40	INTEG	ER, OPTIONAL, INTENT(OUT	) :: lerror		
41	Fortran k	0			
42			DIMS, PERIODS, REORDER, COMM_CART, IERROR)		
43 44		GER COMM_OLD, NDIMS, DIMS CAL PERIODS(*), REORDER	(*), CUMM_CARI, IEKKUK		
45					
46			dle to a new communicator to which the Cartesian $order = false$ then the rank of each process in the		
47			e old group. Otherwise, the function may reorder		

the processes (possibly so as to choose a good embedding of the virtual topology onto the physical machine). If the total size of the Cartesian grid is smaller than the size of the group of comm_old, then some processes are returned MPI_COMM_NULL, in analogy to MPI_COMM_SPLIT. If ndims is zero then a zero-dimensional Cartesian topology is created. The call is erroneous if it specifies a grid that is larger than the group size or if ndims is negative. MPI_CART_CREATE will associate information representing a Cartesian topology with the specified number of dimensions, numbers of MPI processes in each coordinate direction, and periodicity with the new communicator.

### 8.5.2 Cartesian Convenience Function: MPI_DIMS_CREATE

For Cartesian topologies, the function MPI_DIMS_CREATE helps the user select a balanced distribution of processes per coordinate direction, depending on the number of processes in the group to be balanced and optional constraints that can be specified by the user. One use is to partition all the processes (the size of MPI_COMM_WORLD's group) into an *n*-dimensional topology.

18 MPI_DIMS_CREATE(nnodes, ndims, dims) 19 number of nodes in a grid (integer) IN nnodes 2021IN ndims number of Cartesian dimensions (integer) 22 INOUT integer array of size ndims specifying the number of dims 23nodes in each dimension  24 25C binding 26int MPI_Dims_create(int nnodes, int ndims, int dims[]) 2728Fortran 2008 binding 29MPI_Dims_create(nnodes, ndims, dims, ierror) 30 INTEGER, INTENT(IN) :: nnodes, ndims 31 INTEGER, INTENT(INOUT) :: dims(ndims) 32 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 33 Fortran binding 34 MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR) 35INTEGER NNODES, NDIMS, DIMS(*), IERROR 36

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

$$\prod_{i,\mathsf{dims}[i]\neq 0}\mathsf{dims}[i]$$

1

 $\mathbf{2}$ 

3

4

5

6 7

8

9 10

11

12

13

14

15

16 17

389

### Unofficial Draft for Comment Only

46 47 48

37

38

39

40

41

42

43

44

For dims[i] set by the call, dims[i] will be ordered in non-increasing order. Array dims is suitable for use as input to routine MPI_CART_CREATE. MPI_DIMS_CREATE is local. If ndims is zero and nnodes is one, MPI_DIMS_CREATE returns MPI_SUCCESS.

Example	e 8.1		
	dims	function call	dims
	before call		on return
	(0,0)	MPI_DIMS_CREATE(6, 2, dims	
	(0,0)	MPI_DIMS_CREATE(7, 2, dims	
	(0,3,0)	MPI_DIMS_CREATE(6, 3, dims	
	(0,3,0)	MPI_DIMS_CREATE(7, 3, dims	
8.5.3 Gi	raph Constructor		
	/		
MPI_GRA	PH_CREATE(cor	mm_old, nnodes, index, edges, reo	rder, comm_graph)
IN	comm_old	input communicate	or (handle)
IN	nnodes	number of nodes in	ı graph (integer)
IN	index	array of integers de	escribing node degrees (see below
IN	edges	array of integers de	escribing graph edges (see below
IN	reorder	ranking may be red	ordered (true) or not (false)
		(logical)	
OUT	comm_graph	communicator with	n graph topology added (handle
C bindir	ıg		
int MPI_	-	PI_Comm comm_old, int nnode	
	const int	c edges[], int reorder, MPI_	Comm *comm_graph)
Fortran	2008 binding		
MPI_Grap	h_create(comm_	old, nnodes, index, edges,	reorder, comm_graph,
	ierror)		
		TENT(IN) :: comm_old	
		) :: nnodes, index(nnodes),	edges(*)
	CAL, INTENT(IN		
		TENT(OUT) :: comm_graph INTENT(OUT) :: ierror	
		INTENI(001) IEITOI	
Fortran			
MPI_GRAP		OLD, NNODES, INDEX, EDGES,	REORDER, COMM_GRAPH,
тмте	IERROR)	NNODES INDEX(*) EDCES(*)	
	CAL REORDER	NNODES, INDEX(*), EDGES(*),	COMMI_GRAPH, IERRUR
		E returns a handle to a new con	0
topology	information is at	ttached. If reorder $=$ false then	the rank of each process in

new group is identical to its rank in the old group. Otherwise, the function may reorder the

1

 2 

3

processes. If the size, nnodes, of the graph is smaller than the size of the group of comm_old, then some processes are returned MPI_COMM_NULL, in analogy to MPI_CART_CREATE and MPI_COMM_SPLIT. If the graph is empty, i.e., nnodes == 0, then MPI_COMM_NULL is returned in all processes. The call is erroneous if it specifies a graph that is larger than the group size of the input communicator.

The three parameters nnodes, index and edges define the graph structure. nnodes is the number of nodes of the graph. The nodes are numbered from 0 to nnodes-1. The i-th entry of array index stores the total number of neighbors of the first i graph nodes. The lists of neighbors of nodes 0, 1, ..., nnodes-1 are stored in consecutive locations in array edges. The array edges is a flattened representation of the edge lists. The total number of entries in index is nnodes and the total number of entries in edges is equal to the number of graph edges.

The definitions of the arguments nnodes, index, and edges are illustrated with the following simple example.

**Example 8.2** Assume there are four processes 0, 1, 2, 3 with the following adjacency matrix:

process	neighbors	
0	1, 3	
1	0	
2	3	
3	0, 2	

Then, the input arguments are:

edges = 1, 3, 0, 3, 0, 2

Thus, in C, index[0] is the degree of node zero, and index[i] - index[i-1] is the degree of node i, i=1, ..., nnodes-1; the list of neighbors of node zero is stored in edges[j], for  $0 \le j \le index[0] - 1$  and the list of neighbors of node i, i > 0, is stored in edges[j],  $index[i-1] \le j \le index[i] - 1$ .

In Fortran, index(1) is the degree of node zero, and index(i+1) - index(i) is the degree of node i, i=1, ..., nnodes-1; the list of neighbors of node zero is stored in edges(j), for  $1 \le j \le$  index(1) and the list of neighbors of node i, i > 0, is stored in edges(j), index(i)+1 \le j \le index(i+1).

A single process is allowed to be defined multiple times in the list of neighbors of a process (i.e., there may be multiple edges between two processes). A process is also allowed to be a neighbor to itself (i.e., a self loop in the graph). The adjacency matrix is allowed to be non-symmetric.

Advice to users. Performance implications of using multiple edges or a non-symmetric adjacency matrix are not defined. The definition of a node-neighbor edge does not imply a direction of the communication. (*End of advice to users.*)

*Advice to implementors.* The following topology information is likely to be stored with a communicator:

### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 31 

 $45 \\ 46$ 

1 • Type of topology (Cartesian/graph), 2 • For a Cartesian topology: 3 1. ndims (number of dimensions), 4 2. dims (numbers of processes per coordinate direction), 53. periods (periodicity information), 6 7 4. own_position (own position in grid, could also be computed from rank and 8 dims) 9 • For a graph topology: 10 1. index. 11 2. edges. 1213 which are the vectors defining the graph structure. 14For a graph structure the number of nodes is equal to the number of processes in 15the group. Therefore, the number of nodes does not have to be stored explicitly. 16An additional zero entry at the start of array index simplifies access to the topology 17 information. (End of advice to implementors.) 18 19 Distributed Graph Constructor 8.5.4 2021MPI_GRAPH_CREATE requires that each process passes the full (global) communication 22 graph to the call. This limits the scalability of this constructor. With the distributed graph 23interface, the communication graph is specified in a fully distributed fashion. Each process 24specifies only the part of the communication graph of which it is aware. Typically, this 25could be the set of processes from which the process will eventually receive or get data, 26or the set of processes to which the process will send or put data, or some combination of 27such edges. Two different interfaces can be used to create a distributed graph topology. 28MPI_DIST_GRAPH_CREATE_ADJACENT creates a distributed graph communicator with 29 each process specifying each of its incoming and outgoing (adjacent) edges in the logical 30 communication graph and thus requires minimal communication during creation.  31 MPI_DIST_GRAPH_CREATE provides full flexibility such that any process can indicate that 32 communication will occur between any pair of processes in the graph. 33 To provide better possibilities for optimization by the MPI library, the distributed 34graph constructors permit weighted communication edges and take an info argument that 35can further influence process reordering or other optimizations performed by the MPI library. 36 For example, hints can be provided on how edge weights are to be interpreted, the quality 37

- of the reordering, and/or the time permitted for the MPI library to process the graph.
- 39 40

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

MPI_DIST_GRAPH_CREATE_ADJACENT(comm_old, indegree, sources, sourceweights, outdegree, destinations, destweights, info, reorder, comm_dist_graph)

	outdegree, destinations, d	estweights, mo, reorder, comm_dist_graph)		
IN	comm_old	input communicator (handle)	3	
IN	indegree	size of sources and sourceweights arrays (non-negative integer)	4 5 6	
IN	sources	ranks of processes for which the calling process is a destination (array of non-negative integers)	7 8	
IN	sourceweights	weights of the edges into the calling process (array of non-negative integers)	9 10 11	
IN	outdegree	size of destinations and destweights arrays (non-negative integer)	12 13	
IN	destinations	ranks of processes for which the calling process is a source (array of non-negative integers)	14 15	
IN	destweights	weights of the edges out of the calling process (array of non-negative integers)	16 17 18	
IN	info	hints on optimization and interpretation of weights (handle)	19 20	
IN	reorder	the ranks may be reordered (true) or not (false) (logical)	21 22	
OUT	comm_dist_graph	communicator with distributed graph topology (handle)	23 24 25	
C binding				

### Fortran 2008 binding

MPI_Dist_graph_create_adjacent(comm_old, indegree, sources, sourceweights,	33			
outdegree, destinations, destweights, info, reorder,	34			
comm_dist_graph, ierror)	35			
TYPE(MPI_Comm), INTENT(IN) :: comm_old	36			
INTEGER, INTENT(IN) :: indegree, sources(indegree), sourceweights(*),	37			
<pre>outdegree, destinations(outdegree), destweights(*)</pre>	38			
TYPE(MPI_Info), INTENT(IN) :: info	39			
LOGICAL, INTENT(IN) :: reorder	40			
TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph	41			
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	42			
	43			
Fortran binding	44			
MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS,				
OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER,	46			

COMM_DIST_GRAPH, IERROR)

 $47 \\ 48$ 

5MPI_DIST_GRAPH_CREATE_ADJACENT returns a handle to a new communicator 6 to which the distributed graph topology information is attached. Each process passes all 7 information about its incoming and outgoing edges in the virtual distributed graph topology. 8 The calling processes must ensure that each edge of the graph is described in the source 9 and in the destination process with the same weights. If there are multiple edges for a given 10 (source,dest) pair, then the sequence of the weights of these edges does not matter. The 11 complete communication topology is the combination of all edges shown in the sources arrays 12of all processes in **comm_old**, which must be identical to the combination of all edges shown 13 in the destinations arrays. Source and destination ranks must be process ranks of comm_old. 14This allows a fully distributed specification of the communication graph. Isolated processes 15(i.e., processes with no outgoing or incoming edges, that is, processes that have specified 16indegree and outdegree as zero and thus do not occur as source or destination rank in the 17graph specification) are allowed. 18

The call creates a new communicator comm_dist_graph of distributed graph topology type to which topology information has been attached. The number of processes in comm_dist_graph is identical to the number of processes in comm_old. The call to MPI_DIST_GRAPH_CREATE_ADJACENT is collective.

- Weights are specified as non-negative integers and can be used to influence the process 23remapping strategy and other internal MPI optimizations. For instance, approximate count  24 arguments of later communication calls along specific edges could be used as their edge 25weights. Multiplicity of edges can likewise indicate more intense communication between 26pairs of processes. However, the exact meaning of edge weights is not specified by the MPI 27standard and is left to the implementation. In C or Fortran, an application can supply 28the special value MPI_UNWEIGHTED for the weight array to indicate that all edges have 29 the same (effectively no) weight. It is erroneous to supply MPI_UNWEIGHTED for some 30 but not all processes of comm_old. If the graph is weighted but indegree or outdegree is  31 zero, then MPI_WEIGHTS_EMPTY or any arbitrary array may be passed to sourceweights 32 or destweights respectively. Note that MPI_UNWEIGHTED and MPI_WEIGHTS_EMPTY are 33 not special weight values; rather they are special values for the total array argument. In 34Fortran, MPI_UNWEIGHTED and MPI_WEIGHTS_EMPTY are objects like MPI_BOTTOM (not 35 usable for initialization or assignment). See Section 2.5.4. 36
  - Advice to users. In the case of an empty weights array argument passed while constructing a weighted graph, one should not pass NULL because the value of MPI_UNWEIGHTED may be equal to NULL. The value of this argument would then be indistinguishable from MPI_UNWEIGHTED to the implementation. In this case MPI_WEIGHTS_EMPTY should be used instead. (*End of advice to users.*)
    - Advice to implementors. It is recommended that MPI_UNWEIGHTED not be implemented as NULL. (End of advice to implementors.)
- Rationale. To ensure backward compatibility, MPI_UNWEIGHTED may still be imple mented as NULL. See Annex B.3. (End of rationale.)

1

2

3

4

37

38

39

40

41

42 43

44

The meaning of the info and reorder arguments is defined in the description of the following routine.

MPI_DIST_GRAPH_CREATE(comm_old, n, sources,	degrees,	destinations,	weights,	info,
reorder, comm_dist_graph)				

		/	0
IN	comm_old	input communicator (handle)	7
IN	n	number of source nodes for which this process	8
		specifies edges (non-negative integer)	9 10
IN	sources	array containing the ${\sf n}$ source nodes for which this	11
		process specifies edges (array of non-negative	12
		integers)	13
IN	degrees	array specifying the number of destinations for each	14
		source node in the source node array (array of	15
		non-negative integers)	16
IN	destinations	destination nodes for the source nodes in the source	17 18
		node array (array of non-negative integers)	18
IN	weights	weights for source to destination edges (array of	20
	(	non-negative integers)	21
IN	info	hints on optimization and interpretation of weights	22
		(handle)	23
IN	reorder	the ranks may be reordered (true) or not (false)	24
		(logical)	
OUT	comm_dist_graph	communicator with distributed graph topology	
		added (handle)	28
			29
C binding			30
int MPI_D:	ist_graph_create(MPI_Comm	comm_old, int n, const int sources[],	31
	<u> </u>	<pre>const int destinations[],</pre>	32
	MP1_Comm *comm_dist_g	graph)	
	008 binding		
MPI_Dist_{			37
			38
			39
INIEGI		ces(ii), degrees(i), destinations(*),	40
TYPE()	0	info	41
	AL, INTENT(IN) :: reorder		
TYPE(1	MPI_Comm), INTENT(OUT) ::	comm_dist_graph	
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	45
Fortran b	inding		46
	6	SOURCES, DEGREES, DESTINATIONS, WEIGHTS,	47
	INFO, REORDER, COMM_I	DIST_GRAPH, IERROR)	48
IN OUT C binding int MPI_D: Fortran 20 MPI_Dist_8 TYPE(1) INTEGN TYPE(1) INTEGN TYPE(1) INTEGN	<pre>reorder comm_dist_graph ist_graph_create(MPI_Comm const int degrees[], const int weights[], MPI_Comm *comm_dist_ge 008 binding graph_create(comm_old, n, info, reorder, comm_co MPI_Comm), INTENT(IN) :: ER, INTENT(IN) :: n, sour weights(*) MPI_Info), INTENT(IN) :: AL, INTENT(IN) :: reorder MPI_Comm), INTENT(IN) :: ER, OPTIONAL, INTENT(OUT) inding GRAPH_CREATE(COMM_OLD, N,</pre>	<pre>hints on optimization and interpretation of weights (handle) the ranks may be reordered (true) or not (false) (logical) communicator with distributed graph topology added (handle) a comm_old, int n, const int sources[], const int destinations[], MPI_Info info, int reorder, graph) sources, degrees, destinations, weights, dist_graph, ierror) comm_old cces(n), degrees(n), destinations(*), info comm_dist_graph :: ierror SOURCES, DEGREES, DESTINATIONS, WEIGHTS,</pre>	22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47

1 $\mathbf{2}$ 

INTEGER COMM_OLD, N, SOURCES(*), DEGREES(*), DESTINATIONS(*), WEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR

LOGICAL REORDER

MPI_DIST_GRAPH_CREATE returns a handle to a new communicator to which the 5distributed graph topology information is attached. Concretely, each process calls the con-6 structor with a set of directed (source.destination) communication edges as described below. 7 Every process passes an array of n source nodes in the sources array. For each source node, a 8 non-negative number of destination nodes is specified in the degrees array. The destination 9 nodes are stored in the corresponding consecutive segment of the destinations array. More 10 precisely, if the i-th node in sources is s, this specifies degrees[i] edges (s,d) with d of the 11 j-th such edge stored in destinations[degrees[0]+ $\dots$ +degrees[i-1]+j]. The weight of this edge 12is stored in weights[degrees[0]+ $\ldots$ +degrees[i-1]+i]. Both the sources and the destinations 13 arrays may contain the same node more than once, and the order in which nodes are listed 14as destinations or sources is not significant. Similarly, different processes may specify edges 15with the same source and destination nodes. Source and destination nodes must be pro-16cess ranks of comm_old. Different processes may specify different numbers of source and 17destination nodes, as well as different source to destination edges. This allows a fully dis-18 tributed specification of the communication graph. Isolated processes (i.e., processes with 19no outgoing or incoming edges, that is, processes that do not occur as source or destination 20node in the graph specification) are allowed. 21

The call creates a new communicator comm_dist_graph of distributed graph topology type to which topology information has been attached. The number of processes in comm_dist_graph is identical to the number of processes in comm_old. The call to MPI_DIST_GRAPH_CREATE is collective.

If reorder = false, all processes will have the same rank in comm_dist_graph as in comm_old. If reorder = true then the MPI library is free to remap to other processes (of comm_old) in order to improve communication on the edges of the communication graph. The weight associated with each edge is a hint to the MPI library about the amount or intensity of communication on that edge, and may be used to compute a "best" reordering.

Weights are specified as non-negative integers and can be used to influence the process  31 remapping strategy and other internal MPI optimizations. For instance, approximate count 32 arguments of later communication calls along specific edges could be used as their edge 33 weights. Multiplicity of edges can likewise indicate more intense communication between 34pairs of processes. However, the exact meaning of edge weights is not specified by the MPI 35 standard and is left to the implementation. In C or Fortran, an application can supply 36 the special value MPI_UNWEIGHTED for the weight array to indicate that all edges have the 37 same (effectively no) weight. It is erroneous to supply MPI_UNWEIGHTED for some but not 38 all processes of comm_old. If the graph is weighted but n = 0, then MPI_WEIGHTS_EMPTY 39 or any arbitrary array may be passed to weights. Note that MPI_UNWEIGHTED and 40 MPI_WEIGHTS_EMPTY are not special weight values; rather they are special values for the 41 total array argument. In Fortran, MPI_UNWEIGHTED and MPI_WEIGHTS_EMPTY are objects 42like MPI_BOTTOM (not usable for initialization or assignment). See Section 2.5.4. 43

Advice to users. In the case of an empty weights array argument passed while
 constructing a weighted graph, one should not pass NULL because the value of
 MPI_UNWEIGHTED may be equal to NULL. The value of this argument would then
 be indistinguishable from MPI_UNWEIGHTED to the implementation.
 MPI_WEIGHTS_EMPTY should be used instead. (End of advice to users.)

## Unofficial Draft for Comment Only

1

2

3

Advice to implementors. It is recommended that MPI_UNWEIGHTED not be implemented as NULL. (End of advice to implementors.)

*Rationale.* To ensure backward compatibility, MPI_UNWEIGHTED may still be implemented as NULL. See Annex B.3. (*End of rationale.*)

The meaning of the weights argument can be influenced by the info argument. Info arguments can be used to guide the mapping; possible options include minimizing the maximum number of edges between processes on different SMP nodes, or minimizing the sum of all such edges. An MPI implementation is not obliged to follow specific hints, and it is valid for an MPI implementation not to do any reordering. An MPI implementation may specify more info key-value pairs. All processes must specify the same set of key-value info pairs.

Advice to implementors. MPI implementations must document any additionally supported key-value info pairs. MPI_INFO_NULL is always valid, and may indicate the default creation of the distributed graph topology to the MPI library.

An implementation does not explicitly need to construct the topology from its distributed parts. However, all processes can construct the full topology from the distributed specification and use this in a call to MPI_GRAPH_CREATE to create the topology. This may serve as a reference implementation of the functionality, and may be acceptable for small communicators. However, a scalable high-quality implementation would save the topology graph in a distributed way. (*End of advice to implementors.*)

**Example 8.3** As for Example 8.2, assume there are four processes 0, 1, 2, 3 with the following adjacency matrix and unit edge weights:

- 1		
	process	neighbors
	0	1, 3
	1	0
	2	3
	3	0, 2

With MPI_DIST_GRAPH_CREATE, this graph could be constructed in many different ways. One way would be that each process specifies its outgoing edges. The arguments per process would be:

process	n	sources	degrees	destinations	weights
0	1	0	2	1,3	1,1
1	1	1	1	0	1
2	1	2	1	3	1
3	1	3	2	0,2	1,1

Another way would be to pass the whole graph on process 0, which could be done with the following arguments per process:

process	n	sources	degrees	destinations	weights
0	4	0,1,2,3	2,1,1,2	1,3,0,3,0,2	$1,\!1,\!1,\!1,\!1,\!1,\!1$
1	0	-	-	-	-
2	0	-	-	-	-
3	0	-	-	-	

#### Unofficial Draft for Comment Only

 $\overline{7}$ 

In both cases above, the application could supply MPI_UNWEIGHTED instead of explicitly providing identical weights.

MPI_DIST_GRAPH_CREATE_ADJACENT	could be used to specify this graph using the
following arguments:	

process	indegree	sources	sourceweights	outdegree	destinations	destweights
0	2	1,3	1,1	2	1,3	1,1
1	1	0	1	1	0	1
2	1	3	1	1	3	1
3	2	0,2	1,1	2	0,2	1,1

10 11 12

13

14

15

16

17

1

 $\mathbf{2}$ 

> Example 8.4 A two-dimensional PxQ torus where all processes communicate along the dimensions and along the diagonal edges. This cannot be modeled with Cartesian topologies, but can easily be captured with MPI_DIST_GRAPH_CREATE as shown in the following code. In this example, the communication along the dimensions is twice as heavy as the communication along the diagonals:

```
/*
19
 Input:
 dimensions P, Q
20
 Condition: number of processes equal to P*Q; otherwise only
21
 ranks smaller than P*Q participate
22
 */
23
 int rank, x, y;
^{24}
 int sources[1], degrees[1];
25
 int destinations[8], weights[8];
26
 MPI_Comm comm_dist_graph;
27
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
28
29
30
 /* get x and y dimension */
^{31}
 y=rank/P; x=rank%P;
32
33
 /* get my communication partners along x dimension */
34
 destinations[0] = P*y+(x+1)%P; weights[0] = 2;
 destinations[1] = P*y+(P+x-1)%P; weights[1] = 2;
35
36
37
 /* get my communication partners along y dimension */
38
 destinations[2] = P*((y+1)%Q)+x; weights[2] = 2;
39
 destinations[3] = P*((Q+y-1)%Q)+x; weights[3] = 2;
40
41
 /* get my communication partners along diagonals */
42
 destinations[4] = P*((y+1)%Q)+(x+1)%P; weights[4] = 1;
43
 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;
44
45
 destinations[7] = P*((Q+y-1)%Q)+(P+x-1)%P; weights[7] = 1;
46
47
 sources[0] = rank;
48
 degrees [0] = 8;
```

MPI_Dist		ORLD, 1, sources, degrees, destinations, PI_INFO_NULL, 1, &comm_dist_graph);	1 2 3			
8.5.5 To	pology Inquiry Functions		4			
-	gy has been defined with one of bled up using inquiry function	of the above functions, then the topology information ns. They all are local calls.	5 6 7 8			
MPI_TOP	O_TEST(comm, status)		9 10			
IN	comm	communicator (handle)	10			
OUT	status	topology type of communicator comm (state)	12			
~			13 14			
C bindin	g Topo_test(MPI_Comm comm,	int *status)	15			
	2008 binding		16			
	_test(comm, status, ierro	or)	17 18			
	(MPI_Comm), INTENT(IN) :		19			
	GER, INTENT(OUT) :: stati		20			
	GER, OPTIONAL, INTENT(OU	1) :: lerror	21 22			
Fortran	6		23			
	_TEST(COMM, STATUS, IERRO GER COMM, STATUS, IERROR		24			
		returns the type of topology that is assigned to a	25 26			
communic		returns the type of topology that is assigned to a	20 27			
	output value <b>status</b> is one of t	the following:	28			
MPI_GR		graph topology	29			
MPI_GR		Cartesian topology	30 31			
	ST_GRAPH	distributed graph topology	32			
MPI_UN	DEFINED	no topology	33			
			34			
MPI GRA	PHDIMS_GET(comm, nnodes	, nedges)	35 36			
IN	comm	communicator for group with graph structure	37			
	comm	(handle)	38			
OUT	nnodes	number of nodes in graph (same as number of	39			
		processes in the group) (integer)	40 41			
OUT	nedges	number of edges in graph (integer)	42			
			43			
C bindin	0	· · · · · · · · · · · · · · · · · · ·	44			
		omm, int *nnodes, int *nedges)	$45 \\ 46$			
	2008 binding		47			
MP1_Grap.	MPI_Graphdims_get(comm, nnodes, nedges, ierror) 48					

```
1
 TYPE(MPI_Comm), INTENT(IN) :: comm
\mathbf{2}
 INTEGER, INTENT(OUT) :: nnodes, nedges
3
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
 Fortran binding
5
 MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR)
6
 INTEGER COMM, NNODES, NEDGES, IERROR
7
8
 The functions MPI_GRAPHDIMS_GET and MPI_GRAPH_GET retrieve the graph-topol-
9
 ogy information that is associated with the communicator. The information provided by
10
 MPI_GRAPHDIMS_GET can be used to dimension the vectors index and edges correctly for
11
 the following call to MPI_GRAPH_GET.
12
13
 MPI_GRAPH_GET(comm, maxindex, maxedges, index, edges)
14
15
 IN
 comm
 communicator with graph structure (handle)
16
 IN
 maxindex
 length of vector index in the calling program (integer)
17
 length of vector edges in the calling program (integer)
 IN
 maxedges
18
19
 OUT
 index
 array of integers containing the graph structure (for
 details see the definition of MPI_GRAPH_CREATE)
20
21
 OUT
 edges
 array of integers containing the graph structure
22
23
 C binding
24
 int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int index[],
25
 int edges[])
26
 Fortran 2008 binding
27
 MPI_Graph_get(comm, maxindex, maxedges, index, edges, ierror)
28
 TYPE(MPI_Comm), INTENT(IN) :: comm
29
 INTEGER, INTENT(IN) :: maxindex, maxedges
30
 INTEGER, INTENT(OUT) :: index(maxindex), edges(maxedges)
31
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33
 Fortran binding
34
 MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR)
35
 INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR
36
37
38
 MPI_CARTDIM_GET(comm, ndims)
39
40
 IN
 comm
 communicator with Cartesian structure (handle)
41
 OUT
 ndims
 number of dimensions of the Cartesian structure
42
 (integer)
43
44
 C binding
45
 int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
46
47
 Fortran 2008 binding
48
 MPI_Cartdim_get(comm, ndims, ierror)
```

TYPE	(MPI_Comm), INTENT(	IN) :: comm	1
	GER, INTENT(OUT) ::		2
INTE	GER, OPTIONAL, INTE	NT(OUT) :: ierror	3
<b>Fortran</b>	binding		4
	DIM_GET(COMM, NDIMS	, IERROR)	5 6
	GER COMM, NDIMS, IE		7
The f	weating MDL CADTO	M CET and MDI CAPT CET nature the Contagion tangl	8
		M_GET and MPI_CART_GET return the Cartesian topol- ed with the communicator. If comm is associated with a	9
		logy, MPI_CARTDIM_GET returns $ndims = 0$ and	10
	-	itput arguments unchanged.	11
			12
			13
MPI_CAR	T_GET(comm, maxdim	s, dims, periods, coords)	14
IN	comm	communicator with Cartesian structure (handle)	15
IN	maxdims	length of vectors dims, periods, and coords in the	16
		calling program (integer)	17 18
OUT	dims	number of processes for each Cartesian dimension	18
001	umo	(array of integers)	20
OUT	periods	periodicity (true/false) for each Cartesian dimension	21
001	perious	(array of logicals)	22
	oo ordo		23
OUT	coords	coordinates of calling process in Cartesian structure (array of integers)	24
		(array of integers)	25
C bindin	G		26
	0	omm, int maxdims, int dims[], int periods[],	27
1110 111 1_1	int coords[])	omm, int maxaime, int aimetj, int portoastj,	28 29
-			30
	2008 binding		31
		dims, periods, coords, ierror)	32
	(MPI_Comm), INTENT( GER, INTENT(IN) ::		33
		dims(maxdims), coords(maxdims)	34
	CAL, INTENT(OUT) ::		35
	GER, OPTIONAL, INTE	-	36
			37
Fortran	U		38
		DIMS, PERIODS, COORDS, IERROR)	39
		DIMS(*), COORDS(*), IERROR	40
LUGI	CAL PERIODS(*)		41
			42
			43
			$44 \\ 45$
			45 46
			47

```
1
 MPI_CART_RANK(comm, coords, rank)
2
 IN
 comm
 communicator with Cartesian structure (handle)
3
 IN
 coords
 integer array (of size ndims) specifying the Cartesian
4
 coordinates of a process
5
6
 OUT
 rank
 rank of specified process (integer)
7
8
 C binding
9
 int MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank)
10
 Fortran 2008 binding
11
 MPI_Cart_rank(comm, coords, rank, ierror)
12
 TYPE(MPI_Comm), INTENT(IN) :: comm
13
 INTEGER, INTENT(IN) :: coords(*)
14
 INTEGER, INTENT(OUT) :: rank
15
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
 Fortran binding
18
 MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
19
 INTEGER COMM, COORDS(*), RANK, IERROR
20
 For a communicator with an associated Cartesian topology, the function
21
 MPI_CART_RANK translates the logical process coordinates to process ranks. For dimen-
22
 sion i with periods(i) = true, if the coordinate, coords(i), is out of range, that is, coords(i) < 0
23
 or coords(i) \geq dims(i), it is shifted back to the interval 0 \leq coords(i) \leq dims(i) automatically.
^{24}
 Out-of-range coordinates are erroneous for non-periodic dimensions.
25
26
 If comm is associated with a zero-dimensional Cartesian topology, coords is not signif-
 icant and 0 is returned in rank.
27
28
29
 MPI_CART_COORDS(comm, rank, maxdims, coords)
30
 IN
^{31}
 communicator with Cartesian structure (handle)
 comm
32
 IN
 rank of a process within group of comm (integer)
 rank
33
 IN
 maxdims
 length of vector coords in the calling program
34
 (integer)
35
 OUT
 coords
 integer array (of size maxdims) containing the
36
37
 Cartesian coordinates of specified process (array of
38
 integers)
39
40
 C binding
41
 int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])
42
 Fortran 2008 binding
43
 MPI_Cart_coords(comm, rank, maxdims, coords, ierror)
44
 TYPE(MPI_Comm), INTENT(IN) :: comm
45
 INTEGER, INTENT(IN) :: rank, maxdims
46
 INTEGER, INTENT(OUT) :: coords(maxdims)
47
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

Fortran b	inding		1
MPI_CART_	COORDS(COMM, RANK, MAXDIM	MS, COORDS, IERROR)	2
INTEG	ER COMM, RANK, MAXDIMS, C	COORDS(*), IERROR	3
The ir	verse manning rank-to-coor	dinates translation is provided by	4
		iated with a zero-dimensional Cartesian topology,	5
	be unchanged.	autor with a zero annensional cartesian topology,	6
	se anonangea.		7
			8 9
MPI_GRAF	PH_NEIGHBORS_COUNT(com	nm, rank, nneighbors)	9 10
IN	comm	communicator with graph topology (handle)	11
IN	rank	rank of process in group of $comm$ (integer)	12
OUT	nneighbors	number of neighbors of specified process (integer)	13
			14
C binding	5		15 16
int MPI_G	raph_neighbors_count(MPI_	_Comm comm, int rank, int *nneighbors)	10
Fortran 2	008 binding		18
	_neighbors_count(comm, ra	ank, nneighbors, jerror)	19
-	<pre>MPI_Comm), INTENT(IN) ::</pre>	-	20
	ER, INTENT(IN) :: rank		21
INTEG	ER, INTENT(OUT) :: nneigh	nbors	22
	ER, OPTIONAL, INTENT(OUT)		23
Fortran b	inding		24
	_NEIGHBORS_COUNT(COMM, RA	ANK NNETCHBORS TERROR)	25
	ER COMM, RANK, NNEIGHBORS		26
11120			27
			28
MPI GRAF	PH_NEIGHBORS(comm, rank,	maxneighbors, neighbors)	29 30
IN	comm	communicator with graph topology (handle)	31
			32
IN	rank	rank of process in group of <b>comm</b> (integer)	33
IN	maxneighbors	size of array neighbors (integer)	34
OUT	neighbors	ranks of processes that are neighbors to specified	35
		process (array of integers)	36
			37
C binding	ç 🔰		38
int MPI_G	raph_neighbors(MPI_Comm o	comm, int rank, int maxneighbors,	39
	<pre>int neighbors[])</pre>		40
Fortran 2	008 binding		41 42
	6	axneighbors, neighbors, ierror)	43
-	MPI_Comm), INTENT(IN) ::		44
	ER, INTENT(IN) :: rank, n		45
	ER, INTENT(OUT) :: neight	-	46
	ER, OPTIONAL, INTENT(OUT)	-	47
			48

**Unofficial Draft for Comment Only** 

1	Fortran binding
2	MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
3	INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR
4	
5	MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS provide adjacency
6	information for a graph topology. The returned count and array of neighbors for the queried
7	rank will both include <i>all</i> neighbors and reflect the same edge ordering as was specified by
8	the original call to MPI_GRAPH_CREATE. Specifically, MPI_GRAPH_NEIGHBORS_COUNT
9	and MPI_GRAPH_NEIGHBORS will return values based on the original index and edges array
10	passed to MPI_GRAPH_CREATE (for the purpose of this example, we assume that index[-1]
11	is zero):
12	• The number of neighbors ( <b>nneighbors</b> ) returned from
13	MPI_GRAPH_NEIGHBORS_COUNT will be (index[rank] - index[rank-1]).
14	
15	• The neighbors array returned from MPI_GRAPH_NEIGHBORS will be
16	edges[index[rank-1]] through edges[index[rank]-1].
17 18	
19	<b>Example 8.5</b> Assume there are four processes 0, 1, 2, 3 with the following adjacency
20	matrix (note that some neighbors are listed multiple times):
21	
22	process neighbors
23	0 1, 1, 3
24	1  0, 0
25	$\begin{array}{c c} 2 & 3 \\ 2 & 2 & 2 \end{array}$
26	$3 \qquad 0, 2, 2$
27	Thus, the input arguments to MPI_GRAPH_CREATE are:
28 29	
30	$\begin{array}{rl} \mathrm{nnodes} = & 4 \\ \mathrm{index} = & 3, 5, 6, 9 \end{array}$
31	edges = 1, 1, 3, 0, 0, 3, 0, 2, 2
32	
33	Therefore, calling MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS for
34	each of the 4 processes will return:
35	Input rank Count Neighbors
36	$\frac{11 \text{ put tail } \text{ could real real } \text{ real }  r$
37	1   2   0, 0
38 39	2 1 $3$
40	3    3    0, 2, 2
41	
42	<b>Example 9.6</b> Suppose that comp is a communicator with a shuffle analogue topologue
43	<b>Example 8.6</b> Suppose that comm is a communicator with a shuffle-exchange topology. The group has $2^n$ members. Each process is labeled by $a_1, \ldots, a_n$ with $a_i \in \{0, 1\}$ , and has
44	three neighbors: exchange $(a_1, \ldots, a_n) = a_1, \ldots, a_{n-1}, \bar{a}_n$ ( $\bar{a} = 1 - a$ ), shuffle $(a_1, \ldots, a_n) = a_1, \ldots, a_{n-1}, \bar{a}_n$
45	$a_2, \ldots, a_n, a_1$ , and $unshuffle(a_1, \ldots, a_n) = a_1, \ldots, a_{n-1}, a_n$ (a = 1 – a), shuffle( $a_1, \ldots, a_n$ ) = $a_2, \ldots, a_n, a_1$ , and $unshuffle(a_1, \ldots, a_n) = a_n, a_1, \ldots, a_{n-1}$ . The graph adjacency list is
46	illustrated below for $n = 3$ .
47	
48	

node		exchange	shuffle	unshuffle
		neighbors(1)	neighbors(2)	neighbors(3)
0	(000)	1	0	0
1	(001)	0	2	4
2	(010)	3	4	1
3	(011)	2	6	5
4	(100)	5	1	2
5	(101)	4	3	6
6	(110)	7	5	3
7	(111)	6	7	7

Suppose that the communicator **comm** has this topology associated with it. The following code fragment cycles through the three types of neighbors and performs an appropriate permutation for each.

```
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)
			_000111	(0011111,	macgree,	outdegree,	meighted)

			34
IN	comm	communicator with distributed graph topology	35
		(handle)	36
OUT	indegree	number of edges into this process (non-negative	37
		integer)	38
OUT	outdegree	number of edges out of this process (non-negative	39
001	outdegree	integer)	40
		- , ,	41
OUT	weighted	false if MPI_UNWEIGHTED was supplied during	42
		creation, true otherwise (logical)	43
			44
$\alpha$ $1 \cdot 1 \cdot$			

# C binding int MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree, int *outdegree, int *weighted) 45 46 47 48

 24 

1 2 3 4 5 6 7 8 9 10 11 12 13	MPI_Dist TYPE INTEC LOGIC INTEC Fortran M MPI_DIST INTEC	(MPI_Comm), INTENT(IN GER, INTENT(OUT) :: i CAL, INTENT(OUT) :: w GER, OPTIONAL, INTENT binding	Indegree, outdegree weighted C(OUT) :: ierror WT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR)
13 14 15	MPI_DIST		comm, maxindegree, sources, sourceweights, inations, destweights)
16 17	IN	comm	communicator with distributed graph topology (handle)
18 19 20	IN	maxindegree	size of sources and sourceweights arrays (non-negative integer)
21 22	OUT	sources	processes for which the calling process is a destination (array of non-negative integers)
23 24 25	OUT	sourceweights	weights of the edges into the calling process (array of non-negative integers)
26 27	IN	maxoutdegree	size of destinations and destweights arrays (non-negative integer)
28 29	OUT	destinations	processes for which the calling process is a source (array of non-negative integers)
30 31 32	OUT	destweights	weights of the edges out of the calling process (array of non-negative integers)
<ul> <li>33</li> <li>34</li> <li>35</li> <li>36</li> <li>37</li> <li>38</li> <li>39</li> <li>40</li> <li>41</li> <li>42</li> <li>43</li> <li>44</li> <li>45</li> <li>46</li> <li>47</li> <li>48</li> </ul>	Fortran 2 MPI_Dist_ TYPE INTEC INTEC	Dist_graph_neighbors( int sourceweight int destweights 2008 binding _graph_neighbors(comm maxoutdegree, do (MPI_Comm), INTENT(IN GER, INTENT(IN) :: ma	n, maxindegree, sources, sourceweights, estinations, destweights, ierror) N) :: comm axindegree, maxoutdegree sources(maxindegree), axoutdegree) (*), destweights(*)

 24 

#### Fortran binding

MPI_DIST_GRAPH_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, SOURCEWEIGHTS,
MAXOUTDEGREE, DESTINATIONS, DESTWEIGHTS, IERROR)
INTEGER COMM, MAXINDEGREE, SOURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE,
<pre>DESTINATIONS(*), DESTWEIGHTS(*), IERROR</pre>

These calls are local. The number of edges into and out of the process returned by MPI_DIST_GRAPH_NEIGHBORS_COUNT are the total number of such edges given in the call to MPI_DIST_GRAPH_CREATE_ADJACENT or MPI_DIST_GRAPH_CREATE (potentially by processes other than the calling process in the case of MPI_DIST_GRAPH_CREATE). Multiply-defined edges are all counted and returned by MPI_DIST_GRAPH_NEIGHBORS in some order. If MPI_UNWEIGHTED is supplied for sourceweights or destweights or both, or if MPI_UNWEIGHTED was supplied during the construction of the graph then no weight information is returned in that array or those arrays. If the communicator was created with MPI_DIST_GRAPH_CREATE_ADJACENT then for each rank in comm, the order of the values in sources and destinations is identical to the input that was used by the process with the same rank in comm_old in the creation call. If the communicator was created with MPI_DIST_GRAPH_CREATE then the only requirement on the order of values in sources and destinations is that two calls to the routine with same input argument comm will return the same sequence of edges. If maxindegree or maxoutdegree is smaller than the numbers returned by MPI_DIST_GRAPH_NEIGHBORS_COUNT, then only the first part of the full list is returned.

Advice to implementors. Since the query calls are defined to be local, each process needs to store the list of its neighbors with incoming and outgoing edges. Communication is required at the collective MPI_DIST_GRAPH_CREATE call in order to compute the neighbor lists for each process from the distributed graph specification. (*End of advice to implementors.*)

#### 8.5.6 Cartesian Shift Coordinates

If the process topology is a Cartesian structure, an MPI_SENDRECV operation may be used along a coordinate direction to perform a shift of data. As input, MPI_SENDRECV takes the rank of a source process for the receive, and the rank of a destination process for the send. If the function MPI_CART_SHIFT is called for a Cartesian process group, it provides the calling process with the above identifiers, which then can be passed to MPI_SENDRECV. The user specifies the coordinate direction and the size of the step (positive or negative, but not zero). The function is local.

```
1
 MPI_CART_SHIFT(comm, direction, disp, rank_source, rank_dest)
2
 IN
 comm
 communicator with Cartesian structure (handle)
3
 IN
 direction
 coordinate dimension of shift (integer)
4
5
 disp
 IN
 displacement (> 0: upwards shift, < 0: downwards
6
 shift) (integer)
7
 OUT
 rank_source
 rank of source process (integer)
8
 OUT
 rank_dest
 rank of destination process (integer)
9
10
11
 C binding
 int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
12
 int *rank_source, int *rank_dest)
13
14
 Fortran 2008 binding
15
 MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror)
16
 TYPE(MPI_Comm), INTENT(IN) :: comm
17
 INTEGER, INTENT(IN) :: direction, disp
18
 INTEGER, INTENT(OUT) :: rank_source, rank_dest
19
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
 Fortran binding
21
 MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)
22
 INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR
23
24
 The direction argument indicates the coordinate dimension to be traversed by the shift.
25
 The dimensions are numbered from 0 to ndims-1, where ndims is the number of dimensions.
26
 Depending on the periodicity of the Cartesian group in the specified coordinate direc-
27
 tion, MPI_CART_SHIFT provides the identifiers for a circular or an end-off shift. In the case
28
 of an end-off shift, the value MPI_PROC_NULL may be returned in rank_source or rank_dest,
29
 indicating that the source or the destination for the shift is out of range.
30
 It is erroneous to call MPI_CART_SHIFT with a direction that is either negative or
^{31}
 greater than or equal to the number of dimensions in the Cartesian communicator. This
32
 implies that it is erroneous to call MPI_CART_SHIFT with a comm that is associated with
33
 a zero-dimensional Cartesian topology.
34
35
 Example 8.7 The communicator, comm, has a two-dimensional, periodic, Cartesian topol-
36
 ogy associated with it. A two-dimensional array of REALs is stored one element per process,
37
 in variable A. One wishes to skew this array, by shifting column i (vertically, i.e., along the
38
 column) by i steps.
39
40
41
 ! find process rank
42
 CALL MPI_COMM_RANK(comm, rank, ierr)
43
 ! find Cartesian coordinates
44
 CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
45
 ! compute shift source and destination
46
 CALL MPI_CART_SHIFT(comm, 0, coords(2), source, dest, ierr)
47
 ! skew array
48
 CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0, source, 0, comm, &
```

	sta	tus, ierr)	1
DIM dim	S(i+1) nodes, where DIM	the dimension indicated by DIRECTION = i has S is the array that was used to create the grid. In C, the ction = i is the dimension specified by dims[i]. ( <i>End</i>	2 3 4 5 6 7
8.5.7 P	artitioning of Cartesian S	tructures	8 9 10
	RT_SUB(comm, remain_di	ms newcomm)	11
	,	,	12
IN	comm	communicator with Cartesian structure (handle)	13
IN	remain_dims	the i-th entry of remain_dims specifies whether the i-th dimension is kept in the subgrid (true) or is dropped (false) (array of logicals)	14 15 16
OUT	newcomm	communicator containing the subgrid that includes the calling process (handle)	17 18 19
C bindin int MPI_	0	m, const int remain_dims[], MPI_Comm *newcomm)	20 21 22
Fortran	2008 binding		23
	_sub(comm, remain_dim		24
	E(MPI_Comm), INTENT(IN		25 26
	ICAL, INTENT(IN) :: re E(MPI_Comm), INTENT(OU		27
	EGER, OPTIONAL, INTENT		28
			29
	binding [_SUB(COMM, REMAIN_DIM		30
	EGER COMM, NEWCOMM, IE		31
	ICAL REMAIN_DIMS(*)		32
			33 34
		to partition the group associated with a communica-	35
		an topology into subgroups that form lower-dimensional cach subgroup a communicator with the associated sub-	36
		logies of the new communicators describe the subgrids.	37
-		subgrids is the number of remaining dimensions, i.e., the	38
		ims. The numbers of MPI processes in each coordinate	39
		maining numbers of MPI processes in each coordinate di-	40
	_	the original communicator, i.e., the values of the original	41
	_	responding entry in remain_dims is true. The periodic-	42
ity for th	e remaining dimensions i	n the new communicator is preserved from the original	43
		nain_dims are false or comm is already associated with a	44
		gy then newcomm is associated with a zero-dimensional	45
Cartesiar	topology. (This function	is closely related to MPI_COMM_SPLIT.)	46 47

1 2 3	<b>Example 8.8</b> Assume that MPI_Cart_create(, comm) has defined a $(2 \times 3 \times 4)$ grid. Let remain_dims = (true, false, true). Then a call to				
4	MPI_Cart_sub(comm, remain_dims, newcomm)				
5 6 7	will create three communicators each with eight processes in a $2 \times 4$ Cartesian topology. If remain_dims = (false, false, true) then the call to				
8 9	MPI_Ca	rt_sub(comm, remain_dims, ne	ewcomm)		
10 11 12	will creates Cartesian t		eators, each with four processes, in a one-dimensional		
13 14	8.5.8 Lov	v-Level Topology Functions			
15 16 17 18 19	topology fu	unctions. In general they will additional virtual topology ca	in this section can be used to implement all other not be called by the user directly, unless he or she pability other than that provided by MPI. The two		
20 21	MPI_CART		eriods, newrank)		
22	IN	comm	input communicator (handle)		
23	IN	ndims	number of dimensions of Cartesian structure (integer)		
24 25 26	IN	dims	integer array of size ndims specifying the number of processes in each coordinate direction		
27 28	IN	periods	logical array of size ndims specifying the periodicity specification in each coordinate direction		
29 30 31 32	OUT	newrank	reordered rank of the calling process; MPI_UNDEFINED if calling process does not belong to grid (integer)		
33	C binding				
34 35		·	nt ndims, const int dims[],		
35		const int periods[],			
37	Fortran 2	008 binding			
38		map(comm, ndims, dims, pe	eriods, newrank, ierror)		
39	TYPE(	MPI_Comm), INTENT(IN) ::	comm		
40	INTEGER, INTENT(IN) :: ndims, dims(ndims)				
41		AL, INTENT(IN) :: periods			
42		ER, INTENT(OUT) :: newrai			
43 44	INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror		
45	Fortran b	inding			
46		MAP(COMM, NDIMS, DIMS, PH	ERIODS, NEWRANK, IERROR)		
47	INTEG	ER COMM, NDIMS, DIMS(*),	NEWRANK, IERROR		
48	LOGIC	AL PERIODS(*)			

MPI_CART_MAP computes an "optimal" placement for the calling process on the physical machine. A possible implementation of this function is to always return the rank of the calling process, that is, not to perform any reordering.

4 The function MPI_CART_CREATE(comm, ndims, dims, Advice to implementors. 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 9 is zero then a zero-dimensional Cartesian topology is created. 10 11 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 1213encoding of the lost dimensions as color and a single number encoding of the preserved 14dimensions as key. 15All other Cartesian topology functions can be implemented locally, using the topology 16information that is cached with the communicator. (End of advice to implementors.) 1718 The corresponding function for graph structures is as follows. 19 20MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank) 2122 IN input communicator (handle) comm 23IN nnodes number of graph nodes (integer)  24 IN index integer array specifying the graph structure, see 25MPI_GRAPH_CREATE 2627IN edges integer array specifying the graph structure 28OUT reordered rank of the calling process; newrank 29MPI_UNDEFINED if the calling process does not 30 belong to graph (integer) 3132 C binding 33 int MPI_Graph_map(MPI_Comm comm, int nnodes, const int index[], 34 const int edges[], int *newrank) 3536 Fortran 2008 binding 37 MPI_Graph_map(comm, nnodes, index, edges, newrank, ierror) 38 TYPE(MPI_Comm), INTENT(IN) :: comm 39 INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*) 40 INTEGER, INTENT(OUT) :: newrank 41 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 42Fortran binding 43 MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR) 44INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR 4546

Advice to implementors. The function MPI_GRAPH_CREATE(comm, nnodes, index, edges, reorder, comm_graph), with reorder = true can be implemented by calling

1  $\mathbf{2}$ 

3

5

6

7

8

47

MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank), then calling MPI_COMM_SPLIT(comm, color, key, comm_graph), with color = 0 if newrank  $\neq$ MPI_UNDEFINED, color = MPI_UNDEFINED otherwise, and key = newrank.

All other graph topology functions can be implemented locally, using the topology information that is cached with the communicator. (End of advice to implementors.)

#### Neighborhood Collective Communication on Process Topologies 8.6

MPI process topologies specify a communication graph, but they implement no commu-10 nication function themselves. Many applications require sparse nearest neighbor communications that can be expressed as graph topologies. We now describe several collective 12operations that perform communication along the edges of a process topology. All of these 13 functions are collective; i.e., they must be called by all processes in the specified commu-14nicator. See Section 6 for an overview of other dense (global) collective communication 15operations and the semantics of collective operations.

16If the graph was created with MPI_DIST_GRAPH_CREATE_ADJACENT with sources 17and destinations containing  $0, \ldots, n-1$ , where n is the number of processes in the group 18 of comm_old (i.e., the graph is fully connected and also includes an edge from each node 19 to itself), then the sparse neighborhood communication routine performs the same data 20exchange as the corresponding dense (fully-connected) collective operation. In the case of a 21Cartesian communicator, only nearest neighbor communication is provided, corresponding 22 to rank_source and rank_dest in MPI_CART_SHIFT with input disp = 1. 23

 24 

25

26

27

28

29

30

Neighborhood collective communications enable communication on a Rationale. process topology. This high-level specification of data exchange among neighboring processes enables optimizations in the MPI library because the communication pattern is known statically (the topology). Thus, the implementation can compute optimized message schedules during creation of the topology [39]. This functionality can significantly simplify the implementation of neighbor exchanges [35]. (End of rationale.)

 31 For a distributed graph topology, created with MPI_DIST_GRAPH_CREATE, the se-32 quence of neighbors in the send and receive buffers at each process is defined as the sequence 33 returned by MPI_DIST_GRAPH_NEIGHBORS for destinations and sources, respectively. For 34a general graph topology, created with MPI_GRAPH_CREATE, the use of neighborhood col-35 lective communication is restricted to adjacency matrices, where the number of edges be-36 tween any two processes is defined to be the same for both processes (i.e., with a symmetric 37 adjacency matrix). In this case, the order of neighbors in the send and receive buffers is 38 defined as the sequence of neighbors as returned by MPI_GRAPH_NEIGHBORS. Note that 39 general graph topologies should generally be replaced by the distributed graph topologies.

40 For a Cartesian topology, created with MPI_CART_CREATE, the sequence of neigh- 41 bors in the send and receive buffers at each process is defined by order of the dimensions, 42first the neighbor in the negative direction and then in the positive direction with dis-43placement 1. The numbers of sources and destinations in the communication routines are 442^{*}ndims with ndims defined in MPI_CART_CREATE. If a neighbor does not exist, i.e., at 45the border of a Cartesian topology in the case of a non-periodic virtual grid dimension (i.e., 46periods[...]==false), then this neighbor is defined to be MPI_PROC_NULL.

47If a neighbor in any of the functions is MPI_PROC_NULL, then the neighborhood collec-48tive communication behaves like a point-to-point communication with MPI_PROC_NULL in

## **Unofficial Draft for Comment Only**

1

2

3

4

5

6 7

8 9

this direction. That is, the buffer is still part of the sequence of neighbors but it is neither communicated nor updated.

#### Neighborhood Gather 8.6.1

In this function, each process i gathers data items from each process j if an edge (j, i) exists in the topology graph, and each process i sends the same data items to all processes j where an edge (i, j) exists. The send buffer is sent to each neighboring process and the *l*-th block in the receive buffer is received from the l-th neighbor.

#### MPI_NEIGHBOR_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

	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	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)

#### 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)

34 Fortran 2008 binding MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, 35 recvtype, comm, ierror) 36 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf 37 INTEGER, INTENT(IN) :: sendcount, recvcount 38 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 39 TYPE(*), DIMENSION(...) :: recvbuf 40 41 TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4243 MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, 44 recvtype, comm, ierror) 45TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf 46INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount 47TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype

1 2

3 4

5

6

7

8

9 10 11

12

26

27

28

29 30

31

32

33

```
1
 TYPE(*), DIMENSION(...) :: recvbuf
\mathbf{2}
 TYPE(MPI_Comm), INTENT(IN) :: comm
3
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
 Fortran binding
5
 MPI_NEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
6
 RECVTYPE, COMM, IERROR)
7
 <type> SENDBUF(*), RECVBUF(*)
8
 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
9
10
 This function supports Cartesian communicators, graph communicators, and distributed
11
 graph communicators as described in Section 8.6. If comm is a distributed graph commu-
12
 nicator, the outcome is as if each process executed sends to each of its outgoing neighbors
13
 and receives from each of its incoming neighbors:
14
15
 MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
16
 int *srcs=(int*)malloc(indegree*sizeof(int));
17
 int *dsts=(int*)malloc(outdegree*sizeof(int));
18
 MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
19
 outdegree, dsts, MPI_UNWEIGHTED);
20
 int k;
21
 /* assume sendbuf and recvbuf are of type (char*) */
22
23
 for(k=0; k<outdegree; ++k)</pre>
^{24}
 MPI_Isend(sendbuf, sendcount, sendtype,dsts[k],...);
25
26
 for(k=0; k<indegree; ++k)</pre>
27
 MPI_Irecv(recvbuf+k*recvcount*extent(recvtype), recvcount, recvtype,
 srcs[k],...);
28
29
30
 MPI_Waitall(...);
^{31}
 Figure 8.1 shows the neighborhood gather communication of one process with outgoing
32
 neighbors d_0 \ldots d_3 and incoming neighbors s_0 \ldots s_5. The process will send its sendbuf to
33
 all four destinations (outgoing neighbors) and it will receive the contribution from all six
34
 sources (incoming neighbors) into separate locations of its receive buffer.
35
 All arguments are significant on all processes and the argument comm must have iden-
36
 tical values on all processes.
37
 The type signature associated with sendcount, sendtype, at a process must be equal to
38
 the type signature associated with recvcount, recvtype at all other processes. This implies
39
 that the amount of data sent must be equal to the amount of data received, pairwise between
40
 every pair of communicating processes. Distinct type maps between sender and receiver are
41
 still allowed.
42
43
 For optimization reasons, the same type signature is required indepen-
 Rationale.
44
 dently of whether the topology graph is connected or not. (End of rationale.)
45
46
 The "in place" option is not meaningful for this operation.
47
48
```

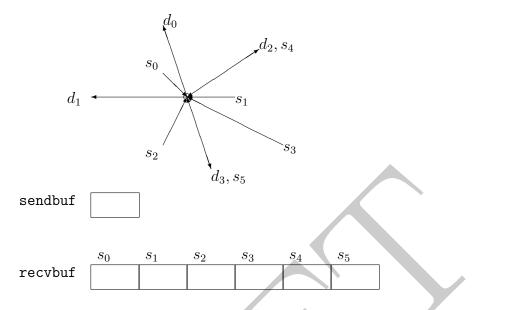


Figure 8.1: Neighborhood gather communication example

**Example 8.9** On a Cartesian virtual 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 8.2.

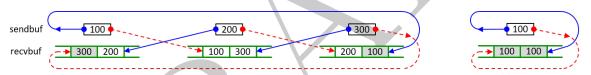


Figure 8.2: Cartesian neighborhood allgather example for 3 and 1 processes in a dimension

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 non-zero (in C) or .TRUE. (in Fortran).

The vector variant of MPI_NEIGHBOR_ALLGATHER allows one to gather different numbers of elements from each neighbor.

 31 

MPI_NEI	GHBOR_ALLGATHERV( recvtype, comm)	(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
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	non-negative integer array (of length indegree) containing the number of elements that are received from each neighbor
IN	displs	integer array (of length indegree). Entry i specifies the displacement (relative to <b>recvbuf</b> ) at which to place the incoming data from neighbor i
IN	recvtype	datatype of receive buffer elements (handle)
IN	comm	communicator with topology structure (handle)
	0	(const woid *gondbuf int gondcount
	MPI_Datatype s	<pre>endtype, void *recvbuf, const int recvcounts[], ls[], MPI_Datatype recvtype, MPI_Comm comm)</pre>
int MPI_	MPI_Datatype s const MPI_Coun	_c(const void *sendbuf, MPI_Count sendcount, endtype, void *recvbuf, t recvcounts[], const MPI_Aint displs[], ecvtype, MPI_Comm comm)
MPI_Neig TYPE INTE TYPE TYPE TYPE INTE	hbor_allgatherv(send displs, recvty (*), DIMENSION(), GER, INTENT(IN) :: ; (MPI_Datatype), INT (*), DIMENSION() (MPI_Comm), INTENT() GER, OPTIONAL, INTEN	sendcount, recvcounts(*), displs(*) ENT(IN) :: sendtype, recvtype :: recvbuf IN) :: comm NT(OUT) :: ierror
TYPE INTE TYPE TYPE INTE TYPE	displs, recvty (*), DIMENSION(), GER(KIND=MPI_COUNT_) (MPI_Datatype), INT (*), DIMENSION() GER(KIND=MPI_ADDRES (MPI_Comm), INTENT()	KIND), INTENT(IN) :: sendcount, recvcounts(*) ENT(IN) :: sendtype, recvtype :: recvbuf S_KIND), INTENT(IN) :: displs(*) IN) :: comm
	IN IN OUT IN OUT IN IN IN IN C bindin int MPI_ IN Fortran MPI_Neig TYPE INTE TYPE INTE TYPE INTE TYPE INTE TYPE INTE TYPE INTE	recvtype, comm) IN sendbuf IN sendcount IN sendtype OUT recvbuf IN recvcounts IN displs IN recvtype IN comm C binding int MPI_Neighbor_allgatherv MPI_Datatype s const int disp int MPI_Neighbor_allgatherv. MPI_Datatype s const MPI_Coun MPI_Datatype r Fortran 2008 binding MPI_Neighbor_allgatherv(sender) displs, recvty TYPE(*), DIMENSION(), INTEGER, INTENT(IN) :: : TYPE(MPI_Datatype), INTENT TYPE(*), DIMENSION() TYPE(*), DIMENSION(), INTEGER, OPTIONAL, INTENT MPI_Neighbor_allgatherv(sender) displs, recvty TYPE(*), DIMENSION() TYPE(*), DIMENSION()

```
1
Fortran binding
MPI_NEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
 \mathbf{2}
 3
 DISPLS, RECVTYPE, COMM, IERROR)
 4
 <type> SENDBUF(*), RECVBUF(*)
 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
 5
 6
 IERROR
 7
 This function supports Cartesian communicators, graph communicators, and distributed
 8
graph communicators as described in Section 8.6. If comm is a distributed graph commu-
 9
nicator, the outcome is as if each process executed sends to each of its outgoing neighbors
 10
and receives from each of its incoming neighbors:
 11
 12
MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
 13
int *srcs=(int*)malloc(indegree*sizeof(int));
 14
int *dsts=(int*)malloc(outdegree*sizeof(int));
 15
MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
 16
 outdegree, dsts, MPI_UNWEIGHTED);
 17
int k;
 18
 19
/* assume sendbuf and recvbuf are of type (char*) */
 20
for(k=0; k<outdegree; ++k)</pre>
 21
 MPI_Isend(sendbuf, sendcount, sendtype, dsts[k],...);
 22
 23
for(k=0; k<indegree; ++k)</pre>
 24
 MPI_Irecv(recvbuf+displs[k]*extent(recvtype), recvcounts[k], recvtype,
 25
 srcs[k],...);
 26
 27
MPI_Waitall(...);
 28
 29
 The type signature associated with sendcount, sendtype, at process j must be equal
 30
to the type signature associated with recvcounts[l], recvtype at any other process with
 31
srcs[l] = = j. This implies that the amount of data sent must be equal to the amount of
 32
data received, pairwise between every pair of communicating processes. Distinct type maps
 33
between sender and receiver are still allowed. The data received from the l-th neighbor is
 34
placed into recvbuf beginning at offset displs[l] elements (in terms of the recvtype).
 35
 The "in place" option is not meaningful for this operation.
 36
 All arguments are significant on all processes and the argument comm must have iden-
 37
```

```
tical values on all processes.
```

```
8.6.2 Neighbor Alltoall
```

In this function, each process i receives data items from each process j if an edge (j,i) exists in the topology graph or Cartesian topology. Similarly, each process i sends data items to all processes j where an edge (i, j) exists. This call is more general than MPI_NEIGHBOR_ALLGATHER in that different data items can be sent to each neighbor. The k-th block in send buffer is sent to the k-th neighboring process and the l-th block in the receive buffer is received from the l-th neighbor.

Unofficial Draft for Comment Only

38 39

40 41

42

43

44

45

```
1
 MPI_NEIGHBOR_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
\mathbf{2}
 comm)
3
 IN
 sendbuf
 starting address of send buffer (choice)
4
 IN
 sendcount
 number of elements sent to each neighbor
5
 (non-negative integer)
6
7
 IN
 sendtype
 datatype of send buffer elements (handle)
8
 OUT
 recvbuf
 starting address of receive buffer (choice)
9
 number of elements received from each neighbor
 IN
 recvcount
10
 (non-negative integer)
11
12
 IN
 recvtype
 datatype of receive buffer elements (handle)
13
 communicator with topology structure (handle)
 IN
 comm
14
15
 C binding
16
 int MPI_Neighbor_alltoall(const void *sendbuf, int sendcount,
17
 MPI_Datatype sendtype, void *recvbuf, int recvcount,
18
 MPI_Datatype recvtype, MPI_Comm comm)
19
20
 int MPI_Neighbor_alltoall_c(const void *sendbuf, MPI_Count sendcount,
21
 MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
22
 MPI_Datatype recvtype, MPI_Comm comm)
23
 Fortran 2008 binding
24
 MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
25
 recvtype, comm, ierror)
26
 TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
27
 INTEGER, INTENT(IN) :: sendcount, recvcount
28
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
29
 TYPE(*), DIMENSION(..) :: recvbuf
30
 TYPE(MPI_Comm), INTENT(IN) :: comm
31
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33
 MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
34
 recvtype, comm, ierror)
35
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
36
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
37
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
38
 TYPE(*), DIMENSION(..) :: recvbuf
39
 TYPE(MPI_Comm), INTENT(IN) :: comm
40
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
 Fortran binding
42
 MPI_NEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
43
 RECVTYPE, COMM, IERROR)
44
 <type> SENDBUF(*), RECVBUF(*)
45
 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
46
47
48
```

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 8.6. If **comm** is a distributed graph communicator, the outcome is as if each process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

```
MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
int *srcs=(int*)malloc(indegree*sizeof(int));
int *dsts=(int*)malloc(outdegree*sizeof(int));
MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
 outdegree, dsts, MPI_UNWEIGHTED);
int k;
/* assume sendbuf and recvbuf are of type (char*) */
for(k=0; k<outdegree; ++k)</pre>
 MPI_Isend(sendbuf+k*sendcount*extent(sendtype), sendcount, sendtype,
 dsts[k],...);
for(k=0; k<indegree; ++k)</pre>
 MPI_Irecv(recvbuf+k*recvcount*extent(recvtype), recvcount, recvtype,
 srcs[k],...);
MPI_Waitall(...);
 The type signature associated with sendcount, sendtype, at a process must be equal to
the type signature associated with recvcount, recvtype at any other process. This implies
that the amount of data sent must be equal to the amount of data received, pairwise between
every pair of communicating processes. Distinct type maps between sender and receiver are
still allowed.
 The "in place" option is not meaningful for this operation.
 All arguments are significant on all processes and the argument comm must have iden-
tical values on all processes.
Example 8.10 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 non-zero (in C) or .TRUE. (in Fortran).
 If each array element of sendbuf and recvbuf are described by sendcount, sendtype and
recvbuf, recvtype, then after MPI_NEIGHBOR_ALLTOALL on a Cartesian communicator re-
turned, the content of the recvbuf is as if the following code is executed:
MPI_Cartdim_get(comm, &ndims);
for(/*direction*/ d=0; d < ndims; d++) {</pre>
 MPI_Cart_shift(comm, /*direction*/ d, /*disp*/ 1, &rank_source, &rank_dest);
 MPI_Sendrecv(sendbuf[d*2+0],sendcount,sendtype,rank_source,/*sendtag*/d*2,
 recvbuf[d*2+1],recvcount,recvtype,rank_dest, /*recvtag*/ d*2,
 comm,&status);/*communication in direction of displacment -1*/
 MPI_Sendrecv(sendbuf[d*2+1],sendcount,sendtype,rank_dest, /*sendtag*/ d*2+1,
```

1

 $\mathbf{2}$ 

3

45

6

10

11 12

13

14

15

16 17

18

19

20 21

22 23

 24 

25

26

27

28

29

30

31 32

33

34

35

36

37

38

39

40

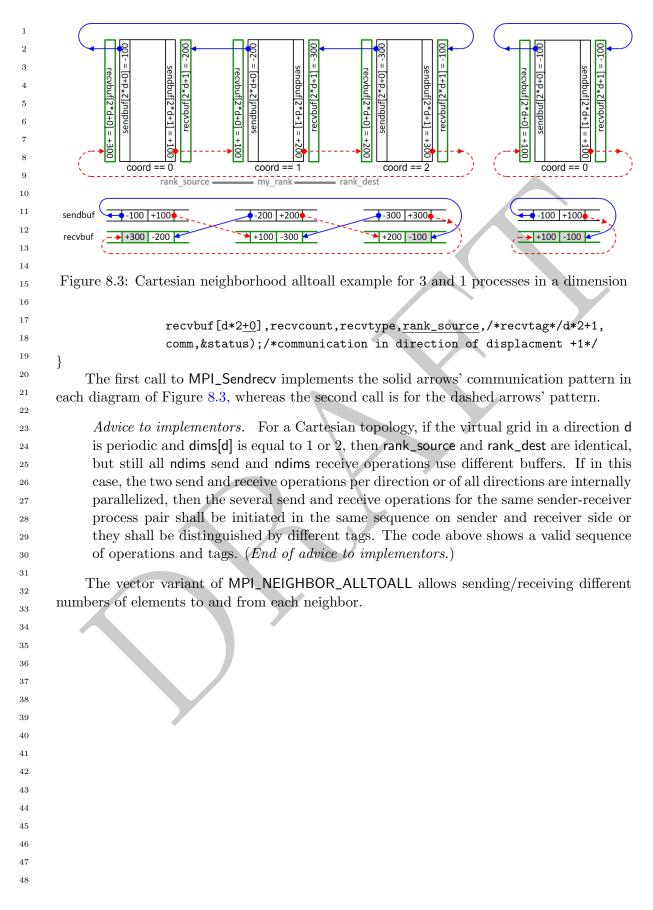
41 42

43 44

45

46

47



MPI_N	EIGHBOR_ALLTOALLV( rdispls, recvtype	sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, , comm)	1 2
IN	sendbuf	starting address of send buffer (choice)	3
IN	sendcounts	non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor	4 5 6 7
IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement (relative to sendbuf) from which to send the outgoing data to neighbor j	8 9 10
IN	sendtype	datatype of send buffer elements (handle)	11 12
OUT	recvbuf	starting address of receive buffer (choice)	12
IN	recvcounts	non-negative integer array (of length indegree) specifying the number of elements that are received from each neighbor	14 15 16
IN	rdispls	integer array (of length indegree). Entry i specifies the displacement (relative to <b>recvbuf</b> ) at which to place the incoming data from neighbor i	17 18 19 20
IN	recvtype	datatype of receive buffer elements (handle)	21
IN	comm	communicator with topology structure (handle)	22
			23
C bind	•		24 25
int MP	-	(const void *sendbuf, const int sendcounts[],	26
		<pre>spls[], MPI_Datatype sendtype, void *recvbuf, vcounts[], const int rdispls[],</pre>	27
		recvtype, MPI_Comm comm)	28
			29
int MP	Ŭ	_c(const void *sendbuf,	30
		<pre>nt sendcounts[], const MPI_Aint sdispls[], sendtype, void *recvbuf,</pre>	31
		<pre>sendtype, void *recvour, nt recvcounts[], const MPI_Aint rdispls[],</pre>	32 33
		recvtype, MPI_Comm comm)	34
-		<b>JI JI JJI JJJJJJJJJJJJJ</b>	35
	in 2008 binding		36
MP1_Ne	U	dbuf, sendcounts, sdispls, sendtype, recvbuf, displs, recvtype, comm, ierror)	37
ту		, INTENT(IN) :: sendbuf	38
		<pre>sendcounts(*), sdispls(*), recvcounts(*),</pre>	39
	rdispls(*)		40
TY	<pre>'PE(MPI_Datatype), IN'</pre>	TENT(IN) :: sendtype, recvtype	41
	<pre>PE(*), DIMENSION()</pre>		42 43
	PE(MPI_Comm), INTENT		43 44
IN	TEGER, OPTIONAL, INT	ENT(OUT) :: ierror	45
MPI_Ne	ighbor_alltoallv(sen	dbuf, sendcounts, sdispls, sendtype, recvbuf,	46
		displs, recvtype, comm, ierror)	47
ΤY	<pre>PE(*), DIMENSION()</pre>	, INTENT(IN) :: sendbuf	48

Unofficial Draft for Comment Only

```
1
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*),
\mathbf{2}
 recvcounts(*)
3
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
4
5
 TYPE(*), DIMENSION(..) :: recvbuf
6
 TYPE(MPI_Comm), INTENT(IN) :: comm
7
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
 Fortran binding
9
 MPI_NEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
10
 RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR)
11
 <type> SENDBUF(*), RECVBUF(*)
12
 INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
13
 RECVTYPE, COMM, IERROR
14
15
 This function supports Cartesian communicators, graph communicators, and distributed
16
 graph communicators as described in Section 8.6. If comm is a distributed graph commu-
17
 nicator, the outcome is as if each process executed sends to each of its outgoing neighbors
18
 and receives from each of its incoming neighbors:
19
 MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
20
21
 int *srcs=(int*)malloc(indegree*sizeof(int));
22
 int *dsts=(int*)malloc(outdegree*sizeof(int));
23
 MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
^{24}
 outdegree, dsts, MPI_UNWEIGHTED);
25
 int k;
26
 /* assume sendbuf and recvbuf are of type (char*) */
27
 for(k=0; k<outdegree; ++k)</pre>
28
 MPI_Isend(sendbuf+sdispls[k]*extent(sendtype), sendcounts[k], sendtype,
29
30
 dsts[k],...);
^{31}
 for(k=0; k<indegree; ++k)</pre>
32
 MPI_Irecv(recvbuf+rdispls[k]*extent(recvtype), recvcounts[k], recvtype,
33
34
 srcs[k],...);
35
 MPI_Waitall(...);
36
37
 The type signature associated with sendcounts[k], sendtype with dsts[k]==j at process
38
 i must be equal to the type signature associated with recvcounts[I], recvtype with srcs[I] = = i
39
 at process j. This implies that the amount of data sent must be equal to the amount of
40
 data received, pairwise between every pair of communicating processes. Distinct type maps
41
 between sender and receiver are still allowed. The data in the sendbuf beginning at offset
42
 sdispls[k] elements (in terms of the sendtype) is sent to the k-th outgoing neighbor. The data
43
 received from the I-th incoming neighbor is placed into recvbuf beginning at offset rdispls[I]
44
 elements (in terms of the recvtype).
45
 The "in place" option is not meaningful for this operation.
46
47
48
```

	rguments are significant es on all processes.	on all processes and the argument comm must have iden-	1 2	
	MPI_NEIGHBOR_ALLTOALLW allows one to send and receive with different datatypes			
to and from each neighbor.			4	
			5	
MPI NEI	GHBOR ALLTOALLW(se	endbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,	6 7	
	rdispls, recvtypes,		8	
IN	sendbuf	starting address of send buffer (choice)	9	
IN	sendcounts	non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor	10 11 12 13	
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)	14 15 16 17	
IN	sendtypes	array of datatypes (of length outdegree). Entry j specifies the type of data to send to neighbor j (array of handles)	18 19 20	
OUT	recvbuf	starting address of receive buffer (choice)	21 22	
IN	recvcounts	non-negative integer array (of length indegree) specifying the number of elements that are received from each neighbor	23 24 25	
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)	26 27 28 29	
IN	recvtypes	array of datatypes (of length indegree). Entry i specifies the type of data received from neighbor i (array of handles)	30 31 32 33	
IN	comm	communicator with topology structure (handle)	34	
C bindin int MPI	_Neighbor_alltoallw(c const MPI_Aint void *recvbuf,	<pre>const void *sendbuf, const int sendcounts[], sdispls[], const MPI_Datatype sendtypes[], const int recvcounts[],</pre>	35 36 37 38 39	
	const MPI_Aint MPI_Comm comm)	<pre>rdispls[], const MPI_Datatype recvtypes[],</pre>	40 41	
int MPI	_Neighbor_alltoallw_c	c(const void *sendbuf,	42 43	
		t sendcounts[], const MPI_Aint sdispls[],	43 44	
		type sendtypes[], void *recvbuf,	45	
		<pre>t recvcounts[], const MPI_Aint rdispls[], type recvtypes[], MPI_Comm comm)</pre>	46	
		-JE- Terrojpooli, in Teonum commu	47	

```
1
 Fortran 2008 binding
\mathbf{2}
 MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
3
 recvcounts, rdispls, recvtypes, comm, ierror)
4
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
5
 INTEGER, INTENT(IN) :: sendcounts(*), recvcounts(*)
6
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
7
 TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
8
 TYPE(*), DIMENSION(..) :: recvbuf
9
 TYPE(MPI_Comm), INTENT(IN) :: comm
10
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
 MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
12
 recvcounts, rdispls, recvtypes, comm, ierror)
13
 TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
14
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*),
15
 recvcounts(*)
16
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
17
 TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
18
 TYPE(*), DIMENSION(..) :: recvbuf
19
 TYPE(MPI_Comm), INTENT(IN) :: comm
20
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
 Fortran binding
23
 MPI_NEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
24
 RECVCOUNTS, RDISPLS, RECVTYPES, COMM, IERROR)
25
 <type> SENDBUF(*), RECVBUF(*)
26
 INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
27
 IERROR
28
 INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
29
 This function supports Cartesian communicators, graph communicators, and distributed
30
 graph communicators as described in Section 8.6. If comm is a distributed graph commu-
31
 nicator, the outcome is as if each process executed sends to each of its outgoing neighbors
32
 and receives from each of its incoming neighbors:
33
34
 MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
35
 int *srcs=(int*)malloc(indegree*sizeof(int));
36
 int *dsts=(int*)malloc(outdegree*sizeof(int));
37
 MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
38
 outdegree, dsts, MPI_UNWEIGHTED);
39
 int k;
40
41
 /* assume sendbuf and recvbuf are of type (char*) */
42
 for(k=0; k<outdegree; ++k)</pre>
43
 MPI_Isend(sendbuf+sdispls[k], sendcounts[k], sendtypes[k], dsts[k],...);
44
45
 for(k=0; k<indegree; ++k)</pre>
46
 MPI_Irecv(recvbuf+rdispls[k], recvcounts[k], recvtypes[k], srcs[k],...);
47
48
```

#### MPI_Waitall(...);

The type signature associated with sendcounts[k], sendtypes[k] with dsts[k]==j at process i must be equal to the type signature associated with recvcounts[l], recvtypes[l] with srcs[l]==i at process j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed.

The "in place" option is not meaningful for this operation.

All arguments are significant on all processes and the argument **comm** must have identical values on all processes.

# 8.7 Nonblocking Neighborhood Communication on Process Topologies

Nonblocking variants of the neighborhood collective operations allow relaxed synchronization and overlapping of computation and communication. The semantics are similar to nonblocking collective operations as described in Section 6.12.

#### 8.7.1 Nonblocking Neighborhood Gather

# MPI_INEIGHBOR_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)

	IN	sendbuf	starting address of send buffer (choice)	24
	IN	sendcount	number of elements sent to each neighbor	25
			(non-negative integer)	26
	IN	sendtype	datatype of send buffer elements (handle)	27 28
	OUT	recvbuf	starting address of receive buffer (choice)	29
	IN	recvcount	number of elements received from each neighbor	30
		reeveount	(non-negative integer)	31
			(non negative meeger)	32
	IN	recvtype	datatype of receive buffer elements (handle)	33
	IN	comm	communicator with topology structure (handle)	34
	ουτ	request	communication request (handle)	35
			1	36
~				37
С	binding			38
in	t MPI_Ir	neighbor_allgather(const	void *sendbuf, int sendcount,	39
		MPI_Datatype sendtype	e, void *recvbuf, int recvcount,	
		• • • • •	e, MPI_Comm comm, MPI_Request *request)	40
			,	41

## 

#### Fortran 2008 binding

MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request, ierror)  $\mathbf{2}$ 

 $13 \\ 14$ 

```
1
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
2
 INTEGER, INTENT(IN) :: sendcount, recvcount
3
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
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
 MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
9
 recvtype, comm, request, ierror)
10
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
11
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
12
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
13
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
14
 TYPE(MPI_Comm), INTENT(IN) :: comm
15
 TYPE(MPI_Request), INTENT(OUT) :: request
16
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18
 Fortran binding
19
 MPI_INEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
20
 RECVTYPE, COMM, REQUEST, IERROR)
21
 <type> SENDBUF(*), RECVBUF(*)
 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
22
23
 This call starts a nonblocking variant of MPI_NEIGHBOR_ALLGATHER.
24
25
26
 MPI_INEIGHBOR_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
27
 recvtype, comm, request)
28
 IN
 sendbuf
 starting address of send buffer (choice)
29
 IN
 sendcount
 number of elements sent to each neighbor
30
^{31}
 (non-negative integer)
32
 IN
 sendtype
 datatype of send buffer elements (handle)
33
 OUT
 recvbuf
 starting address of receive buffer (choice)
34
 IN
 recvcounts
 non-negative integer array (of length indegree)
35
 containing the number of elements that are received
36
 from each neighbor
37
38
 IN
 displs
 integer array (of length indegree). Entry i specifies
39
 the displacement (relative to recvbuf) at which to
40
 place the incoming data from neighbor i
41
 IN
 datatype of receive buffer elements (handle)
 recvtype
42
 IN
 communicator with topology structure (handle)
 comm
43
 OUT
 request
 communication request (handle)
44
45
46
 C binding
47
 int MPI_Ineighbor_allgatherv(const void *sendbuf, int sendcount,
48
 MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
```

1 const int displs[], MPI_Datatype recvtype, MPI_Comm comm, 2 MPI_Request *request) int MPI_Ineighbor_allgatherv_c(const void *sendbuf, MPI_Count sendcount, MPI_Datatype sendtype, void *recvbuf, 5 const MPI_Count recvcounts[], const MPI_Aint displs[], 6 MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) 7 Fortran 2008 binding 9 MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, 10 displs, recvtype, comm, request, ierror) 11 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf INTEGER, INTENT(IN) :: sendcount 1213 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf 1415INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*) 16 TYPE(MPI_Comm), INTENT(IN) :: comm 17TYPE(MPI_Request), INTENT(OUT) :: request 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 19 MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, 20displs, recvtype, comm, request, ierror) 21TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 22 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount 23TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 24TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 25INTEGER(KIND=MPI COUNT KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*) 26INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*) 27TYPE(MPI_Comm), INTENT(IN) :: comm 28TYPE(MPI_Request), INTENT(OUT) :: request 29 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 30 31Fortran binding 32 MPI_INEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, 33 DISPLS, RECVTYPE, COMM, REQUEST, IERROR) 34 <type> SENDBUF(*), RECVBUF(*) 35 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, 36 REQUEST, IERROR 37 This call starts a nonblocking variant of MPI_NEIGHBOR_ALLGATHERV. 38 39 40 41 4243 44 4546

```
428
 CHAPTER 8. PROCESS TOPOLOGIES
1
 8.7.2
 Nonblocking Neighborhood Alltoall
\mathbf{2}
3
4
 MPI_INEIGHBOR_ALLTOALL(sendbuf, sendcount, sendtype, recybuf, recycount, recytype,
5
 comm, request)
6
 IN
 sendbuf
 starting address of send buffer (choice)
7
8
 IN
 sendcount
 number of elements sent to each neighbor
 (non-negative integer)
9
10
 IN
 sendtype
 datatype of send buffer elements (handle)
11
 OUT
 recvbuf
 starting address of receive buffer (choice)
12
 IN
 number of elements received from each neighbor
13
 recvcount
14
 (non-negative integer)
15
 IN
 recvtype
 datatype of receive buffer elements (handle)
16
 IN
 communicator with topology structure (handle)
 comm
17
 OUT
 communication request (handle)
18
 request
19
20
 C binding
21
 int MPI_Ineighbor_alltoall(const void *sendbuf, int sendcount,
22
 MPI_Datatype sendtype, void *recvbuf, int recvcount,
23
 MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
24
 int MPI_Ineighbor_alltoall_c(const void *sendbuf, MPI_Count sendcount,
25
 MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
26
 MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
27
28
 Fortran 2008 binding
29
 MPI_Ineighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
30
 recvtype, comm, request, ierror)
31
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
32
 INTEGER, INTENT(IN) :: sendcount, recvcount
33
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
34
 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
35
 TYPE(MPI_Comm), INTENT(IN) :: comm
36
 TYPE(MPI_Request), INTENT(OUT) :: request
37
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
 MPI_Ineighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
39
 recvtype, comm, request, ierror)
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_Request), INTENT(OUT) :: request
46
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
```

	binding		1 2
MPI_INE.		DBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, M, REQUEST, IERROR)	3
<typ< td=""><td>pe&gt; SENDBUF(*), RECV</td><td></td><td>4</td></typ<>	pe> SENDBUF(*), RECV		4
INTE	EGER SENDCOUNT, SEND	TYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR	5
This	call starts a nonblocking	ng variant of MPI_NEIGHBOR_ALLTOALL.	6 7
			8 9
MPI_INE	IGHBOR_ALLTOALLV(s rdispls, recvtype,	comm, request)	9 10 11
IN	sendbuf	starting address of sond buffer (choice)	12
IN	sendcounts	specifying the number of elements to send to each	13 14 15
IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement (relative to sendbuf) from which send the outgoing data to neighbor j	16 17 18 19
IN	sendtype	datatype of send buffer elements (handle)	20
OUT	recvbuf	starting address of receive buffer (choice)	21
IN	recvcounts	non-negative integer array (of length indegree) specifying the number of elements that are received from each neighbor	22 23 24
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	25 26 27 28
IN	recvtype	datatype of receive buffer elements (handle)	29
IN	comm	communicator with topology structure (handle)	30
OUT	request	communication request (handle)	31 32
C bindi	ng		33 34
int MPI	_Ineighbor_alltoallv	(const void *sendbuf, const int sendcounts[],	35
	const int reco	vcounts[], const int rdispls[],	36 37 38
int MPT			39
	const MPI_Cour MPI_Datatype s const MPI_Cour	nt sendcounts[], const MPI_Aint sdispls[], sendtype, void *recvbuf, nt recvcounts[], const MPI_Aint rdispls[], recvtype, MPI_Comm comm, MPI_Request *request)	40 41 42 43
Fortran	2008 binding		44 45
MPI_Ine:	ighbor_alltoallv(sen recvcounts, ro	dbuf, sendcounts, sdispls, sendtype, recvbuf, displs, recvtype, comm, request, ierror)	46 47
TYPI	E(*), DIMENSION(),	INTENT(IN), ASYNCHRONOUS :: sendbuf	48

# Unofficial Draft for Comment Only

```
1
 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
2
 recvcounts(*), rdispls(*)
3
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
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
 MPI_Ineighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
9
 recvcounts, rdispls, recvtype, comm, request, ierror)
10
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
11
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
12
 sendcounts(*), recvcounts(*)
13
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
14
 rdispls(*)
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
 Fortran binding
22
 MPI_INEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
23
 RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)
24
 <type> SENDBUF(*), RECVBUF(*)
25
 INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
26
 RECVTYPE, COMM, REQUEST, IERROR
27
 This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALLV.
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

MPI_INEI	GHBOR_ALLTOALLW(sendbuf, rdispls, recvtypes, comm,	, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, request)	$\frac{1}{2}$	
IN	sendbuf	starting address of send buffer (choice)	$\frac{3}{4}$	
IN	sendcounts	non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor	4 5 6 7	
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)	8 9 10 11	
IN	sendtypes	array of datatypes (of length outdegree). Entry j specifies the type of data to send to neighbor j (array of handles)	12 13 14 15	
OUT	recvbuf	starting address of receive buffer (choice)	16	
IN	recvcounts	non-negative integer array (of length indegree) specifying the number of elements that are received from each neighbor	17 18 19	
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)	20 21 22 23 24	
IN	recvtypes	array of datatypes (of length indegree). Entry i specifies the type of data received from neighbor i (array of handles)	25 26 27	
IN	comm	communicator with topology structure (handle)	28	
OUT	request	communication request (handle)	29 30	
C binding	neighbor_alltoallw(const const MPI_Aint sdisp void *recvbuf, const	<pre>ls[], const MPI_Datatype recvtypes[],</pre>	31 32 33 34 35 36 37	
<pre>int MPI_Ineighbor_alltoallw_c(const void *sendbuf,</pre>				
<pre>Fortran 2008 binding MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,</pre>				

```
1
 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
\mathbf{2}
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
3
 rdispls(*)
4
 TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
5
 recvtypes(*)
6
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
7
 TYPE(MPI_Comm), INTENT(IN) :: comm
8
 TYPE(MPI_Request), INTENT(OUT) :: request
9
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
 MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
11
 recvcounts, rdispls, recvtypes, comm, request, ierror)
12
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
13
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
14
 sendcounts(*), recvcounts(*)
15
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
16
 rdispls(*)
17
 TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
18
 recvtypes(*)
19
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
20
 TYPE(MPI_Comm), INTENT(IN) :: comm
21
 TYPE(MPI_Request), INTENT(OUT) :: request
22
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
24
 Fortran binding
25
 MPI_INEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
26
 RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)
27
 <type> SENDBUF(*), RECVBUF(*)
28
 INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
29
 REQUEST, IERROR
30
 INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
31
 This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALLW.
32
33
34
 Persistent Neighborhood Communication on Process Topologies
 8.8
35
36
 Persistent variants of the neighborhood collective operations can offer significant perfor-
37
 mance benefits for programs with repetitive communication patterns. The semantics are
38
 similar to persistent collective operations as described in Section 6.13.
39
40
41
42
43
44
45
46
47
48
```

### 8.8.1 Persistent Neighborhood Gather

## MPI_NEIGHBOR_ALLGATHER_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, info, request)

		6
IN sendbuf	starting address of send buffer (choice)	7
IN sendcount	number of elements sent to each neighbor	8
	(non-negative integer)	9
IN sendtype	datatype of send buffer elements (handle)	10
OUT recvbuf	starting address of receive buffer (choice)	11 12
IN recvcount	number of elements received from each neighbor	13
	(non-negative integer)	14
IN recvtype	datatype of receive buffer elements (handle)	15
IN comm	communicator with topology structure (handle)	16 17
IN info	info argument (handle)	17
		19
OUT request	communication request (handle)	20
		21
C binding		22

## C binding

<pre>int MPI_Neighbor_allgather_init(const void *sendbuf, int sendcount,</pre>
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
MPI_Request *request)
<pre>int MPI_Neighbor_allgather_init_c(const void *sendbuf, MPI_Count sendcount,</pre>
MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
MPI_Request *request)

### Fortran 2008 binding

MPI_Neighbor_allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, info, 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_Info), INTENT(IN) :: info TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Neighbor_allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, info, request, ierror) TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype

TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf

5

6 7 8

30 31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

4647

1		E(MPI_Comm), INT				
2	TYPE(MPI_Info), INTENT(IN) :: info					
3	TYPE(MPI_Request), INTENT(OUT) :: request					
4	INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
5						
6	Fortran binding					
7	MPI_NEIGHBOR_ALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,					
8	RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR)					
9	<type> SENDBUF(*), RECVBUF(*)</type>					
10	INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,					
11		IERROR				
12	Crea	tes a persistent co	llective communication request for the neighborhood allgather			
13	operation					
14	1					
15						
16	MPI_NEI		IERV_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcounts,			
17		displs, recvt	ype, comm, info, request)			
18	IN	sendbuf	starting address of send buffer (choice)			
19	IN	sendcount	number of elements sent to each neighbor			
20	IIN	Senacount	(non-negative integer)			
21						
22	IN	sendtype	datatype of send buffer elements (handle)			
23	OUT	recvbuf	starting address of receive buffer (choice)			
24	IN	recvcounts	non-negative integer array (of length indegree)			
25 26			containing the number of elements that are received			
20			from each neighbor			
28	IN	displs	integer array (of length indegree). Entry i specifies			
29			the displacement (relative to recvbuf) at which to			
30			place the incoming data from neighbor i			
31	IN	recvtype	datatype of receive buffer elements (handle)			
32	IN	comm	communicator with topology structure (handle)			
33 34	IN	info	info argument (handle)			
35						
36	OUT	request	communication request (handle)			
37						
38	C bindi	U				
39	int MPI_	0	herv_init(const void *sendbuf, int sendcount,			
40		MPI_Dataty	<pre>vpe sendtype, void *recvbuf, const int recvcounts[],</pre>			
40			<pre>displs[], MPI_Datatype recvtype, MPI_Comm comm,</pre>			
42		MPI_Info i	nfo, MPI_Request *request)			
42	int MPT	Neighbor allgat	herv_init_c(const void *sendbuf,			
44			sendcount, MPI_Datatype sendtype, void *recvbuf,			
45			_Count recvcounts[], const MPI_Aint displs[],			
46			vpe recvtype, MPI_Comm comm, MPI_Info info,			
47		-	st *request)			
48		··· +_•···quor				

```
Fortran 2008 binding
 1
 2
MPI_Neighbor_allgatherv_init(sendbuf, sendcount, sendtype, recvbuf,
 3
 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
 10
 TYPE(MPI_Info), INTENT(IN) :: info
 TYPE(MPI_Request), INTENT(OUT) :: request
 11
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 12
 13
MPI_Neighbor_allgatherv_init(sendbuf, sendcount, sendtype, recvbuf,
 14
 recvcounts, displs, recvtype, comm, info, request, ierror)
 15
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
 16
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
 17
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 18
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 19
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
 20
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
 21
 TYPE(MPI_Comm), INTENT(IN) :: comm
 22
 TYPE(MPI_Info), INTENT(IN) :: info
 23
 TYPE(MPI_Request), INTENT(OUT) :: request
 ^{24}
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 25
 26
Fortran binding
 27
MPI_NEIGHBOR_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
 28
 RECVCOUNTS, DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)
 29
 <type> SENDBUF(*), RECVBUF(*)
 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
 30
 31
 INFO, REQUEST, IERROR
 32
 Creates a persistent collective communication request for the neighborhood allgathery
 33
operation.
 34
 35
 36
 37
 38
 39
 40
 41
 42
 43
 44
 45
 46
```

	436		CHAPTER 8. PROCESS TOPOLOGIES			
1 2 3	8.8.2 Persistent Neighborhood Alltoall					
4 5	MPI_NEIGHBOR_ALLTOALL_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, info, request)					
6 7	IN sendbuf starting address of send buffer (choice)					
8 9	IN	sendcount	number of elements sent to each neighbor (non-negative integer)			
10	IN	sendtype	datatype of send buffer elements (handle)			
11 12	OUT	recvbuf	starting address of receive buffer (choice)			
13 14	IN	recvcount	number of elements received from each neighbor (non-negative integer)			
15	IN	recvtype	datatype of receive buffer elements (handle)			
16 17	IN	comm	communicator with topology structure (handle)			
18	IN	info	info argument (handle)			
19 20	OUT	request	communication request (handle)			
22 23 24 25 26 27 28 29 30 31	int MPI_N	<pre>Meighbor_alltoall_init(con MPI_Datatype sendtyp MPI_Datatype recvtyp MPI_Request *request Weighbor_alltoall_init_c( MPI_Datatype sendtyp MPI_Datatype recvtyp MPI_Request *request</pre>	const void *sendbuf, MPI_Count sendcount, e, void *recvbuf, MPI_Count recvcount, e, MPI_Comm comm, MPI_Info info,			
32 33 34 35 36 37 38 39 40 41 42	<pre>Fortran 2008 binding MPI_Neighbor_alltoall_init(sendbuf, sendcount, sendtype, recvbuf,</pre>					
43 44 45 46 47 48	<pre>MPI_Neighbor_alltoall_init(sendbuf, sendcount, sendtype, recvbuf,</pre>					

<pre>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_NEIGHBOR_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR Creates a persistent collective communication request for the neighborhood alltoall</type></pre>					
operat	ion.		13 14		
MPI_N		/_INIT(sendbuf, sendcounts, sdispls, sendtype, recvbuf, ispls, recvtype, comm, info, request)	15 16 17		
IN	sendbuf	starting address of send buffer (choice)	18 19		
IN	sendcounts	non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor	19 20 21 22		
IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement (relative to sendbuf) from which send the outgoing data to neighbor j	23 24 25		
IN	sendtype	datatype of send buffer elements (handle)	26 27		
OUT	recvbuf	starting address of receive buffer (choice)	27		
IN	recvcounts	non-negative integer array (of length indegree) specifying the number of elements that are received from each neighbor	29 30 31		
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	32 33 34		
IN	recvtype	datatype of receive buffer elements (handle)	35 36		
IN	comm	communicator with topology structure (handle)	37		
IN	info	info argument (handle)	38		
OUT	request	communication request (handle)	39 40		
			41		
C bin		lv_init(const void *sendbuf,	42		
THE M	0	endcounts[], const int sdispls[],	43 44		
		e sendtype, void *recvbuf, const int recvcounts[],	45		
		displs[], MPI_Datatype recvtype, MPI_Comm comm,	46		
	MPI_Info info, MPI_Request *request)				

```
1
 int MPI_Neighbor_alltoallv_init_c(const void *sendbuf,
\mathbf{2}
 const MPI_Count sendcounts[], const MPI_Aint sdispls[],
3
 MPI_Datatype sendtype, void *recvbuf,
4
 const MPI_Count recvcounts[], const MPI_Aint rdispls[],
5
 MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
6
 MPI_Request *request)
7
 Fortran 2008 binding
8
 MPI_Neighbor_alltoallv_init(sendbuf, sendcounts, sdispls, sendtype,
9
 recvbuf, recvcounts, rdispls, recvtype, comm, info, request,
10
 ierror)
11
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
12
 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
13
 recvcounts(*), rdispls(*)
14
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
15
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
16
 TYPE(MPI_Comm), INTENT(IN) :: comm
17
 TYPE(MPI_Info), INTENT(IN) :: info
18
 TYPE(MPI_Request), INTENT(OUT) :: request
19
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
 MPI_Neighbor_alltoallv_init(sendbuf, sendcounts, sdispls, sendtype,
22
 recvbuf, recvcounts, rdispls, recvtype, comm, info, request,
23
 ierror)
24
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
25
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
26
 sendcounts(*), recvcounts(*)
27
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
28
 rdispls(*)
29
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
30
 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
31
 TYPE(MPI_Comm), INTENT(IN) :: comm
32
 TYPE(MPI_Info), INTENT(IN) :: info
33
 TYPE(MPI_Request), INTENT(OUT) :: request
34
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
 Fortran binding
36
 MPI_NEIGHBOR_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE,
37
 RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST,
38
 IERROR)
39
 <type> SENDBUF(*), RECVBUF(*)
40
 INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
41
 RECVTYPE, COMM, INFO, REQUEST, IERROR
42
43
 Creates a persistent collective communication request for the neighborhood alloally
44
 operation.
45
46
47
48
```

MPI_NEI		(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, ecvtypes, comm, info, request)	1 $2$
IN	sendbuf	starting address of send buffer (choice)	3
IN	sendcounts	non-negative integer array (of length outdegree)	4 5
		specifying the number of elements to send to each	6
		neighbor	7
IN	sdispls	integer array (of length outdegree). Entry <b>j</b> specifies	8
		the displacement in bytes (relative to $sendbuf)$ from	9
		which to take the outgoing data destined for	10 11
		neighbor j (array of integers)	12
IN	sendtypes	array of datatypes (of length outdegree). Entry j	13
		specifies the type of data to send to neighbor j (array of handles)	14
0.UT			15
OUT	recvbuf	starting address of receive buffer (choice)	16
IN	recvcounts	non-negative integer array (of length indegree)	17 18
		specifying the number of elements that are received from each neighbor	19
	Paula		20
IN	rdispls	integer array (of length indegree). Entry i specifies the displacement in bytes (relative to recvbuf) at	21
		which to place the incoming data from neighbor i	22
	(array of integers)	23	
IN	recvtypes	array of datatypes (of length indegree). Entry i	24 25
		specifies the type of data received from neighbor i	26
		(array of handles)	27
IN	comm	communicator with topology structure (handle)	28
IN	info	info argument (handle)	29 30
OUT	request	communication request (handle)	31
			32
C bindin	ıg		33
int MPI_	Neighbor_alltoallw_init		34
		<pre>nts[], const MPI_Aint sdispls[],</pre>	35
		e sendtypes[], void *recvbuf, nts[], const MPI_Aint rdispls[],	$\frac{36}{37}$
		e recvtypes[], MPI_Comm comm, MPI_Info info,	38
	MPI_Request *requ		39
int MPT	Neighbor alltoallw init	t_c(const void *sendbuf,	40
1110 III I_	-	endcounts[], const MPI_Aint sdispls[],	41
		e sendtypes[], void *recvbuf,	42
		<pre>ecvcounts[], const MPI_Aint rdispls[],</pre>	$\frac{43}{44}$
	• •	e recvtypes[], MPI_Comm comm, MPI_Info info,	45
	MPI_Request *requ	est)	46
Fortran	2008 binding		47
MPI_Neig	hbor_alltoallw_init(se	ndbuf, sendcounts, sdispls, sendtypes,	48

```
1
 recvbuf, recvcounts, rdispls, recvtypes, comm, info, request,
2
 ierror)
3
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
4
 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
5
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
6
 rdispls(*)
7
 TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
8
 recvtypes(*)
9
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
10
 TYPE(MPI_Comm), INTENT(IN) :: comm
11
 TYPE(MPI_Info), INTENT(IN) :: info
12
 TYPE(MPI_Request), INTENT(OUT) :: request
13
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
 MPI_Neighbor_alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes,
15
 recvbuf, recvcounts, rdispls, recvtypes, comm, info, request,
16
 ierror)
17
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
18
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
19
 sendcounts(*), recvcounts(*)
20
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
21
 rdispls(*)
22
 TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
23
 recvtypes(*)
24
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
25
 TYPE(MPI_Comm), INTENT(IN) :: comm
26
 TYPE(MPI_Info), INTENT(IN) :: info
27
 TYPE(MPI_Request), INTENT(OUT) :: request
28
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
 Fortran binding
31
 MPI_NEIGHBOR_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES,
32
 RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST,
33
 IERROR)
34
 <type> SENDBUF(*), RECVBUF(*)
35
 INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
36
 INFO, REQUEST, IERROR
37
 INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
38
 Creates a persistent collective communication request for the neighborhood alltoallw
39
 operation.
40
```

43 44

## 8.9 An Application Example

Example 8.11 The example in Figures 8.4-8.7 shows how the grid definition and inquiry
 functions can be used in an application program. A partial differential equation, for instance
 the Poisson equation, is to be solved on a rectangular domain. First, the processes organize
 themselves in a two-dimensional structure. Each process then inquires about the ranks of

its neighbors in the four directions (up, down, right, left). The numerical problem is solved by an iterative method, the details of which are hidden in the subroutine **relax**.

In each relaxation step each process computes new values for the solution grid function at the points u(1:100,1:100) owned by the process. Then the values at inter-process boundaries have to be exchanged with neighboring processes. For example, the newly calculated values in u(1,1:100) must be sent into the halo cells u(101,1:100) of the left-hand neighbor with coordinates (own_coord(1)-1,own_coord(2)).

 $\mathbf{5}$ 

```
INTEGER ndims, num_neigh
1
 LOGICAL reorder
2
 PARAMETER (ndims=2, num_neigh=4, reorder=.true.)
3
 INTEGER comm, comm_size, comm_cart, dims(ndims), ierr
4
 INTEGER neigh_rank(num_neigh), own_coords(ndims), i, j, it
5
 LOGICAL periods(ndims)
6
 REAL u(0:101,0:101), f(0:101,0:101)
7
 DATA dims / ndims * 0 /
8
 comm = MPI_COMM_WORLD
9
 CALL MPI_COMM_SIZE(comm, comm_size, ierr)
10
 Set process grid size and periodicity
 !
11
 CALL MPI_DIMS_CREATE(comm_size, ndims, dims, ierr)
12
 periods(1) = .TRUE.
13
 periods(2) = .TRUE.
14
 Create a grid structure in WORLD group and inquire about own position
15
 CALL MPI_CART_CREATE(comm, ndims, dims, periods, reorder, &
16
 comm_cart, ierr)
17
 CALL MPI_CART_GET(comm_cart, ndims, dims, periods, own_coords, ierr)
18
 i = own_coords(1)
19
 j = own_coords(2)
20
 ! Look up the ranks for the neighbors. Own process coordinates are (i,j).
21
 ! Neighbors are (i-1,j), (i+1,j), (i,j-1), (i,j+1) modulo (dims(1),dims(2))
22
 CALL MPI_CART_SHIFT(comm_cart, 0,1, neigh_rank(1), neigh_rank(2), ierr)
23
 CALL MPI_CART_SHIFT(comm_cart, 1,1, neigh_rank(3), neigh_rank(4), ierr)
24
 ! Initialize the grid functions and start the iteration
25
 CALL init(u, f)
26
 DO it=1,100
27
 CALL relax(u, f)
28
 Exchange data with neighbor processes
 !
29
 CALL exchange(u, comm_cart, neigh_rank, num_neigh)
30
 END DO
31
 CALL output(u)
32
33
34
 Figure 8.4: Set-up of process structure for two-dimensional parallel Poisson solver
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

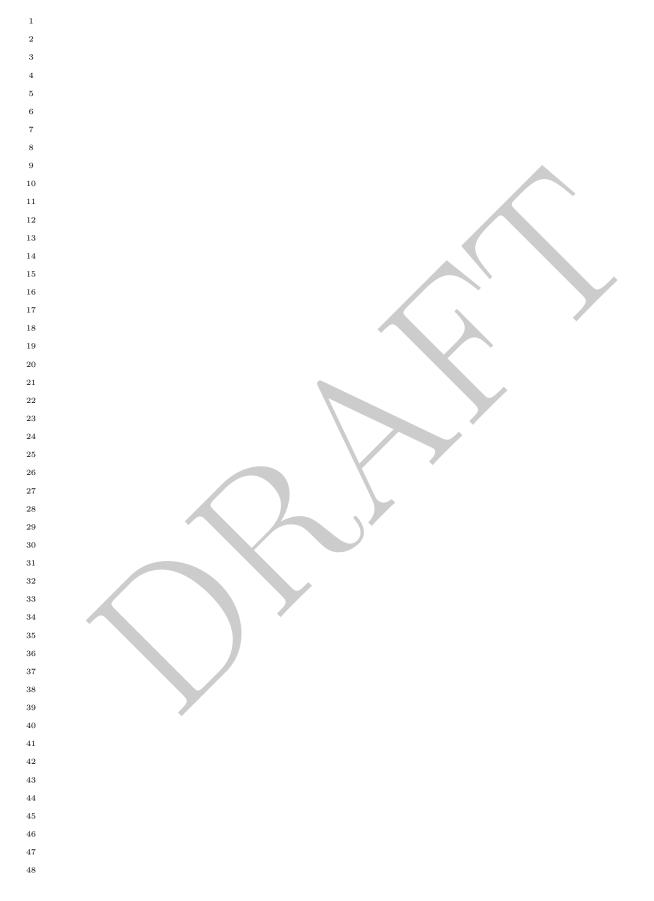
```
SUBROUTINE exchange(u, comm_cart, neigh_rank, num_neigh)
 1
REAL u(0:101,0:101)
 \mathbf{2}
INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
 3
REAL sndbuf(100,num_neigh), rcvbuf(100,num_neigh)
 4
INTEGER ierr
 5
sndbuf(1:100,1) = u(1,1:100)
 6
sndbuf(1:100,2) = u(100,1:100)
sndbuf(1:100,3) = u(1:100, 1)
sndbuf(1:100,4) = u(1:100,100)
CALL MPI_NEIGHBOR_ALLTOALL(sndbuf, 100, MPI_REAL, rcvbuf, 100, MPI_REAL, &
 10
 comm_cart, ierr)
 11
! instead of
 12
! CALL MPI_IRECV(rcvbuf(1,1),100,MPI_REAL, neigh_rank(1),..., rq(1), ierr)
 13
! CALL MPI_ISEND(sndbuf(1,2),100,MPI_REAL, neigh_rank(2),..., rq(2), ierr)
 14
!
 Always pairing a receive from rank_source with a send to rank_dest
 15
 of the same direction in MPI_CART_SHIFT!
!
 16
! CALL MPI_IRECV(rcvbuf(1,2),100,MPI_REAL, neigh_rank(2),..., rq(3), ierr)
 17
! CALL MPI_ISEND(sndbuf(1,1),100,MPI_REAL, neigh_rank(1),..., rq(4), ierr)
 18
! CALL MPI_IRECV(rcvbuf(1,3),100,MPI_REAL, neigh_rank(3),..., rq(5), ierr)
 19
! CALL MPI_ISEND(sndbuf(1,4),100,MPI_REAL, neigh_rank(4),..., rq(6), ierr)
 20
! CALL MPI_IRECV(rcvbuf(1,4),100,MPI_REAL, neigh_rank(4),..., rq(7), ierr)
 21
! CALL MPI_ISEND(sndbuf(1,3),100,MPI_REAL, neigh_rank(3),..., rq(8), ierr)
 22
1
 Of course, one can first start all four IRECV and then all four ISEND,
 23
I.
 Or vice versa, but both in the sequence shown above. Otherwise, the
 ^{24}
 matching would be wrong for 2 or only 1 processes in a direction.
!
 25
! CALL MPI_WAITALL(2*num_neigh, rq, statuses, ierr)
 26
u(0,1:100) = rcvbuf(1:100,1)
 27
u(101,1:100) = rcvbuf(1:100,2)
 28
u(1:100, 0) = rcvbuf(1:100,3)
 29
u(1:100,101) = rcvbuf(1:100,4)
 30
END
 31
 32
 33
Figure 8.5: Communication routine with local data copying and sparse neighborhood all-
 34
to-all
 35
 36
 37
 38
 39
 40
 41
 42
 43
 44
 45
 46
 47
```

```
SUBROUTINE exchange(u, comm_cart, neigh_rank, num_neigh)
1
 IMPLICIT NONE
2
 USE MPI
3
 REAL u(0:101,0:101)
4
 INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
5
 INTEGER sndcounts(num_neigh), sndtypes(num_neigh)
6
 INTEGER rcvcounts(num_neigh), rcvtypes(num_neigh)
7
 INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal
 INTEGER (KIND=MPI_ADDRESS_KIND) sdispls(num_neigh), rdispls(num_neigh)
8
 INTEGER type_vec, ierr
9
 ! The following initialization need to be done only once
10
 ! before the first call of exchange.
11
 CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lb, sizeofreal, ierr)
12
 CALL MPI_TYPE_VECTOR(100, 1, 102, MPI_REAL, type_vec, ierr)
13
 CALL MPI_TYPE_COMMIT(type_vec, ierr)
14
 sndtypes(1:2) = type_vec
 sndcounts(1:2) = 1
15
 sndtypes(3:4) = MPI_REAL
16
 sndcounts(3:4) = 100
17
 rcvtypes = sndtypes
18
 rcvcounts = sndcounts
19
 sdispls(1) = (1 + 1*102) * sizeofreal ! first element of u(1
 1:100)
20
 sdispls(2) = (100 +
 1*102) * sizeofreal ! first element of u(100
 1:100
21
 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
22
)
 rdispls(1) = (0 + 1*102) * sizeofreal ! first element of u(0
 , 1:100)
23
 1*102) * sizeofreal ! first element of u(101
 rdispls(2) = (101 +
 1:100)
24
 rdispls(3) = (1 +
 0*102) * sizeofreal ! first element of u(1:100, 0
)
25
 rdispls(4) = (1 + 101*102) * sizeofreal ! first element of u(1:100,101
)
26
 ! the following communication has to be done in each call of exchange
27
 CALL MPI_NEIGHBOR_ALLTOALLW(u, sndcounts, sdispls, sndtypes, &
28
 u, rcvcounts, rdispls, rcvtypes, &
29
 comm_cart, ierr)
 ! The following finalizing need to be done only once
30
 ! after the last call of exchange.
31
 CALL MPI_TYPE_FREE(type_vec, ierr)
32
 END
33
34
35
 Figure 8.6: Communication routine with sparse neighborhood all-to-all-w and without local
36
 data copying
37
38
39
40
41
42
43
44
45
46
47
48
```

```
INTEGER ndims, num_neigh
 1
LOGICAL reorder
 2
PARAMETER (ndims=2, num_neigh=4, reorder=.true.)
INTEGER comm, comm_size, comm_cart, dims(ndims), it, ierr
LOGICAL periods(ndims)
 5
REAL u(0:101,0:101), f(0:101,0:101)
 6
DATA dims / ndims * 0 /
INTEGER sndcounts(num_neigh), sndtypes(num_neigh)
INTEGER rcvcounts(num_neigh), rcvtypes(num_neigh)
INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal
 10
INTEGER (KIND=MPI_ADDRESS_KIND) sdispls(num_neigh), rdispls(num_neigh)
 11
INTEGER type_vec, request, status
 12
comm = MPI_COMM_WORLD
 13
CALL MPI_COMM_SIZE(comm, comm_size, ierr)
 14
 Set process grid size and periodicity
 15
CALL MPI_DIMS_CREATE(comm_size, ndims, dims, ierr)
 16
periods(1) = .TRUE.
 17
periods(2) = .TRUE.
 18
 Create a grid structure in WORLD group
!
 19
CALL MPI_CART_CREATE(comm, ndims, dims, periods, reorder, &
 20
 comm_cart, ierr)
 21
! Create datatypes for the neighborhood communication
 22
i
 23
! Insert code from example in Figure 7.4 to create and initialize
 24
! sndcounts, sdispls, sndtypes, rcvcounts, rdispls, and rcvtypes
 25
Ţ
 26
! Initialize the neighborhood all-to-all-w operation
 27
CALL MPI_NEIGHBOR_ALLTOALLW_INIT(u, sndcounts, sdispls, sndtypes, &
 28
 u, rcvcounts, rdispls, rcvtypes, &
 29
 comm_cart, info, request, ierr)
 30
! Initialize the grid functions and start the iteration
 31
CALL init(u, f)
 32
DO it=1,100
 33
 Start data exchange with neighbor processes
1
 34
 CALL MPI_START(request, ierr)
 35
 Compute inner cells
ļ
 36
 CALL relax_inner (u, f)
 37
 Check on completion of neighbor exchange
!
 38
 CALL MPI_WAIT(request, status, ierr)
 39
!
 Compute edge cells
 40
 CALL relax_edges(u, f)
 41
END DO
 42
CALL output(u)
 43
CALL MPI_REQUEST_FREE(request, ierr)
 44
CALL MPI_TYPE_FREE(type_vec, ierr)
 45
 46
```

Figure 8.7: Two-dimensional parallel Poisson solver with persistent sparse neighborhood all-to-all-w and without local data copying

47



## Chapter 9

# **MPI** Environmental Management

1 2 3

15

16

17

1819

2021

22

23

 24 25

26

2728

29

30

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.

#### Implementation Information 9.1

## 9.1.1 Version Inquiries

In order to cope with changes to the MPI Standard, there are both compile-time and runtime ways to determine which version of the standard is in use in the environment one is using.

The "version" will be represented by two separate integers, for the version and subversion: In C,

```
#define MPI_VERSION
 3
#define MPI_SUBVERSION 1
```

in Fortran.				
III FOITIAII,				
INTEGER :: MPI_VERSION, MPI_SUBVERSION				
PARAMETER $(MPI_VERSION = 3)$				
PARAMETER (MPI_SUBVERSION	= 1)	35		
		36		
For runtime determination,		37		
		38		
MPI_GET_VERSION(version, subversion)				
Υ.	,	40		
OUT version	version number (integer)	41		
OUT subversion	subversion number (integer)	42		
		43		
C binding		44		
int MPI_Get_version(int *version)	on. int *subversion)	45		
		46		
Fortran 2008 binding		47		
MPI_Get_version(version, subver	rsion, ierror)	48		

```
1
 INTEGER, INTENT(OUT) :: version, subversion
\mathbf{2}
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
 Fortran binding
4
 MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)
5
 INTEGER VERSION, SUBVERSION, IERROR
6
7
 MPI_GET_VERSION can be called at any time in an MPI program. This function must
8
 always be thread-safe, as defined in Section 11.6. Valid (MPI_VERSION, MPI_SUBVERSION)
9
 pairs in this and previous versions of the MPI standard are (4,0), (3,1), (3,0), (2,2), (2,1),
10
 (2,0), and (1,2).
11
12
 MPI_GET_LIBRARY_VERSION(version, resultlen)
13
14
 OUT
 version
 version number (string)
15
 OUT
 Length (in printable characters) of the result
 resultlen
16
 returned in version (integer)
17
18
 C binding
19
 int MPI_Get_library_version(char *version, int *resultlen)
20
21
 Fortran 2008 binding
22
 MPI_Get_library_version(version, resultlen, ierror)
23
 CHARACTER(LEN=MPI_MAX_LIBRARY_VERSION_STRING), INTENT(OUT) :: version
24
 INTEGER, INTENT(OUT) :: resultlen
25
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
 Fortran binding
27
 MPI_GET_LIBRARY_VERSION(VERSION, RESULTLEN, IERROR)
28
 CHARACTER*(*) VERSION
29
 INTEGER RESULTLEN, IERROR
30
^{31}
 This routine returns a string representing the version of the MPI library. The version
32
 argument is a character string for maximum flexibility.
33
34
 Advice to implementors. An implementation of MPI should return a different string
35
 for every change to its source code or build that could be visible to the user. (End of
36
 advice to implementors.)
37
38
 The argument version must represent storage that is
39
 MPI_MAX_LIBRARY_VERSION_STRING characters long. MPI_GET_LIBRARY_VERSION may
40
 write up to this many characters into version.
41
 The number of characters actually written is returned in the output argument, resultlen.
42
 In C, a null character is additionally stored at version[resultlen]. The value of resultlen cannot
43
 be larger than MPI_MAX_LIBRARY_VERSION_STRING - 1. In Fortran, version is padded on
44
 the right with blank characters. The value of resultlen cannot be larger than
45
 MPI_MAX_LIBRARY_VERSION_STRING.
46
 MPI_GET_LIBRARY_VERSION can be called at any time in an MPI program. This
47
 function must always be thread-safe, as defined in Section 11.6.
```

CHAPTER 9. MPI ENVIRONMENTAL MANAGEMENT

448

9.1.2 Environmental Inquiries	1
When using the World Model (Section 11.2), a set of attributes that describe the execution environment is attached to the communicator MPI_COMM_WORLD when MPI is initialized.	2 3
The values of these attributes can be inquired by using the function	4
MPI_COMM_GET_ATTR described in Section 7.7 and in Section 19.3.7. It is erroneous to	5
delete these attributes, free their keys, or change their values.	6
The list of predefined attribute keys include	7 8
MPI_TAG_UB Upper bound for tag value.	9 10
<b>MPI_HOST</b> Host process rank, if such exists, MPI_PROC_NULL, otherwise.	10
<b>MPI_IO</b> rank of a node that has regular I/O facilities (possibly myrank). Nodes in the same communicator may return different values for this parameter.	12 13 14
<b>MPI_WTIME_IS_GLOBAL</b> Boolean variable that indicates whether clocks are synchronized.	15 16
When using the Sessions Model (Section 11.3), only the $MPI_TAG_UB$ attribute is available.	17 18
Vendors may add implementation-specific parameters (such as node number, real mem-	19
ory size, virtual memory size, etc.)	20
These predefined attributes do not change value between MPI initialization (MPI_INIT)	21
and MPI completion (MPI_FINALIZE), and cannot be updated or deleted by users.	22
	23
Advice to users. Note that in the C binding, the value returned by these attributes	24
is a <i>pointer</i> to an <i>int</i> containing the requested value. (End of advice to users.)	25
The required parameter values are discussed in more detail below:	26
The required parameter values are discussed in more detail below.	27
Tag Values	28 29
	29 30
Tag values range from 0 to the value returned for MPI_TAG_UB, inclusive. These values are	31
guaranteed to be unchanging during the execution of an MPI program. In addition, the tag	32
upper bound value must be at least 32767. An MPI implementation is free to make the	33
value of MPI_TAG_UB larger than this; for example, the value $2^{30} - 1$ is also a valid value for MPI_TAG_UB	34
for MPI_TAG_UB. In the Sessions Model, the attribute MPI_TAG_UB is attached to all communicators	35
created by MPI_COMM_CREATE_FROM_GROUP and	36
MPI_INTERCOMM_CREATE_FROM_GROUPS, with the same value on all MPI processes	37
in the communicator. In the World Model, the attribute MPI_TAG_UB has the same value	38
on all processes of MPI_COMM_WORLD.	39
	40
Host Rank	41
	42
The value returned for MPI_HOST gets the rank of the <i>HOST</i> process in the group associated	43
with communicator MPI_COMM_WORLD, if there is such. MPI_PROC_NULL is returned if	44
there is no host. MPI does not specify what it means for a process to be a $HOST$ , nor does it means that a $HOST$ satisfy	45
it requires that a <i>HOST</i> exists. The attribute MPI_HOST has the same value on all processes of MPI_COMM_WORLD.	$46 \\ 47$
The autibute with interior has the same value on an processes of wirt_COWW_WORLD.	48

5	MPI_IO is the rank of a processor that can provide language-standard an, this means that all of the Fortran I/O operations are supported ITE). For C, this means that all of the ISO C I/O operations are for intf_lacek)				
$_{3}$ The value returned for $I$ I/O facilities For Fortu	an, this means that all of the Fortran I/O operations are supported ITE). For C, this means that all of the ISO C I/O operations are				
⁵ (e.g., OPEN, REWIND, WF supported (e.g. fopen	IDIIICI, ISEEK).				
7 If every process ca 8 will be returned. Oth	a provide language-standard I/O, then the value MPI_ANY_SOURCE erwise, if the calling process can provide language-standard I/O,				
¹⁰ I/O then the rank of ¹¹ returned by all processo	then its rank will be returned. Otherwise, if some process can provide language-standard I/O then the rank of one such process will be returned. The same value need not be returned by all processes. If no process can provide language-standard I/O, then the value MPI_PROC_NULL will be returned.				
¹² MPI_PROC_NULL will b	e returned.				
14	Note that input is not collective, and this attribute does <i>not</i> indicate or does provide input. ( <i>End of advice to users.</i> )				
¹⁶ 17 Clock Synchronization					
<ul> <li>¹⁹ MPI_COMM_WORLD ar</li> <li>²⁰ synchronized if explicit</li> <li>²¹ the variation in time, a</li> <li>²² round-trip time for an</li> <li>²³ before a send and at ar</li> <li>²⁴ be always higher than a</li> <li>²⁵ The attribute MPI</li> <li>²⁶ synchronized (however, attribute may be assoc</li> <li>²⁸ The attribute MPI</li> <li>²⁹ MPI_COMM_WORLD.</li> <li>³⁰ Inquire Processor Name</li> </ul>	MPI_WTIME_IS_GLOBAL is 1 if clocks at all processes in e synchronized, 0 otherwise. A collection of clocks is considered effort has been taken to synchronize them. The expectation is that is measured by calls to MPI_WTIME, will be less then one half the MPI message of length zero. If time is measured at a process just other process just after a matching receive, the second time should he first one. _WTIME_IS_GLOBAL need not be present when the clocks are not the attribute key MPI_WTIME_IS_GLOBAL is always valid). This ated with communicators other then MPI_COMM_WORLD. _WTIME_IS_GLOBAL has the same value on all processes of				
32					
<ul> <li>MPI_GET_PROCESSOI</li> </ul>	2_NAME(name, resultlen)				
36 OUT name 37	A unique specifier for the actual (as opposed to virtual) node.				
<ul> <li>³⁸ OUT resultlen</li> <li>⁴⁰</li> </ul>	Length (in printable characters) of the result returned in name				
<ul> <li>⁴¹ C binding</li> <li>⁴² int MPI_Get_process</li> <li>⁴³</li> </ul>	r_name(char *name, int *resultlen)				
44 Fortran 2008 bindin	-				
46     CHARACTER (LEN=M)       47     INTEGER, INTENT	me(name, resultlen, ierror) PI_MAX_PROCESSOR_NAME), INTENT(OUT) :: name OUT) :: resultlen L, INTENT(OUT) :: ierror				

Unofficial Draft for Comment Only

## Fortran binding MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERROR) CHARACTER*(*) NAME INTEGER RESULTLEN, IERROR

This routine returns the name of the processor on which it was called at the moment of the call. The name is a character string for maximum flexibility. From this value it must be possible to identify a specific piece of hardware; possible values include "processor 9 in rack 4 of mpp.cs.org" and "231" (where 231 is the actual processor number in the running homogeneous system). The argument name must represent storage that is at least MPI_MAX_PROCESSOR_NAME characters long. MPI_GET_PROCESSOR_NAME may write up to this many characters into name.

The number of characters actually written is returned in the output argument, resultlen. In C, a null character is additionally stored at name[resultlen]. The value of resultlen cannot be larger than MPI_MAX_PROCESSOR_NAME-1. In Fortran, name is padded on the right with blank characters. The value of resultlen cannot be larger than MPI_MAX_PROCESSOR_NAME.

*Rationale.* This function allows MPI implementations that do process migration to return the current processor. Note that nothing in MPI *requires* or defines process migration; this definition of MPI_GET_PROCESSOR_NAME simply allows such an implementation. (*End of rationale.*)

Advice to users. The user must provide at least MPI_MAX_PROCESSOR_NAME space to write the processor name—processor names can be this long. The user should examine the output argument, resultlen, to determine the actual length of the name. (End of advice to users.)

## 9.2 Memory Allocation

In some systems, message-passing and remote-memory-access (RMA) operations run faster when accessing specially allocated memory (e.g., memory that is shared by the other processes in the communicating group on an SMP). MPI provides a mechanism for allocating and freeing such special memory. The use of such memory for message-passing or RMA is not mandatory, and this memory can be used without restrictions as any other dynamically allocated memory. However, implementations may restrict the use of some RMA functionality as defined in Section 12.5.3.

MPI_ALLOC_MEM(size, info, baseptr)

IN	size	size of memory segment in bytes (non-negative integer)	40 41
IN	info	info argument (handle)	42
OUT	baseptr	pointer to beginning of memory segment allocated	43
001		pointer to beginning of memory beginent anotated	44
			45

C binding							
int	MPI_Alloc_mem(MPI_Aint	size,	MPI_Info	info,	void	<pre>*baseptr)</pre>	

 $\mathbf{2}$ 

3

 $\frac{4}{5}$ 

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

```
1
 Fortran 2008 binding
\mathbf{2}
 MPI_Alloc_mem(size, info, baseptr, ierror)
3
 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
4
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
5
 TYPE(MPI_Info), INTENT(IN) :: info
6
 TYPE(C_PTR), INTENT(OUT) :: baseptr
7
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
 Fortran binding
9
 MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
10
 INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
11
 INTEGER INFO, IERROR
12
13
 If the Fortran compiler provides TYPE(C_PTR), then the following generic interface must
14
 be provided in the mpi module and should be provided in mpif.h through overloading,
15
 i.e., with the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND)
16
 BASEPTR, but with a different specific procedure name:
17
18
 INTERFACE MPI_ALLOC_MEM
19
 SUBROUTINE MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
 IMPORT :: MPI_ADDRESS_KIND
20
21
 INTEGER INFO, IERROR
22
 INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
23
 END SUBROUTINE
^{24}
 SUBROUTINE MPI_ALLOC_MEM_CPTR(SIZE, INFO, BASEPTR, IERROR)
25
 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
26
 IMPORT :: MPI_ADDRESS_KIND
27
 INTEGER :: INFO, IERROR
 INTEGER(KIND=MPI_ADDRESS_KIND) ::
 SIZE
28
 TYPE(C_PTR) :: BASEPTR
29
30
 END SUBROUTINE
31
 END INTERFACE
32
 The base procedure name of this overloaded function is MPI_ALLOC_MEM_CPTR. The
33
 implied specific procedure names are described in Section 19.1.5.
34
 By default, the allocated memory shall be aligned to at least the alignment required
35
 for load/store accesses of any datatype corresponding to a predefined MPI datatype. The
36
 info argument may be used to specify a desired alternative minimum alignment in bytes for
37
 the allocated memory by setting the value of the key "mpi_minimum_memory_alignment" to an
38
 integral number equal to a power of two. An implementation may ignore values smaller than
39
 the default required alignment. The info argument can also be used to provide directives
40
 that control the desired location of the allocated memory. Such a directive does not affect
41
 the semantics of the call. The corresponding info values are implementation-dependent. A
42
 null directive value of info = MPI_INFO_NULL is always valid.
43
 The function MPI_ALLOC_MEM may return an error code of class MPI_ERR_NO_MEM
44
 to indicate it failed because memory is exhausted.
45
46
47
48
```

IN	base	initial address of memory segment allocated by
		MPI_ALLOC_MEM (choice)

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/)

```
1
 size = 4 * \text{shape}(1) * \text{shape}(2)
 ! assuming 4 bytes per REAL
2
 CALL MPI_Alloc_mem(size, MPI_INFO_NULL, p, ierr) ! memory is allocated and
3
 CALL C_F_POINTER(p, a, shape) ! intrinsic
 ! now accessible via a(i,j)
4
 ! in ISO_C_BINDING
 . . .
5
 a(3,5) = 2.71
6
 . . .
7
 CALL MPI_Free_mem(a, ierr)
 ! memory is freed
8
9
 Example 9.2 Example of use of MPI_ALLOC_MEM, in Fortran with non-standard Cray-
10
 pointers. We assume 4-byte REALS, and assume that these pointers are address-sized.
11
12
 REAL A
13
 ! no memory is allocated
 POINTER (P, A(100,100))
14
 INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
15
 SIZE = 4*100*100
16
 CALL MPI_ALLOC_MEM(SIZE, MPI_INFO_NULL, P, IERR)
17
 ! memory is allocated
18
 . . .
19
 A(3,5) = 2.71
20
 . . .
21
 CALL MPI_FREE_MEM(A, IERR) ! memory is freed
22
 This code is not Fortran 77 or Fortran 90 code. Some compilers may not support this
23
 code or need a special option, e.g., the GNU gFortran compiler needs -fcray-pointer.
24
25
 Advice to implementors. Some compilers map Cray-pointers to address-sized integers,
26
 some to TYPE(C_PTR) pointers (e.g., Cray Fortran, version 7.3.3). From the user's
27
 viewpoint, this mapping is irrelevant because Examples 9.2 should work correctly
28
 with an MPI-3.0 (or later) library if Cray-pointers are available. (End of advice to
29
 implementors.)
30
31
32
 Example 9.3 Same example, in C.
33
 float (* f)[100][100];
34
 /* no memory is allocated */
35
 MPI_Alloc_mem(sizeof(float)*100*100, MPI_INFO_NULL, &f);
36
 /* memory allocated */
37
 . . .
38
 (*f)[5][3] = 2.71;
39
40
 MPI_Free_mem(f);
41
42
43
 Error Handling
 9.3
44
```

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.

 $\mathbf{2}$ 

 $\mathbf{5}$ 

 24 

 31 

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 MPI_COMM_SELF is not initialized (i.e., before MPI_INIT / MPI_INIT_THREAD or after MPI_FINALIZE) 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 non-trivial 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 

 $\mathbf{2}$ 

3

4

5

6 7

8

9

10

11

12 13 or the user terminates the application. The user can make the determination of whether or not to attempt to continue when handling such an error.

Advice to users. For example, users may be unable to correct errors corresponding to some error classes, such as MPI_ERR_INTERN. Such errors may cause subsequent MPI calls to complete in error. (*End of advice to users.*)

Advice to implementors. A high-quality implementation will, to the greatest possible extent, circumscribe the impact of an error, so that normal processing can continue after an error handler was invoked. The implementation documentation will provide information on the possible effect of each class of errors and available recovery actions. (End of advice to implementors.)

An MPI error handler is an opaque object, which is accessed by a handle. MPI calls are provided to create new error handlers, to associate error handlers with objects, and to test which error handler is associated with an object. C has distinct typedefs for user defined error handling callback functions that accept communicator, file, window, and session arguments. In Fortran there are four user routines.

¹⁹ An error handler object is created by a call to MPI_XXX_CREATE_ERRHANDLER, ²⁰ where XXX is, respectively, COMM, WIN, FILE, or SESSION.

An error handler is attached to a communicator, window, file, or session by a call to MPI_XXX_SET_ERRHANDLER. The error handler must be either a predefined error handler, or an error handler that was created by a call to MPI_XXX_CREATE_ERRHANDLER, with matching XXX. An error handler can also be attached to a session using the errorhandler argument to MPI_SESSION_INIT. The predefined error handlers MPI_ERRORS_RETURN and MPI_ERRORS_ARE_FATAL can be attached to communicators, windows, files, or sessions.

The error handler currently associated with a communicator, window, file, or session can be retrieved by a call to MPI_XXX_GET_ERRHANDLER.

The MPI function MPI_ERRHANDLER_FREE can be used to free an error handler that was created by a call to MPI_XXX_CREATE_ERRHANDLER.

³¹ MPI_XXX_GET_ERRHANDLER behave as if a new error handler object is created. That ³² is, once the error handler is no longer needed, MPI_ERRHANDLER_FREE should be called ³³ with the error handler returned from MPI_XXX_GET_ERRHANDLER to mark the error ³⁴ handler for deallocation. This provides behavior similar to that of MPI_COMM_GROUP ³⁵ and MPI_GROUP_FREE.

Advice to implementors. High-quality implementations should raise an error when an error handler that was created by a call to MPI_XXX_CREATE_ERRHANDLER is attached to an object of the wrong type with a call to MPI_YYY_SET_ERRHANDLER. To do so, it is necessary to maintain, with each error handler, information on the typedef of the associated user function. (*End of advice to implementors.*)

The syntax for these calls is given below.

 $43 \\ 44$ 

36

37

38

39

40

- 45
- 46
- 47 48

9.3. ERROR HANDLING	457
9.3.1 Error Handlers for Communicators	
MPI_COMM_CREATE_ERRHANDLER(comm_errhandler_fn, errhandler)	
IN comm_errhandler_fn user defined error handling procedure (function)	
OUTerrhandlerMPI error handler (handle)	
C binding int MPI_Comm_create_errhandler(	
<pre>Fortran 2008 binding MPI_Comm_create_errhandler(comm_errhandler_fn, errhandler, ierror) PROCEDURE(MPI_Comm_errhandler_function), INTENT(IN) ::</pre>	۶
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, we is defined as typedef void MPI_Comm_errhandler_function(MPI_Comm *comm, int *error_cod );	
The first argument is the communicator in use. The second is the error code to returned by the MPI routine that raised the error. If the routine would have return	

error code to be ld 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:

```
38
ABSTRACT INTERFACE
 39
 SUBROUTINE MP1_Comm_errhandler_function(comm, error_code)
 TYPE(MPI_Comm) :: comm
 ^{41}
 INTEGER :: error_code
 42
With the Fortran mpi module and mpif.h, the user routine COMM_ERRHANDLER_FN
 43
should be of the form:
 44
SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)
 INTEGER COMM, ERROR_CODE
```

 31 

	458		CHAPTER 9. MPI ENVIRONMENTAL MANAGEMENT	
1 2 3		standard hook for provi	able argument list is provided because it provides an ISO- iding additional information to the error handler; without this additional arguments. ( <i>End of rationale.</i> )	
4 5 6 7 8		is associated with the a "global" error handle	newly created communicator inherits the error handler that "parent" communicator. In particular, the user can specify er for all communicators by associating this handler with the MM_WORLD immediately after initialization. ( <i>End of advice to</i>	
9 10 11		users.)		
12	MPI_0	COMM_SET_ERRHAND	DLER(comm, errhandler)	
13	INO	UT comm	communicator (handle)	
14 15	IN	errhandler	new error handler for communicator (handle)	
16	C bir	ding		
17 18			ler(MPI_Comm comm, MPI_Errhandler errhandler)	
19				
20		an 2008 binding	comm, errhandler, ierror)	
21		YPE(MPI_Comm), INTE		
22			, INTENT(IN) :: errhandler	
23			NTENT(OUT) :: ierror	
24 25	Fortr	an binding		
26		•	COMM, ERRHANDLER, IERROR)	
27		NTEGER COMM, ERRHAN		
28	۸	ttachas a nave annon ha	andler to a communicator. The error handler must be either	
29				
30	-	edefined error handler, or an error handler created by a call toCOMM_CREATE_ERRHANDLER.		
31				
32 33				
34	MPI_0	COMM_GET_ERRHANI	DLER(comm, errhandler)	
35	IN	comm	communicator (handle)	
36	OUT	errhandler	error handler currently associated with	
37			communicator (handle)	
38				
39	C bir	0		
40 41	int M	Pl_Comm_get_errhand	ler(MPI_Comm comm, MPI_Errhandler *errhandler)	
42	Fortr	an 2008 binding		
43		0	comm, errhandler, ierror)	
44		YPE(MPI_Comm), INTE		
45			, INTENT(OUT) :: errhandler	
46	I	NIEGER, UPTIONAL, I	NTENT(OUT) :: ierror	
47	Fortr	an binding		
48	MPI_C	OMM_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

			10
MPI_WIN_CREATE_ERRHANDLER(win_errhandler_fn, errhandler)			
IN	win_errhandler_fn	user defined error handling procedure (function)	12
OUT	errhandler	MPI error handler (handle)	13
001	ermanuer	with error manufer (manufe)	14
C bindin	۲.		15
	s √in_create_errhandler(		16 17
1110 111 1_1		unction *win_errhandler_fn,	18
	MPI_Errhandler *errh		19
<b>D</b> anta a f			20
	2008 binding	andlen fr. emphandlen termen)	21
		<pre>nandler_fn, errhandler, ierror) function), INTENT(IN) :: win_errhandler_fn</pre>	22
	(MPI_Errhandler), INTENT((		23
	GER, OPTIONAL, INTENT(OUT)		24
			25
Fortran	3		26
MPI_WIN_CREATE_ERRHANDLER(WIN_ERRHANDLER_FN, ERRHANDLER, IERROR)			27
EXTERNAL WIN_ERRHANDLER_FN			28
			29 30
		be attached to a window object. The user routine	31
	,	Win_errhandler_function which is defined as	32
<pre>typedef void MPI_Win_errhandler_function(MPI_Win *win, int *error_code,</pre>			33
);			34
The first argument is the window in use, the second is the error code to be re-			35
		varargs" arguments whose number and meaning is	36
implement	tation-dependent. An impleme	entation should clearly document these arguments.	37
With the l	Fortran mpi_f08 module, the u	ser routine win_errhandler_fn should be of the form:	38
ABSTRACT	INTERFACE		39
	TINE MPI_Win_errhandler_fu	<pre>inction(win, error_code)</pre>	40
	(MPI_Win) :: win		41
INTE(	GER :: error_code		42
With the l	Fortran mpi module and mpif.	h, the user routine WIN_ERRHANDLER_FN should	43 44
be of the form:			44 45
SUBROUTI	SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)		
INTE	INTEGER WIN, ERROR_CODE		
			47 48

/indows

1 2

3

4

 $\mathbf{5}$ 

6 7

```
1
 MPI_WIN_SET_ERRHANDLER(win, errhandler)
\mathbf{2}
 INOUT
 window object (handle)
 win
3
 IN
 errhandler
 new error handler for window (handle)
4
5
6
 C binding
7
 int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)
8
 Fortran 2008 binding
9
 MPI_Win_set_errhandler(win, errhandler, ierror)
10
 TYPE(MPI_Win), INTENT(IN) :: win
11
 TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
12
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
 Fortran binding
15
 MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
16
 INTEGER WIN, ERRHANDLER, IERROR
17
 Attaches a new error handler to a window. The error handler must be either a pre-
18
 defined error handler, or an error handler created by a call to
19
 MPI_WIN_CREATE_ERRHANDLER.
20
21
22
 MPI_WIN_GET_ERRHANDLER(win, errhandler)
23
 IN
 window object (handle)
 win
^{24}
 OUT
 errhandler
 error handler currently associated with window
25
26
 (handle)
27
28
 C binding
29
 int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)
30
 Fortran 2008 binding
^{31}
 MPI_Win_get_errhandler(win, errhandler, ierror)
32
 TYPE(MPI_Win), INTENT(IN) :: win
33
 TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
34
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
 Fortran binding
37
 MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
38
 INTEGER WIN, ERRHANDLER, IERROR
39
 Retrieves the error handler currently associated with a window.
40
41
42
43
44
45
46
47
48
```

9.3. ERROR HANDLING	461	
9.3.3 Error Handlers for Files	1	
	3	
	4	
MPI_FILE_CREATE_ERRHANDLER(file_errhandler_fn, errhandler)	5	
IN file_errhandler_fn user defined error handling procedu	re (function) ₆	
OUT         errhandler         MPI error handler (handle)	7	
	8	
C binding	9	
<pre>int MPI_File_create_errhandler(</pre>	10	
MPI_File_errhandler_function *file_errhandler_fn	<b>,</b> 11 , 12	
MPI_Errhandler *errhandler)	12	
Fortran 2008 binding	14	
<pre>MPI_File_create_errhandler(file_errhandler_fn, errhandler, ier</pre>	rror) 15	
<pre>PROCEDURE(MPI_File_errhandler_function), INTENT(IN) ::</pre>	16	
file_errhandler_fn	17	
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler	18	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	19	
Fortran binding	20 21	
MPI_FILE_CREATE_ERRHANDLER(FILE_ERRHANDLER_FN, ERRHANDLER, IEF	21 ROR) 22	
EXTERNAL FILE_ERRHANDLER_FN	23	
INTEGER ERRHANDLER, IERROR	24	
Creates an error handler that can be attached to a file object. The use	er routine should $_{25}$	
be, in C, a function of type $MPI_File_errhandler_function,$ which is defined as		
<pre>typedef void MPI_File_errhandler_function(MPI_File *file, int</pre>	*error_code, 27	
);	28	
The first argument is the file in use, the second is the error code to be r	eturned. The re-	
maining arguments are "varargs" arguments whose number and meaning is		
dependent. An implementation should clearly document these arguments.	31	
With the Fortran mpi_f08 module, the user routine file_errhandler_fn shoul	d be of the form: $32$	
ABSTRACT INTERFACE	34	
<pre>SUBROUTINE MPI_File_errhandler_function(file, error_code) TYPE(MPI_File) :: file</pre>	35	
INTEGER :: error_code	36	
	37	
With the Fortran mpi module and mpif.h, the user routine FILE_ERRHAN	DLER_FN should 38	
be of the form:		
SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)	40	
INTEGER FILE, ERROR_CODE	41	
	42 43	
	43	
	45	
	46	

```
1
 MPI_FILE_SET_ERRHANDLER(file, errhandler)
2
 INOUT
 file
 file (handle)
3
 IN
 errhandler
 new error handler for file (handle)
4
5
6
 C binding
7
 int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)
8
 Fortran 2008 binding
9
 MPI_File_set_errhandler(file, errhandler, ierror)
10
 TYPE(MPI_File), INTENT(IN) :: file
11
 TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
12
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
 Fortran binding
 MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
15
16
 INTEGER FILE, ERRHANDLER, IERROR
17
 Attaches a new error handler to a file. The error handler must be either a predefined
18
 error handler, or an error handler created by a call to MPI_FILE_CREATE_ERRHANDLER.
19
20
21
 MPI_FILE_GET_ERRHANDLER(file, errhandler)
22
 IN
 file
 file (handle)
23
 OUT
 errhandler
 error handler currently associated with file (handle)
^{24}
25
26
 C binding
27
 int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)
28
 Fortran 2008 binding
29
 MPI_File_get_errhandler(file, errhandler, ierror)
30
 TYPE(MPI_File), INTENT(IN) :: file
^{31}
 TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
32
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
 Fortran binding
35
 MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
36
 INTEGER FILE, ERRHANDLER, IERROR
37
 Retrieves the error handler currently associated with a file.
38
39
40
41
42
43
44
45
46
47
48
```

9.3.4 Error Handlers for Sessions			
			2
			3
MPI_SESS	ION_CREATE_ERRHANDLER	(session_errhandler_fn, errhandler)	4 5
IN	session_errhandler_fn	user defined error handling procedure (function)	6
OUT	errhandler	MPI error handler (handle)	7
			8
C binding		,	9 10
int MP1_S	Session_create_errhandler MPI_Session_errhandl	er_function *session_errhandler_fn,	11
	MPI_Errhandler *errh		12
Fortran 2	2008 binding		13
	_	sion_errhandler_fn, errhandler, ierror)	14 15
PROCE	DURE(MPI_Session_errhand	<pre>ler_function), INTENT(IN) ::</pre>	16
	session_errhandler_:		17
	MPI_Errhandler), INTENT(( ER, OPTIONAL, INTENT(OUT)		18
			19 20
Fortran b	8	SION_ERRHANDLER_FN, ERRHANDLER, IERROR)	21
	NAL SESSION_ERRHANDLER_F		22
INTEG	ER ERRHANDLER, IERROR		23
Create	es an error handler that can	be attached to a session object. In C, the	24 25
and a subardlar for a summary thread has a function of target MDL Cardina to the function			26
			27
typedef v		er_function(MPI_Session *session,	28
	<pre>int *error_code,</pre>		29 30
		ise, the second is the error code to be returned. The	0.1
		uments whose number and meaning is implementation- learly document these arguments.	32
		e session_errhandler_fn argument should be of the	33
form:	-		34 35
	INTERFACE		36
	<pre>INE MP1_Session_errhandle (MP1_Session) :: session</pre>	er_function(session, error_code)	37
	ER :: error_code		38
With the	Fortran mni module and mn	if.h, the SESSION_ERRHANDLER_FN argument	$\frac{39}{40}$
	of the form:		40
SUBROUTIN	E SESSION_ERRHANDLER_FUN	CTION(SESSION, ERROR_CODE)	42
INTEG	ER SESSION, ERROR_CODE		43
			44
			45 46
			47

```
1
 MPI_SESSION_SET_ERRHANDLER(session, errhandler)
2
 INOUT
 session
 session (handle)
3
 IN
 errhandler
 new error handler for session (handle)
4
5
6
 C binding
 int MPI_Session_set_errhandler(MPI_Session session,
7
8
 MPI_Errhandler errhandler)
9
 Fortran 2008 binding
10
 MPI_Session_set_errhandler(session, errhandler, ierror)
11
 TYPE(MPI_Session), INTENT(IN) :: session
12
 TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
13
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
 Fortran binding
16
 MPI_SESSION_SET_ERRHANDLER(SESSION, ERRHANDLER, IERROR)
17
 INTEGER SESSION, ERRHANDLER, IERROR
18
 Attaches a new error handler to a session. The error handler must be either a pre-
19
 defined error handler, or an error handler created by a call to
20
 MPI_SESSION_CREATE_ERRHANDLER.
21
22
23
 MPI_SESSION_GET_ERRHANDLER(session, errhandler)
^{24}
 IN
 session
 session (handle)
25
26
 OUT
 errhandler
 error handler currently associated with session
 (handle)
27
28
29
 C binding
30
 int MPI_Session_get_errhandler(MPI_Session session,
^{31}
 MPI_Errhandler *errhandler)
32
 Fortran 2008 binding
33
 MPI_Session_get_errhandler(session, errhandler, ierror)
34
 TYPE(MPI_Session), INTENT(IN) :: session
35
 TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
36
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
 Fortran binding
39
 MPI_SESSION_GET_ERRHANDLER(SESSION, ERRHANDLER, IERROR)
40
 INTEGER SESSION, ERRHANDLER, IERROR
41
 Retrieves the error handler currently associated with a session.
42
43
44
45
46
47
48
```

9.3. E	RROR HANDLING	465	
9.3.5	Freeing Errorhandlers and R	etrieving Error Strings	1 2
			3
MPI F	RRHANDLER_FREE(errhandle	r)	4
	``	,	5
INOL	JT errhandler	MPI error handler (handle)	6
<b>a</b> 1 •	1		7
C bin	aing PI_Errhandler_free(MPI_Er:	whendlen terrhendlen)	8
IIIC M	-1_EIIMANGIEI_IIEE(MF1_EI		9 10
	an 2008 binding		10
	rhandler_free(errhandler		12
	PE(MPI_Errhandler), INTE		13
11	VTEGER, OPTIONAL, INTENT()	UUT) :: lerror	14
Fortra	an binding		15
MPI_EF	RHANDLER_FREE (ERRHANDLER	, IERROR)	16
II	NTEGER ERRHANDLER, IERROR		17
Μ	arks the error handler associa	ted with errhandler for deallocation and sets errhandler	18
		ror handler will be deallocated after all the objects	19
associa	tted with it (communicator, w	indow, or file) have been deallocated.	20 21
			21
	RROR_STRING(errorcode, stri	ng resultion)	23
	· ·		24
IN	errorcode	Error code returned by an MPI routine	25
OUT	string	Text that corresponds to the errorcode	26
OUT	resultlen	Length (in printable characters) of the result	27
		returned in string	28
			29
C bin	ding		30
int MF	PI_Error_string(int error	code, char *string, int *resultlen)	31
Fortra	an 2008 binding		32 33
	rror_string(errorcode, st	ring, resultlen, ierror)	34
	NTEGER, INTENT(IN) :: err	-	35
		R_STRING), INTENT(OUT) :: string	36
IN	NTEGER, INTENT(OUT) :: real	sultlen	37
II	TEGER, OPTIONAL, INTENT(	DUT) :: ierror	38
Fortra	an binding		39
	RROR_STRING(ERRORCODE, ST	RING, RESULTLEN, IERROR)	40
INTEGER ERRORCODE, RESULTLEN, IERROR			41
	HARACTER*(*) STRING		42
П	tunna the owner string according	ted with an amon and an class. The annument stairs	43
	0	st MPI_MAX_ERROR_STRING characters long.	44 45
	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.

¹⁸ When an operation raises an error, it may not satisfy its specification (for example, a ²⁰ synchronizing operation may not have synchronized) and the content of the output buffers, ²¹ targeted memory, or output parameters is undefined. However, a valid error code shall ²² always be set when an operation raises an error, whether in the return value, error field in ²³ 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,

31 32 33

34

35

36

37

38

 $0 = \mathsf{MPI}_\mathsf{SUCCESS} < \mathsf{MPI}_\mathsf{ERR}_\dots \ \leq \mathsf{MPI}_\mathsf{ERR}_\mathsf{LASTCODE}.$ 

*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.*)

#### 39 40 41

⁴² MPI_ERROR_CLASS(errorcode, errorclass)

43	IN	errorcode	Error code returned by an MPI routine
44 45	OUT	errorclass	Error class associated with errorcode
46			
47	C bindir	ıg	

48 int MPI_Error_class(int errorcode, int *errorclass)

1

2

3

4

5

6

7 8

9 10

11

12

13

14

15

16

	NT	-
MPI_SUCCESS	No error	1 2
MPI_ERR_ACCESS	Permission denied	
MPI_ERR_AMODE	Error related to the amode passed to	3
MPI_ERR_ARG	MPI_FILE_OPEN Invalid argument of some other kind	4 5
	Invalid assertion argument	э 6
MPI_ERR_ASSERT MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)	7
MPI_ERR_BASE	Invalid base passed to MPI_FREE_MEM	8
MPI_ERR_BUFFER	Invalid buffer pointer argument	9
MPI_ERR_COMM	Invalid communicator argument	10
MPI_ERR_CONVERSION	An error occurred in a user supplied data	11
	conversion function	12
MPI_ERR_COUNT	Invalid count argument	13
MPI_ERR_DIMS	Invalid dimension argument	14
MPI_ERR_DISP	Invalid displacement argument	15
MPI_ERR_DUP_DATAREP	Conversion functions could not be regis-	16
	tered because a data representation identi-	17
	fier that was already defined was passed to	18
	MPI_REGISTER_DATAREP	19
MPI_ERR_FILE	Invalid file handle argument	20
MPI_ERR_FILE_EXISTS	File exists	21
MPI_ERR_FILE_IN_USE	File operation could not be completed, as	22
	the file is currently open by some process	23
MPI_ERR_GROUP	Invalid group argument	24
MPI_ERR_INFO	Invalid info argument	25
MPI_ERR_INFO_KEY	Key longer than $MPI_MAX_INFO_KEY$	26
MPI_ERR_INFO_NOKEY	Invalid key passed to MPI_INFO_DELETE	27
MPI_ERR_INFO_VALUE	Value longer than MPI_MAX_INFO_VAL	28
MPI_ERR_IN_STATUS	Error code is in status	29
MPI_ERR_INTERN	Internal MPI (implementation) error	30
MPI_ERR_IO	Other I/O error	31
MPI_ERR_KEYVAL	Invalid keyval argument	32
MPI_ERR_LOCKTYPE	Invalid locktype argument	33
MPI_ERR_NAME	Invalid service name passed to	34
	MPI_LOOKUP_NAME	35
MPI_ERR_NO_MEM	MPI_ALLOC_MEM failed because memory	36
	is exhausted	37
MPI_ERR_NO_SPACE	Not enough space	38
MPI_ERR_NO_SUCH_FILE	File does not exist	39
MPI_ERR_NOT_SAME	Collective argument not identical on all	40
	processes, or collective routines called in	41
	a different order by different processes	42 43
		43 44
		44
Table 9	.1: Error classes (Part 1)	46

Unofficial Draft for Comment Only

1	MPI_ERR_OP	Invalid operation argument
2	MPI_ERR_OTHER	Known error not in this list
3	MPI_ERR_PENDING	Pending request
4	MPI_ERR_PORT	Invalid port name passed to
5		MPI_COMM_CONNECT
6	MPI_ERR_PROC_ABORTED	Operation failed because a peer process has
7		aborted
8	MPI_ERR_QUOTA	Quota exceeded
9	MPI_ERR_RANK	Invalid rank argument
10	MPI_ERR_READ_ONLY	Read-only file or file system
11	MPI_ERR_REQUEST	Invalid request argument
12	MPI_ERR_RMA_ATTACH	Memory cannot be attached (e.g., because
13		of resource exhaustion)
14	MPI_ERR_RMA_CONFLICT	Conflicting accesses to window
15	MPI_ERR_RMA_FLAVOR	Passed window has the wrong flavor for the
16		called function
17	MPI_ERR_RMA_RANGE	Target memory is not part of the win-
18		dow (in the case of a window created
19		with MPI_WIN_CREATE_DYNAMIC, tar-
20		get memory is not attached)
21	MPI_ERR_RMA_SHARED	Memory cannot be shared (e.g., some pro-
22		cess in the group of the specified commu-
23		nicator cannot expose shared memory)
24	MPI_ERR_RMA_SYNC	Wrong synchronization of RMA calls
25	MPI_ERR_ROOT	Invalid root argument
26	MPI_ERR_SERVICE	Invalid service name passed to
27		MPI_UNPUBLISH_NAME
28	MPI_ERR_SESSION	Invalid session argument
29	MPI_ERR_SIZE	Invalid size argument
30	MPI_ERR_SPAWN	Error in spawning processes
31	MPI_ERR_TAG	Invalid tag argument
32	MPI_ERR_TOPOLOGY	Invalid topology argument
33	MPI_ERR_TRUNCATE	Message truncated on receive
34	MPI_ERR_TYPE	Invalid datatype argument
35	MPI_ERR_UNKNOWN	Unknown error
36	MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to
37		MPI_FILE_SET_VIEW
38	MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on
39		a file which supports sequential access only
40	MPI_ERR_VALUE_TOO_LARGE	Value is too large to store
41	MPI_ERR_WIN	Invalid window argument
42	MPI_ERR_LASTCODE	Last error code
43		
44		an alagana (Dart 2)
45	Table 9.2: Err	or classes (Part 2)
46		
47		
48		

Fortran 2008 binding MPI_Error_class(errorcode, errorclass, ierror) INTEGER, INTENT(IN) :: errorcode INTEGER, INTENT(OUT) :: errorclass	1 2 3 4
INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR) INTEGER ERRORCODE, ERRORCLASS, IERROR	5 6 7 8 9
The function MPI_ERROR_CLASS maps each standard error code (error class) onto itself. This function must always be thread-safe, as defined in Section 11.6. It is one of the few routines that may be called before MPI is initialized or after MPI is finalized.	10 11 12 13
9.5 Error Classes, Error Codes, and Error Handlers	14 15 16 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:	18 19 20
1. add a new error class to the ones an MPI implementation already knows.	21
2. associate error codes with this error class, so that MPI_ERROR_CLASS works.	22 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 26
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.	27 28 29 30 31
MPI_ADD_ERROR_CLASS(errorclass)	32 33
OUT errorclass value for the new error class (integer)	34 35
C binding int MPI_Add_error_class(int *errorclass)	36 37 38
Fortran 2008 binding MPI_Add_error_class(errorclass, ierror) INTEGER, INTENT(OUT) :: errorclass INTEGER, OPTIONAL, INTENT(OUT) :: ierror	39 40 41 42
Fortran binding MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR) INTEGER ERRORCLASS, IERROR	43 44 45
Creates a new error class and returns the value for it.	46 47 48

1	Rationale. To avoid conflicts with existing error codes and classes, the value is set
2	by the implementation and not by the user. (End of rationale.)
3	
4	Advice to users. Since a call to MPI_ADD_ERROR_CLASS is local, the same errorclass
5	may not be returned on all processes that make this call. Thus, it is not safe to assume
6	that registering a new error on a set of processes at the same time will yield the same
7	errorclass on all of the processes. Getting the "same" error on multiple processes may
8	not cause the same value of error code to be generated. (End of advice to users.)
9	
10	The value of MPI_ERR_LASTCODE is a constant value and is not affected by new user-
11	defined error codes and classes. Instead, a predefined attribute key $MPI_LASTUSEDCODE$ is
12	associated with $MPI_COMM_WORLD.$ The attribute value corresponding to this key is the
13	current maximum error class including the user-defined ones. This is a local value and may
14	be different on different processes. The value returned by this key is always greater than or
15	equal to MPI_ERR_LASTCODE.
16	
17	Advice to users. The value returned by the key MPI_LASTUSEDCODE will not change
18	unless the user calls a function to explicitly add an error class/code. In a multithreaded
19	environment, the user must take extra care in assuming this value has not changed.
20	Note that error codes and error classes are not necessarily dense. A user may not
21	assume that each error class below $MPI_LASTUSEDCODE$ is valid. (End of advice to
22	users.)
23	
24	
25	MPI_ADD_ERROR_CODE(errorclass, errorcode)
26	IN error class (integer)
27	
28	OUT         errorcode         new error code to be associated with errorclass
29	(integer)
30	
31	C binding
32	<pre>int MPI_Add_error_code(int errorclass, int *errorcode)</pre>
33	Fortran 2008 binding
34	MPI_Add_error_code(errorclass, errorcode, ierror)
35	INTEGER, INTENT(IN) :: errorclass
36	INTEGER, INTENT(OUT) :: errorcode
37	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38	
39	Fortran binding
40	MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR)
41	INTEGER ERRORCLASS, ERRORCODE, IERROR
42	Creates new error code associated with errorclass and returns its value in errorcode.
43	Oreares new error code associated with errorclass and returns its value in errorcode.
44	Rationale. To avoid conflicts with existing error codes and classes, the value of the
45	new error code is set by the implementation and not by the user. ( <i>End of rationale.</i> )
46	
47	
48	

MPI_AD	D_ERROR_STRING(	errorcode, string)	1
IN	errorcode	error code or class (integer)	2
IN	string	text corresponding to errorcode (string)	3 4
			4 5
C bindi	ng		6
		(int errorcode, const char *string)	7
Fortran	2008 binding		8
	0	orcode, string, ierror)	9
	EGER, INTENT(IN)		10 11
		TENT(IN) :: string	12
INT	EGER, OPTIONAL, I	NTENT(OUT) :: ierror	13
Fortran	binding		14
		ORCODE, STRING, IERROR)	15
	EGER ERRORCODE, I		16 17
CHA	RACTER*(*) STRING		17
		ng with an error code or class. The string must be no m	19
		NG characters long. The length of the string is as defined in	20
0	0 0 0	of the string does not include the null terminator in C. Trai tran. Calling MPI_ADD_ERROR_STRING for an errorcode t	21
		lace the old string with the new string. It is erroneous to	call
	• •	for an error code or class with a value $\leq$ MPI_ERR_LASTCC	23
		is called when no string has been set, it will return a em	
0 (	ll spaces in Fortran,		26
		ne methods for creating and associating error handlers w	with 27
commun	icators, files, window	75, and sessions.	28
			29
MPI_CO	MM_CALL_ERRHAN	NDLER(comm, errorcode)	30 31
IN	comm	communicator with error handler (handle)	32
IN	errorcode	error code (integer)	33
			34
C bindi	ng		35
int MPI	_Comm_call_errhan	dler(MPI_Comm comm, int errorcode)	36 37
Fortran	2008 binding		38
MPI_Com	m_call_errhandler	(comm, errorcode, ierror)	39
	E(MPI_Comm), INTE		40
	EGER, INTENT(IN)		41
	CGER, UPIIUNAL, I	NTENT(OUT) :: ierror	42
	binding		43 44
		(COMM, ERRORCODE, IERROR)	44 45
	EGER COMM, ERRORC	UDE, IERKUK	46
This	s function invokes th	e error handler assigned to the communicator with the e	rror

This function invokes the error handler assigned to the communicator with the error  $_{47}$  code supplied. This function returns MPI_SUCCESS in C and the same value in IERROR if  $_{48}$ 

```
1
 the error handler was successfully called (assuming the process is not aborted and the error
\mathbf{2}
 handler returns).
3
4
 MPI_WIN_CALL_ERRHANDLER(win, errorcode)
5
6
 IN
 window with error handler (handle)
 win
7
 IN
 errorcode
 error code (integer)
8
9
 C binding
10
 int MPI_Win_call_errhandler(MPI_Win win, int errorcode)
11
12
 Fortran 2008 binding
13
 MPI_Win_call_errhandler(win, errorcode, ierror)
14
 TYPE(MPI_Win), INTENT(IN) :: win
15
 INTEGER, INTENT(IN) :: errorcode
16
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
 Fortran binding
18
 MPI_WIN_CALL_ERRHANDLER(WIN, ERRORCODE, IERROR)
19
 INTEGER WIN, ERRORCODE, IERROR
20
21
 This function invokes the error handler assigned to the window with the error code
22
 supplied. This function returns MPI_SUCCESS in C and the same value in IERROR if the
23
 error handler was successfully called (assuming the process is not aborted and the error
^{24}
 handler returns).
25
26
 Advice to users.
 In contrast to communicators, the error handler
27
 MPI_ERRORS_ARE_FATAL is associated with a window when it is created. (End of
28
 advice to users.)
29
30
^{31}
 MPI_FILE_CALL_ERRHANDLER(fh, errorcode)
32
33
 IN
 fh
 file with error handler (handle)
34
 IN
 errorcode
 error code (integer)
35
36
 C binding
37
 int MPI_File_call_errhandler(MPI_File fh, int errorcode)
38
39
 Fortran 2008 binding
40
 MPI_File_call_errhandler(fh, errorcode, ierror)
41
 TYPE(MPI_File), INTENT(IN) :: fh
42
 INTEGER, INTENT(IN) :: errorcode
43
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
 Fortran binding
45
 MPI_FILE_CALL_ERRHANDLER(FH, ERRORCODE, IERROR)
46
 INTEGER FH, ERRORCODE, IERROR
47
48
```

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, MPI_SECCION_CALL_ERRHANDLER,

 $\mathsf{MPI}_\mathsf{WIN}_\mathsf{CALL}_\mathsf{ERRHANDLER},$  or  $\mathsf{MPI}_\mathsf{SESSION}_\mathsf{CALL}_\mathsf{ERRHANDLER}$  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

#### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 24 

 $45 \\ 46$ 

1 timers (both in POSIX 1003.1-1988 and 1003.4D 14.1 and in Fortran 90) are either incon- $\mathbf{2}$ venient or do not provide adequate access to high resolution timers. See also Section 2.6.4. 3 4 MPI_WTIME() 56 C binding 7 double MPI_Wtime(void) 8 9 Fortran 2008 binding 10 DOUBLE PRECISION MPI_Wtime() 11 Fortran binding 12DOUBLE PRECISION MPI_WTIME() 13 14MPI_WTIME returns a floating-point number of seconds, representing elapsed wall-15clock time since some time in the past. 16The "time in the past" is guaranteed not to change during the life of the process. 17The user is responsible for converting large numbers of seconds to other units if they are 18 preferred. 19This function is portable (it returns seconds, not "ticks"), and it allows high-resolution. 20One would use it like this: 2122ſ 23double starttime, endtime; 24starttime = MPI_Wtime(); 25. . . stuff to be timed 26= MPI_Wtime(); endtime 27printf("That took %f seconds\n", endtime-starttime); 28} 29 The times returned are local to the node that called them. There is no requirement 30 that different nodes return "the same time." (But see also the discussion of  31 MPI_WTIME_IS_GLOBAL in Section 9.1.2). 32 33 34MPI_WTICK() 35 36 C binding 37 double MPI_Wtick(void) 38 39 Fortran 2008 binding DOUBLE PRECISION MPI_Wtick() 4041 Fortran binding 42DOUBLE PRECISION MPI_WTICK() 43 44MPI_WTICK returns the resolution of MPI_WTIME in seconds. That is, it returns, 45as a double precision value, the number of seconds between successive clock ticks. For 46example, if the clock is implemented by the hardware as a counter that is incremented 47every millisecond, the value returned by MPI_WTICK should be  $(10^{-3})$ . 48

# 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, MPI_INFO_GET_VALUELEN, MPI_INFO_GET, 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 IN or INOUT argument to any MPI routine, it is parsed before that routine returns, so that it may be read, modified or freed immediately after return.

 24 

 31 

 $45 \\ 46$ 

1 When the descriptions refer to a key or value as being a boolean, an integer, or a list,  $\mathbf{2}$ they mean the string representation of these types. An implementation may define its own 3 rules for how info value strings are converted to other types, but to ensure portability, every 4 implementation must support the following representations. Valid values for a boolean must 5include the strings "true" and "false" (all lowercase). For integers, valid values must include 6 string representations of decimal values of integers that are within the range of a standard 7 integer type in the program. (However it is possible that not every integer is a valid value 8 for a given key.) On positive numbers, + signs are optional. No space may appear between 9 a + or - sign and the leading digit of a number. For comma separated lists, the string 10 must contain valid elements separated by commas. Leading and trailing spaces are stripped 11automatically from the types of info values described above and for each element of a comma 12separated list. These rules apply to all info values of these types. Implementations are free 13 to specify a different interpretation for values of other info keys. 1415MPI_INFO_CREATE(info) 1617OUT info info object created (handle) 18 19C binding 20int MPI_Info_create(MPI_Info *info) 21Fortran 2008 binding 22 MPI_Info_create(info, ierror) 23TYPE(MPI_Info), INTENT(OUT) :: info 24INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2526Fortran binding 27MPI_INFO_CREATE(INFO, IERROR) 28INTEGER INFO, IERROR 29 MPI_INFO_CREATE creates a new info object. The newly created object contains no 30 key/value pairs.  31 32 33 MPI_INFO_SET(info, key, value) 34 INOUT info info object (handle) 35 36 IN key key (string) 37 IN value value (string) 38 39 C binding 40 int MPI_Info_set(MPI_Info info, const char *key, const char *value) 41 42Fortran 2008 binding 43 MPI_Info_set(info, key, value, ierror) 44 TYPE(MPI_Info), INTENT(IN) :: info 45CHARACTER(LEN=*), INTENT(IN) :: key, value 46 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4748

Fortran binding			1
			2
INTE	GER INFO, IERROR		3
CHAR	ACTER*(*) KEY, VALUE		4
MPI	INFO SET adds the (key value	) pair to info, and overrides the value if a value for	5
	· · ·	l value are null-terminated strings in C. In Fortran,	6
	· · ·	value are stripped. If either key or value are larger	7 8
0	· ·	MPI_ERR_INFO_KEY or MPI_ERR_INFO_VALUE are	8 9
raised, res	pectively.		10
			11
	DELETE(info kov)		12
	DELETE(info, key)		13
INOUT	info	info object (handle)	14
IN	key	key (string)	15
			16
C bindin	g		17
int MPI_	Info_delete(MPI_Info info	, const char *key)	18 19
Fortran 2	2008 binding		20
	_delete(info, key, ierror)		20 21
TYPE	(MPI_Info), INTENT(IN) ::	info	22
	ACTER(LEN=*), INTENT(IN)		23
INTE	GER, OPTIONAL, INTENT(OUT)	) :: ierror	24
Fortran l	binding		25
MPI_INFO	_DELETE(INFO, KEY, IERROR)	)	26
INTEG	GER INFO, IERROR		27
CHAR	ACTER*(*) KEY		28 29
MPI_INFO_DELETE deletes a (key,value) pair from info. If key is not defined in info,			29 30
the call reigns on orner of class MDL EDD INFO NOKEY			31
			32
			33
	_GET(info, key, valuelen, value	2,	34
IN	info	info object (handle)	35
IN	key	key (string)	36
IN	valuelen	length of value arg (integer)	37
OUT	value	value (string)	38 39
OUT	flag	true if key defined, false if not (boolean)	40
001	nag	the firkey defined, tase if not (boolean)	41
C bindin	ď		42
	-	onst char *key, int valuelen, char *value,	43
· · ·	int *flag)	, , <u></u>	44
Fontan (	4		
			46 47
MPI_Info_get(info, key, valuelen, value, flag, ierror)47TYPE(MPI_Info), INTENT(IN) :: info48			47 48
111 15		111.0	-10

1			
1		ACTER(LEN=*), INT	
2		GER, INTENT(IN) :	
3			n), INTENT(OUT) :: value
4		CAL, INTENT(OUT)	0
5	INTEC	GER, OPTIONAL, IN	TENT(OUT) :: ierror
6	Fortran l	ainding	
7		0	ALUELEN, VALUE, FLAG, IERROR)
8		GER INFO, VALUELE	
9		ACTER*(*) KEY, VA	
10		CAL FLAG	
11	LUGI		
12			ne value associated with key in a previous call to
13			y exists, it sets flag to true and returns the value in value,
14		•	d leaves value unchanged. valuelen is the number of characters
15			than the actual size of the value, the value is truncated. In
16			s than the amount of allocated space to allow for the null
17	terminator		
18	If key	is larger than MPI_I	MAX_INFO_KEY, the call is erroneous.
19			
20		CET VALUELENI	nfo, key, valuelen, flag)
21			
22 23	IN	info	info object (handle)
23	IN	key	key (string)
25	OUT	valuelen	length of value arg (integer)
26	OUT	flag	true if key defined, false if not (boolean)
27			
28	C bindin	g	
29	int MPI_	_ [nfo_get_valuelen	(MPI_Info info, const char *key, int *valuelen,
30		int *flag)	
31	Fontman	0000 hinding	
32		2008 binding	, hur unluslen flag isource)
33		_get_valuelen(inf (MPI_Info), INTEN	o, key, valuelen, flag, ierror)
34		ACTER(LEN=*), INTEN	
35		GER, INTENT(OUT)	
36		CAL, INTENT(OUT)	
37			TENT(OUT) :: ierror
38		ER, UPIIONAL, IN	
39	Fortran l	oinding	
40 41	MPI_INFO_	_GET_VALUELEN(INF	O, KEY, VALUELEN, FLAG, IERROR)
41	INTEC	GER INFO, VALUELE	N, IERROR
	CHARA	ACTER*(*) KEY	
43 44	LOGIC	CAL FLAG	
44	Retrie	eves the length of the	e value associated with key. If key is defined, valuelen is set to
46		ē	tue and flag is set to true. If key is not defined, valuelen is not
47	-		The length returned in C does not include the end-of-string
48	character	TO HOR TO DOT TO HORSE.	The longen regulated in C does not include the chu-of-stilling

 48 

character.

If key	is larger than $MPI_MAX_INFC$	<b>D_KEY</b> , the call is erroneous.	1
			2
MPI_INFC	_GET_STRING(info, key, bufle	en, value, flag)	3 4
IN	info	info object (handle)	5
IN	key	key (string)	6
	-		7
INOUT	buflen	length of buffer (integer)	8
OUT	value	value (string)	9 10
OUT	flag	true if key defined, false if not (boolean)	11
C bindin	σ		12
	-	info, const char *key, int *buflen,	13 14
-	char *value, int *fl	-	14 15
Fortran	2008 binding		16
	0	flen, value, flag, ierror)	17
	(MPI_Info), INTENT(IN) ::		18
CHAR	ACTER(LEN=*), INTENT(IN)	:: key	19
	GER, INTENT(INOUT) :: buf		20
	ACTER(LEN=*), INTENT(OUT)	:: value	21 22
LUGICAL, INIENI(UUI) :: IIag			22
			24
<b>Fortran</b>	6		25
		FLEN, VALUE, FLAG, IERROR)	26
	GER INFO, BUFLEN, IERROR ACTER*(*) KEY, VALUE		27
	CAL FLAG		28
			29 30
		ssociated with key in a previous call to	31
MPI_INFO_SET. If such a key exists, it sets flag to true and returns the value in value,			32
		it is the size of the buffer needed to store the value	33
		etion is less than the actual size needed to store the	34
9	-	in C), the value is truncated. On return, the value	35
of buflen $v$	will be set to the required buf	fer size to hold the value string. If buflen is set to	36
		ludes the required space for the null terminator. In	37
		ed string in all cases where the <b>buflen</b> input value is	38 39
greater th			40
11 кеу	is larger than MPI_MAX_INFC	<b>D_KEY</b> , the call is erroneous.	41
Adv	ice to users. The MPI_INFO	_GET_STRING function can be used to obtain the	42
		lue string by setting the <b>buflen</b> to 0. The returned	43
		e memory before calling MPI_INFO_GET_STRING	44
agai	n to obtain the value string. (	End of advice to users.)	45
			$46 \\ 47$
			48

```
1
 MPI_INFO_GET_NKEYS(info, nkeys)
2
 IN
 info
 info object (handle)
3
 OUT
 nkeys
 number of defined keys (integer)
4
5
6
 C binding
7
 int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)
8
 Fortran 2008 binding
9
 MPI_Info_get_nkeys(info, nkeys, ierror)
10
 TYPE(MPI_Info), INTENT(IN) :: info
11
 INTEGER, INTENT(OUT) :: nkeys
12
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
 Fortran binding
15
 MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)
16
 INTEGER INFO, NKEYS, IERROR
17
 MPI_INFO_GET_NKEYS returns the number of currently defined keys in info.
18
19
20
 MPI_INFO_GET_NTHKEY(info, n, key)
21
 info object (handle)
 IN
 info
22
 IN
 key number (integer)
23
 n
^{24}
 OUT
 key
 key (string)
25
26
 C binding
27
 int MPI_Info_get_nthkey(MPI_Info info, int n, char *key)
28
 Fortran 2008 binding
29
 MPI_Info_get_nthkey(info, n, key, ierror)
30
^{31}
 TYPE(MPI_Info), INTENT(IN) :: info
32
 INTEGER, INTENT(IN) :: n
 CHARACTER(LEN=*), INTENT(OUT) :: key
33
34
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
 Fortran binding
36
 MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERROR)
37
 INTEGER INFO, N, IERROR
38
 CHARACTER*(*) KEY
39
40
 This function returns the nth defined key in info. Keys are numbered 0 \dots N-1 where
41
 N is the value returned by MPI_INFO_GET_NKEYS. All keys between 0 and N-1 are
42
 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.
43
44
45
46
47
48
```

MPI_INFC	DUP(info, newinfo)		1 2
IN	info	info object (handle)	3
OUT	newinfo	info object (handle)	4
			5
C bindin	g		6
int MPI_	Info_dup(MPI_Info info, M	IPI_Info *newinfo)	7
Fortran (	2008 binding		8
	_dup(info, newinfo, ierro	or)	9
	(MPI_Info), INTENT(IN) ::		10 11
TYPE	(MPI_Info), INTENT(OUT) :	: newinfo	12
INTE	GER, OPTIONAL, INTENT(OUT	') :: ierror	13
Fortran l	binding		14
	_DUP(INFO, NEWINFO, IERRC	)R)	15
INTE	GER INFO, NEWINFO, IERROF		16
MPI	INFO DUP duplicates an exi	sting info object, creating a new object, with the	17
	value) pairs and the same or		18
	,) Forme on a construction of the		19
			20 21
MPI_INFC	)_FREE(info)		22
INOUT	info	info object (handle)	23
			24
C bindin	0		25
int MPI_	Info_free(MPI_Info *info)		26
Fortran 2	2008 binding		27
MPI_Info	_free(info, ierror)		28
	(MPI_Info), INTENT(INOUT)		29 30
INTE	GER, OPTIONAL, INTENT(OUT	') :: ierror	31
Fortran l	binding		32
MPI_INFO	_FREE(INFO, IERROR)		33
INTE	GER INFO, IERROR		34
This	function frees info and sets it	to MPI INFO NULL.	35
		terpreted each time the info is passed to a routine.	36
		routine do not affect that interpretation.	37
			38 39
	CREATE_ENV(info)		40
			41
OUT	info	info object (handle)	42
<i></i>			43
C bindin	-		44
<pre>int MPI_Info_create_env(int argc, char argv[], MPI_Info *info)</pre>			45
Fortran 2008 binding			46
MPI_Info	_create_env(info, ierror)		47
			48

1 2	TYPE(MPI_Info), INTENT(OUT) :: info INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3	
4	Fortran binding
5	MPI_INFO_CREATE_ENV(INFO, IERROR)
6	INTEGER INFO, IERROR
7	This routine produces an output object info with the same construction as
8	MPI_INFO_ENV as created during MPI_INIT or MPI_INIT_THREAD when the same argu-
9	ments are used. This construction is described in Section 11.2.1; however, this function can
10	be called when not using the World Model, e.g., when using the Sessions Model. This object
11	is not a direct copy or alias of the MPI_INFO_ENV object and could contain different values
12	based on the input arguments and other sources. Multiple calls to this procedure that are
13	given the same input arguments will produce info objects consistent with the definition of
14	$MPI_INFO_ENV.$ The version for ISO C accepts the $argc$ and $argv$ that are provided by the
15	arguments to main or 0 for argc and NULL for argv. The user is responsible for freeing the
16	info object via MPI_INFO_FREE. This procedure is local.
17	This procedure must always be thread-safe, as defined in Section 11.6. It is one of the
18	few routines that may be called before MPI is initialized or after MPI is finalized.
19 20	Advice to users.
20	
22	In some circumstances (e.g., when passing 0 to argc and NULL to argv in C or in Fortran
23	where such arguments do not exist), the info object may not be populated or may be
24	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
25	values in the resulting info object at different MPI processes.
26	
27	(End of advice to users.)
28	
29	
30 31	
32	
33	
34	
35	
36	
37	
38	
39	
40	
41	
42 43	
43 44	
45	
46	
47	
48	

# 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_THREAD and before a call to MPI_FINALIZE. An application can employ both of these Process Models concurrently. In multi-component MPI applications, for example, a component such as a library can make use of the Sessions Model to instantiate MPI resources without impacting the rest of the application.

Both of these models also support the *Dynamic Process Model* (see Section 11.7), which provides for the creation and management of additional processes after an MPI application has been started. A major impetus for the *Dynamic Process Model* comes from the PVM [25] research effort. This work has provided a wealth of experience with process management and resource control that illustrates their benefits and potential pitfalls.

In developing the *Dynamic Process Model*, the MPI Forum decided not to address resource control because it was not able to design a portable interface that would be ap-

 24 

# 484 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

propriate for the broad spectrum of existing and potential resource and process controllers.
 MPI assumes that resource control is provided externally.

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
 applications with parallel modules, and problems that require a run-time assessment of the
 number and type of processes that should be started.

The following goals are central to the design of MPI process management:

- The MPI process model must apply to the vast majority of current parallel environments.
- MPI must not take over operating system responsibilities. It should instead provide a clean interface between an application and system software,
- MPI must guarantee communication determinism in the presence of dynamic processes, i.e., dynamic process management must not introduce unavoidable race conditions.
- MPI must not contain features that compromise performance.
- 18 19 20

21

22

23

27

32

33 34

7

8 9

10 11

12

13 14

15

16

17

The Dynamic Process Model addresses these issues in two ways. First, MPI remains

The *Dynamic Process Model* addresses these issues in two ways. First, MPI remains primarily a communication library. It does not manage the parallel environment in which a parallel program executes, though it provides a minimal interface between an application and external resource and process managers.

Second, MPI maintains a consistent concept of a communicator, regardless of how its members came into existence. A communicator is never changed once created, and it is always created using deterministic collective operations.

- ²⁸ 11.2 The World Model
- ³⁰ 11.2.1 Starting MPI Processes ³¹

When using the World Model, MPI is initialized by calling either MPI_INIT or MPI_INIT_THREAD.

³⁵ MPI_INIT()

```
<sup>37</sup> C binding
```

```
<sup>38</sup> int MPI_Init(int *argc, char ***argv)
```

```
<sup>39</sup>
40 Fortran 2008 binding
```

41 MPI_Init(ierror)

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

```
<sup>43</sup> Fortran binding
```

```
44 MPI_INIT(IERROR)
45 INTEGED IERDOD
```

```
45 INTEGER IERROR
46
```

```
In the World Model, an MPI program must contain exactly one call to an MPI ini-
tialization routine: MPI_INIT_OF MPI_INIT_THREAD. MPI_COMM_WORLD and
```

MPI_COMM_SELF are not valid for use as communicators prior to invocation of MPI_INIT or MPI_INIT_THREAD. Subsequent calls to either of these initialization routines are erroneous. A subset of MPI functions may be invoked before MPI initialization routines are called. See Section 11.4. MPI_INIT accepts the argc and argv that are provided by the arguments to main or NULL:

```
int main(int argc, char *argv[])
{
 MPI_Init(&argc, &argv);
 /* parse arguments */
 /* main program */
 MPI_Finalize(); /* see below */
 return 0;
}
```

The Fortran version takes only IERROR.

Conforming implementations of MPI are required to allow applications to pass NULL for both the argc and argv arguments of main in C.

Failures may disrupt the execution of the program before or during MPI initialization. A high-quality implementation shall not deadlock during MPI initialization, even in the presence of failures. Except for functions with the MPI_T_ prefix, failures in MPI operations prior to or during MPI initialization are reported by invoking the initial error handler. Users can use the "mpi_initial_errhandler" info key during the launch of MPI processes (e.g., MPI_COMM_SPAWN / MPI_COMM_SPAWN_MULTIPLE, or mpiexec) to set a non-fatal initial error handler before MPI initialization. When the initial error handler is set to MPI_ERRORS_ABORT, raising an error before or during initialization aborts the local MPI process (i.e., it is similar to calling MPI_ABORT on MPI_COMM_SELF). An implementation may not always be capable of determining, before MPI initialization, what constitutes the local MPI process, or the set of connected processes. In this case, errors before initialization, the initial error handler is associated with MPI_COMM_WORLD, MPI_COMM_SELF, and the communicator returned by MPI_COMM_GET_PARENT (if any).

Advice to implementors. Some failures may leave MPI in an undefined state, or raise an error before the error handling capabilities are fully operational, in which cases the implementation may be incapable of providing the desired error handling behavior. Of note, in some implementations, the notion of an MPI process is not clearly established in the early stages of MPI initialization (for example, when the implementation considers threads that called MPI_INIT as independent MPI processes); in this case, before MPI is initialized, the MPI_ERRORS_ABORT error handler may abort what would have become multiple MPI processes.

When a failure occurs during MPI initialization, the implementation may decide to return MPI_SUCCESS from the MPI initialization function instead of raising an error. It is recommended that an implementation masks an initialization error only when it expects that later MPI calls will result in well-specified behavior (i.e., barring additional failures, either the outcome of any call will be correct, or the call will raise an 

# Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 24 

# 486 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

appropriate error). For example, it may be difficult for an implementation to avoid unspecified behavior when the group of MPI_COMM_WORLD does not contain the same set of MPI processes at all members of the communicator, or if the communicator returned from MPI_COMM_GET_PARENT was not initialized correctly. (End of advice to implementors.)

After MPI is initialized, the application can access information about the execution 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: 10

- "command" Name of program executed. 12
- 13"argv" Space separated arguments to command. 14
- "maxprocs" Maximum number of MPI processes to start. 15
- 16"mpi_initial_errhandler" Name of the initial errhandler. 17
- 18 "soft" Allowed values for number of processors. 19
- "host" Hostname. 20

1

2

3

4

5

6

7

8

9

11

26

27

2829

30

 31 

32 33

34

35

36 37

38

39

- 21"arch" Architecture name. 22
- 23"wdir" Working directory of the MPI process.  24
- "file" Value is the name of a file in which additional information is specified. 25
  - "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 4041 or, equivalently, with a startup mechanism that supports multiple process specifications, 42then the values stored in the info object MPI_INFO_ENV at a process are those values that affect the local MPI process. 43

```
44
 Example 11.1 If MPI is started with a call to
45
46
 mpiexec -n 5 -arch x86_64 ocean : -n 10 -arch power9 atmos
47
48
```

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 489).
- 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.

### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $\overline{7}$ 

 24 

1 These values are monotonic; i.e., MPI_THREAD_SINGLE < MPI_THREAD_FUNNELED <

 $\mathbf{2}$ MPI_THREAD_SERIALIZED < MPI_THREAD_MULTIPLE.

3 Different processes in MPI_COMM_WORLD may require different levels of thread sup-4 port.

5The call returns in **provided** information about the actual level of thread support that 6 will be provided by MPI. It can be one of the four values listed above.

 $\overline{7}$ The level(s) of thread support that can be provided by MPI_INIT_THREAD will depend 8 on the implementation, and may depend on information provided by the user before the 9 program started to execute (e.g., with arguments to mpiexec). If possible, the call will 10 return provided = required. Failing this, the call will return the least supported level such 11that provided > required (thus providing a stronger level of support than required by the 12user). Finally, if the user requirement cannot be satisfied, then the call will return in 13provided the highest supported level.

14A thread compliant MPI implementation will be able to return provided 15= MPI_THREAD_MULTIPLE. Such an implementation may always return provided 16

= MPI_THREAD_MULTIPLE, irrespective of the value of required.

17An MPI library that is not thread compliant must always return provided =

18 MPI_THREAD_SINGLE, even if MPI_INIT_THREAD is called on a multithreaded process. 19 The library should also return correct values for the MPI calls that can be executed before 20initialization, even if multiple threads have been spawned.

- Such code is erroneous, but if the MPI initialization is performed by a Rationale. library, the error cannot be detected until MPI_INIT_THREAD is called. The requirements in the previous paragraph ensure that the error can be properly detected. (End of rationale.)
- 2526

21

22

23

24

A call to MPI_INIT has the same effect as a call to MPI_INIT_THREAD with a required 27= MPI_THREAD_SINGLE. 28

Vendors may provide (implementation dependent) means to specify the level(s) of 29 thread support available when the MPI program is started, e.g., with arguments to mpiexec. 30 This will affect the outcome of calls to MPI_INIT and MPI_INIT_THREAD. Suppose, for  31 example, that an MPI program has been started so that only MPI_THREAD_MULTIPLE is 32 available. Then MPI_INIT_THREAD will return provided = MPI_THREAD_MULTIPLE, irre-33 spective of the value of required; a call to MPI_INIT will also initialize the MPI thread support 34level to MPI_THREAD_MULTIPLE. Suppose, instead, that an MPI program has been started 35 so that all four levels of thread support are available. Then, a call to MPI_INIT_THREAD 36 will return provided = required; alternatively, a call to  $MPI_INIT$  will initialize the MPI 37 thread support level to MPI_THREAD_SINGLE. 38

39 Various optimizations are possible when MPI code is executed single-Rationale. 40 threaded, or is executed on multiple threads, but not concurrently: mutual exclusion 41 code may be omitted. Furthermore, if only one thread executes, then the MPI library 42can use library functions that are not thread safe, without risking conflicts with user 43 threads. Also, the model of one communication thread, multiple computation threads 44fits many applications well, e.g., if the process code is a sequential Fortran/C program 45with MPI calls that has been parallelized by a compiler for execution on an SMP node, 46in a cluster of SMPs, then the process computation is multithreaded, but MPI calls 47 will likely execute on a single thread. 48

The design accommodates a static specification of the thread support level, for environments that require static binding of libraries, and for compatibility for current multithreaded MPI codes. (*End of rationale.*)

Advice to implementors. If provided is not MPI_THREAD_SINGLE then the MPI library should not invoke C or Fortran library calls that are not thread safe, e.g., in an environment where malloc is not thread safe, then malloc should not be used by the MPI library.

Some implementors may want to use different MPI libraries for different levels of thread support. They can do so using dynamic linking and selecting which library will be linked when MPI_INIT_THREAD is invoked. If this is not possible, then optimizations for lower levels of thread support will occur only when the level of thread support required is specified at link time.

Note that required need not be the same value on all processes of MPI_COMM_WORLD. (*End of advice to implementors.*)

As with MPI_INIT, discussed in Section 11.2.1, the version for ISO C accepts the argc and argv that are provided by the arguments to main or NULL for both arguments.

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

MPI_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) :: provide INTEGER, OPTIONAL, INTENT(OUT) Fortran binding MPI_QUERY_THREAD(PROVIDED, IERROR) INTEGER PROVIDED, IERROR The call returns in provided the currer returned in provided by MPI_INIT_THRE MPI_INIT_THREAD(). This function is initialize MPI. In the case of application	ed :: ierror

 $\mathbf{2}$ 

 $\overline{7}$ 

```
1
 MPI_IS_THREAD_MAIN(flag)
2
 OUT
 flag
 true if calling thread is main thread, false otherwise
3
 (logical)
4
5
 C binding
6
 int MPI_Is_thread_main(int *flag)
7
8
 Fortran 2008 binding
9
 MPI_Is_thread_main(flag, ierror)
10
 LOGICAL, INTENT(OUT) :: flag
11
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
 Fortran binding
13
 MPI_IS_THREAD_MAIN(FLAG, IERROR)
14
 LOGICAL FLAG
15
 INTEGER IERROR
16
17
 This function can be called by a thread to determine if it is the main thread (the thread
18
 that called MPI_INIT or MPI_INIT_THREAD). This function is only applicable when using
19
 the World Model to initialize MPI. In the case of applications using both the World Model
20
 and the Sessions Model, this function only returns the thread support level returned in
21
 provided by MPI_INIT_THREAD.
22
 All routines listed in this section must be supported by all MPI implementations.
23
24
 MPI libraries are required to provide these calls even if they do not
 Rationale.
25
 support threads, so that portable code that contains invocations to these functions
26
 can link correctly. MPI_INIT continues to be supported so as to provide compatibility
27
 with current MPI codes. (End of rationale.)
28
 It is possible to spawn threads before MPI is initialized, but
29
 Advice to users.
 MPI_COMM_WORLD and MPI_COMM_SELF cannot be used until the World Model is
30
 active, i.e. until MPI_INIT_THREAD is invoked by one thread (which, thereby, be-
31
 comes the main thread). In particular, it is possible to enter the MPI execution with
32
33
 a multithreaded process.
34
 In the World Model, the level of thread support provided is a global property of the
35
 MPI process that can be specified only once, when MPI is initialized on that process (or
36
 before). Portable third party libraries have to be written so as to accommodate any
37
 provided level of thread support. Otherwise, their usage will be restricted to specific
38
 level(s) of thread support. If such a library can run only with specific level(s) of thread
39
 support, e.g., only with MPI_THREAD_MULTIPLE, then MPI_QUERY_THREAD can be
40
 used to check whether the user initialized MPI to the correct level of thread support
41
 and. (End of advice to users.)
42
43
 11.2.2 Finalizing MPI
44
45
46
 MPI_FINALIZE()
47
48
```

C binding int MPI_Finalize(void)

Fortran 2008 binding MPI_Finalize(ierror) INTEGER, OPTIONAL, INTENT(OUT) :: ierror

# Fortran binding

MPI_FINALIZE(IERROR) INTEGER IERROR

This routine cleans up all MPI state associated with the World Model. If an MPI program terminates normally (i.e., not due to a call to MPI_ABORT or an unrecoverable error) then each process must call MPI_FINALIZE before it exits.

Before an MPI process invokes MPI_FINALIZE, the process must perform all MPI calls needed to complete its involvement in MPI communications associated with the World Model. It must locally complete all MPI operations that it initiated and must execute matching calls needed to complete MPI communications initiated by other processes. For example, if the process executed a nonblocking send, it must eventually call MPI_WAIT, MPI_TEST, MPI_REQUEST_FREE, or any derived function; if the process is the target of a send, then it must post the matching receive; if it is part of a group executing a collective operation, then it must have completed its participation in the operation.

The call to MPI_FINALIZE does not clean up MPI state associated with objects created using MPI_SESSION_INIT and other Sessions Model methods, nor objects created using the communicator returned by MPI_COMM_GET_PARENT. See Sections 11.3 and 11.8.

The call to MPI_FINALIZE does not free objects created by MPI calls; these objects are freed using MPI_XXX_FREE calls.

MPI_FINALIZE is collective over all connected processes. If no processes were spawned, accepted or connected then this means over MPI_COMM_WORLD; otherwise it is collective over the union of all processes that have been and continue to be connected, as explained in Section 11.10.4.

The following examples illustrate these rules.

**Example 11.2** The following code is correct

Process O	Process 1
<pre>MPI_Init();</pre>	<pre>MPI_Init();</pre>
<pre>MPI_Send(dest=1);</pre>	<pre>MPI_Recv(src=0);</pre>
<pre>MPI_Finalize();</pre>	<pre>MPI_Finalize();</pre>

**Example 11.3** Without a matching receive, the program is erroneous

Process O	Process 1
<pre>MPI_Init(); MPI_Send (dest=1);</pre>	<pre>MPI_Init();</pre>
<pre>MPI_Finalize();</pre>	<pre>MPI_Finalize();</pre>

1 2

3

4

5

6

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

 24 

25

26

27

28

29

30

31 32

Example 11.4 This program is correct: Process 0 calls MPI_Finalize after it has executed
 the MPI calls that complete the send operation. Likewise, process 1 executes the MPI call
 that completes the matching receive operation before it calls MPI_Finalize.

```
Process 0
 Process 1
5

6
 MPI_Init();
 MPI_Init();
7
 MPI_Isend(dest=1);
 MPI_Recv(src=0);
8
 MPI_Request_free();
 MPI_Finalize();
9
 MPI_Finalize();
 exit();
10
 exit();
11
12
13
 Example 11.5 This program is correct. The attached buffer is a resource allocated by
14
 the user, not by MPI; it is available to the user after MPI is finalized.
15
16
 Process 0
 Process 1

17
 MPI_Init();
 MPI_Init();
18
 MPI_Recv(src=0);
 buffer = malloc(1000000);
19
 MPI_Buffer_attach();
 MPI_Finalize();
20
 MPI_Send(dest=1));
 exit();
21
 MPI_Finalize();
22
 free(buffer);
23
 exit();
24
25
26
 This program is correct. The cancel operation must succeed, since the
 Example 11.6
27
 send cannot complete normally. The wait operation, after the call to MPI_Cancel, is local-
28
 no matching MPI call is required on process 1. Cancelling a send request by calling
29
 MPI_CANCEL is deprecated.
30
31
 Process 0
 Process 1
32

33
 MPI_Issend(dest=1);
 MPI_Finalize();
34
 MPI_Cancel();
35
 MPI_Wait();
36
 MPI_Finalize();
37
38
 Advice to implementors. Even though a process has executed all MPI calls needed to
39
 complete the communications it is involved with, such communication may not yet be
40
 completed from the viewpoint of the underlying MPI system. For example, a blocking
41
 send may have returned, even though the data is still buffered at the sender in an MPI
42
 buffer; an MPI process may receive a cancel request for a message it has completed
43
 receiving. The MPI implementation must ensure that a process has completed any
44
 involvement in MPI communication before MPI_FINALIZE returns. Thus, if a process
45
 exits after the call to MPI_FINALIZE, this will not cause an ongoing communication
46
 to fail. The MPI implementation should also complete freeing all objects marked for
47
 deletion by MPI calls that freed them. (End of advice to implementors.)
48
```

. . .

Failures may disrupt MPI operations during and after MPI finalization. A high quality implementation shall not deadlock in MPI finalization, even in the presence of failures. The normal rules for MPI error handling continue to apply. After MPI_COMM_SELF has been "freed" (see Section 11.2.4), errors that are not associated with a communicator, window, or file raise the initial error handler (set during the launch operation, see 11.8.4).

Although it is not required that all processes return from MPI_FINALIZE, it is required that, when it has not failed or aborted, at least the MPI process that was assigned rank 0 in MPI_COMM_WORLD returns, so that users can know that the MPI portion of the computation is over. In addition, in a POSIX environment, users may desire to supply an exit code for each process that returns from MPI_FINALIZE.

Note that a failure may terminate the MPI process that was assigned rank 0 in MPI_COMM_WORLD, in which case it is possible that no MPI process returns from MPI_FINALIZE.

Advice to users. Applications that handle errors are encouraged to implement all rank-specific code before the call to MPI_FINALIZE. In Example 11.7 below, the process with rank 0 in MPI_COMM_WORLD may have been terminated before, during, or after the call to MPI_FINALIZE, possibly leading to the code after MPI_FINALIZE never being executed. (*End of advice to users.*)

**Example 11.7** The following illustrates the use of requiring that at least one process return and that it be known that process 0 is one of the processes that return. One wants code like the following to work no matter how many processes return.

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
...
MPI_Finalize();
if (myrank == 0) {
 resultfile = fopen("outfile", "w");
 dump_results(resultfile);
 fclose(resultfile);
}
exit(0);
```

11.2.3 Determining Whether MPI Has Been Initialized When Using the World Model

One of the goals of MPI is to allow for layered libraries. For a library using the World Model, it needs to know if MPI has been initialized using 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.

 $44 \\ 45$ 

 $\mathbf{2}$ 

 $\overline{7}$ 

```
1
 MPI_INITIALIZED(flag)
2
 OUT
 flag
 Flag is true if MPI_INIT has been called and false
3
 otherwise (logical)
4
5
 C binding
6
 int MPI_Initialized(int *flag)
7
8
 Fortran 2008 binding
9
 MPI_Initialized(flag, ierror)
10
 LOGICAL, INTENT(OUT) :: flag
11
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
 Fortran binding
13
 MPI_INITIALIZED(FLAG, IERROR)
14
 LOGICAL FLAG
15
 INTEGER IERROR
16
17
 This routine may be used to determine whether MPI_INIT or MPI_INIT_THREAD has
18
 been called. MPI_INITIALIZED returns true if the calling process has called either of these
19
 MPI procedures. Whether MPI_FINALIZE has been called does not affect the behavior of
20
 MPI_INITIALIZED. This function must always be thread-safe, as defined in Section 11.6.
21
 This function returns false for applications using the Sessions Model exclusively.
22
23
 MPI_FINALIZED(flag)
^{24}
25
 OUT
 true if MPI was finalized (logical)
 flag
26
27
 C binding
28
 int MPI_Finalized(int *flag)
29
30
 Fortran 2008 binding
^{31}
 MPI_Finalized(flag, ierror)
32
 LOGICAL, INTENT(OUT) :: flag
33
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
 Fortran binding
35
 MPI_FINALIZED(FLAG, IERROR)
36
 LOGICAL FLAG
37
 INTEGER IERROR
38
39
 This routine returns true if MPI_FINALIZE has completed. It is valid to call
40
 MPI_FINALIZED before MPI_INIT and after MPI_FINALIZE. This function must always be
41
 thread-safe, as defined in Section 11.6.
42
43
 11.2.4 Allowing User Functions at MPI Finalization
44
 In the context of the World Model, there are times in which it would be convenient to
45
 have actions happen when an MPI process finalizes MPI. For example, a routine may do
46
 initializations that are useful until the MPI job (or that part of the job that is being termi-
47
 nated in the case of dynamically created processes) finalizes MPI. This can be accomplished
48
```

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.

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 33 an MPI Session handle, which can be used to query the runtime system about character-34istics of the job within which the process is running, as well as other system resources. Using this information, the MPI process can then create an MPI Group based on appli-35 cation requirements and available resources, which in turn can be used to create an MPI 36 37 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 which 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.

46The Sessions Model introduces a concept of isolation. MPI objects derived from differ-47ent MPI Session handles shall not be intermixed with each other in a single MPI procedure 48 call. MPI objects derived from the Sessions Model shall not be intermixed in a single MPI

### **Unofficial Draft for Comment Only**

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6  $\overline{7}$ 

8 9

10

11

12

13

14

15

1617 18

19 20

21

22

23 24 

2526

2728

29

30

 31 

32

38 39

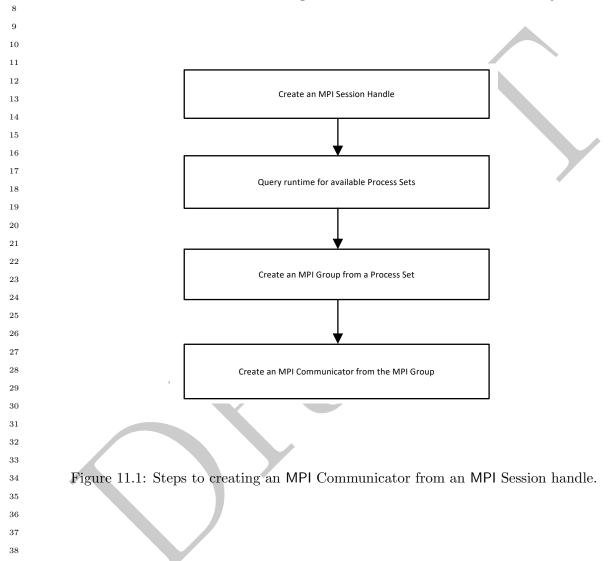
40 41

42

4344

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

# 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.*)

 24 

1 Advice to implementors. Owing to the restrictions of the MPI_THREAD_SINGLE 2 thread support level, implementators are discouraged from making this the default 3 thread support level for Sessions. (End of advice to implementors.) 4 56 MPI_SESSION_FINALIZE(session) 7 IN session session to be finalized (handle) 8 9 10 C binding 11int MPI_Session_finalize(MPI_Session *session) 12Fortran 2008 binding 13 MPI_Session_finalize(session, ierror) 14TYPE(MPI_Session), INTENT(INOUT) :: session 15INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1617Fortran binding 18 MPI_SESSION_FINALIZE(SESSION, IERROR) 19INTEGER SESSION, IERROR 20This routine cleans up all MPI state associated with the supplied session. Every instantiated 21Session must be finalized using MPI_SESSION_FINALIZE. The handle session is set to 22 MPI_SESSION_NULL by the call. 23Before an MPI process invokes MPI_SESSION_FINALIZE, the process must perform  24 all MPI calls needed to complete its involvement in MPI communications: it must locally 25complete all MPI operations that it initiated and it must execute matching calls needed to 26complete MPI communications initiated by other processes. 27The call to MPI_SESSION_FINALIZE does not free objects created by MPI calls; these 28 objects are freed using MPI_XXX_FREE calls. 29 MPI_SESSION_FINALIZE is collective over all MPI processes that are connected via 30 MPI Communicators, Windows, or Files that were created as part of the Session and still  31 exist. If processes were spawned, accepted, or connected using MPI Communicators created 32 as part of this session, this operation is collective over the union of all processes that have 33 been and continue to be connected via those objects, as explained in Section 11.10.4. 34 35 An MPI implementation should be able to implement Advice to implementors. 36 the semantics of MPI_SESSION_FINALIZE without synchronization with other MPI 37 processes, provided an application frees all MPI windows, closes all MPI files, and uses 38 MPI_COMM_DISCONNECT to free all MPI communicators associated with a session 39 prior to invoking MPI_SESSION_FINALIZE on the corresponding session handle. (End 40 of advice to implementors.) 41 4211.3.2 Processes Sets 43 44Process sets are the mechanism for MPI applications to query the runtime. Process sets are 45identified by process set names. Process set names have a Uniform Resource Identifier (URI) 46format. Two process set names are mandated: "mpi://WORLD" and 47"mpi://SELF". Additional process set names may be defined, for example, 48 "mpix://UNIVERSE" and "hwloc://L3Cache" may be defined by the MPI implementation. The

 31 

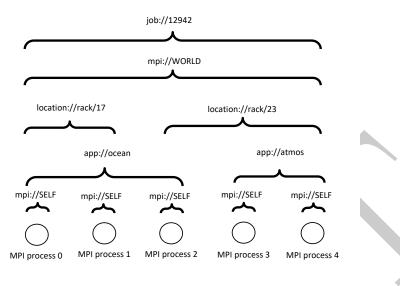


Figure 11.2: Examples of process sets. Illustrated are the two mandated process sets - "mpi://WORLD" and "mpi://SELF" - along with several optional ones that a runtime could define. In this example, MPI_SESSION_GET_NUM_PSETS would return five at each MPI process.

"mpi://" namespace is reserved for exclusive use by the MPI standard. Figure 11.2 depicts process sets that the runtime could associate with an instance of an MPI job. In this example, the two mandated process sets are defined, in addition to optional, implementation specific ones.

Mechanisms for defining process sets and how system resources are assigned to these sets is considered to be implementation dependent.

A process set caches key/value tuples that are accessible to the application via an MPI_Info object. The "mpi_size" key is mandatory for all process sets.

# 11.3.3 Runtime Query Functions

MPI_SESSION_GET_NUM_PSETS(session, info, npset_names)		38	
		39	
IN	session	session (handle)	40
IN	info	info object (handle)	41
OUT	npset_names	number of available process sets (non-negative	42
001	npset_names	integer)	43
			44
C binding int MPI_Session_get_num_psets(MPI_Session session, MPI_Info info, int *npset_names)		46	
		47	
		48	

```
1
 Fortran 2008 binding
\mathbf{2}
 MPI_Session_get_num_psets(session, info, npset_names, ierror)
3
 TYPE(MPI_Session), INTENT(IN) :: session
4
 TYPE(MPI_Info), INTENT(IN) :: info
5
 INTEGER, INTENT(OUT) :: npset_names
6
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
 Fortran binding
8
 MPI_SESSION_GET_NUM_PSETS(SESSION, INFO, NPSET_NAMES, IERROR)
9
 INTEGER SESSION, INFO, NPSET_NAMES, IERROR
10
11
 This function is used to query the runtime for the number of available process sets in
12
 which the calling MPI process is a member. An MPI implementation is allowed to increase
13
 the number of available process sets during the execution of an MPI application when new
14
 process sets become available. However, MPI implementations are not allowed to change
15
 the index of a particular process set name, or to change the name of the process set at a
16
 particular index, or to delete a process set name once it has been added. When a process
17
 set becomes invalid, for example, when some processes become unreachable due to failures
18
 in the communication system, subsequent usage of the process set name should raise an
19
 error. For example, creating an MPI_Group from such a process set might succeed because it
20
 is a local operation, but creating an MPI_Comm from that group and attempting collective
21
 communication should raise an error.
22
 Advice to implementation. It is anticipated that an MPI implementation may be re-
23
24
 lying on an external runtime system to provide process sets. Such runtime systems
25
 may have the ability to dynamically create process sets during the course of appli-
26
 cation execution. Requiring the number of process sets returned by
 MPI_SESSION_GET_NUM_PSETS to be constant over the course of application exe-
27
 cution would prevent an application from taking advantage of such capabilities. (End
28
 of advice to implementors.)
29
30
31
32
 MPI_SESSION_GET_NTH_PSET(session, info, n, pset_len, pset_name)
33
34
 IN
 session
 session (handle)
35
 IN
 info
 info object (handle)
36
 IN
 index of the desired process set name (integer)
 n
37
 INOUT
 length of the pset_name argument (integer)
38
 pset_len
39
 OUT
 pset_name
 name of the nth process set (string)
40
41
 C binding
42
 int MPI_Session_get_nth_pset(MPI_Session session, MPI_Info info, int n,
43
 int *pset_len, char *pset_name)
44
 Fortran 2008 binding
45
46
 MPI_Session_get_nth_pset(session, info, n, pset_len, pset_name, ierror)
47
 TYPE(MPI_Session), INTENT(IN) :: session
48
 TYPE(MPI_Info), INTENT(IN) :: info
```

```
1
 INTEGER, INTENT(IN) :: n
 2
 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
 9
 10
 This function returns the name of the nth process set in the supplied pset_name buffer.
 11
pset_len is the size of the buffer needed to store the nth process set name. If the pset_len
passed into the function is less than the actual buffer size needed for the process set name,
 12
 13
then the string value returned in pset_name is truncated. If pset_len is set to 0, pset_name is
 14
not changed. On return, the value of pset_len will be set to the required buffer size to hold
 15
the process set name. In C, pset_len includes the required space for the null terminator. In
 16
C, this function returns a null terminated string in all cases where the pset_len input value
 17
is greater than 0.
 18
 If two MPI processes get the same process set name, then the intersection of the two
 19
process sets shall either be the empty set or identical to the union of the two process sets.
 20
 After a successful call to MPI_SESSION_GET_NTH_PSET, subsequent calls to routines
 21
that query information about the same process set name and same session handle must
 22
return the same information. An MPI implementation is not allowed to alter any of the
 23
returned process set names.
 24
 Process set names have an implementation-defined maximum length of
 25
MPI_MAX_PSET_NAME_LEN.
 26
 Advice to users. MPI_MAX_PSET_NAME_LEN might be very large, so it might not
 27
 be wise to declare a string of that size. Users are encouraged to use
 28
 MPI_SESSION_GET_NTH_PSET both for obtaining the length of a pset_name and
 29
 30
 the process set name. (End of advice to users.)
 31
 32
 33
MPI_SESSION_GET_INFO(session, info_used)
 34
 IN
 session
 session (handle)
 35
 36
 OUT
 info_used
 see explanation below (handle)
 37
 38
C binding
 39
int MPI_Session_get_info(MPI_Session session, MPI_Info *info_used)
 40
 41
Fortran 2008 binding
 42
MPI_Session_get_info(session, info_used, ierror)
 TYPE(MPI_Session), INTENT(IN) :: session
 43
 TYPE(MPI_Info), INTENT(OUT) :: info_used
 44
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 45
 46
Fortran binding
 47
MPI_SESSION_GET_INFO(SESSION, INFO_USED, IERROR)
 48
```

```
1
 INTEGER SESSION, INFO_USED, IERROR
\mathbf{2}
 MPI_SESSION_GET_INFO returns a new info object containing the hints of the MPI
3
 Session associated with session. The current setting of all hints related to this MPI Session
4
 is returned in info_used. An MPI implementation is required to return all hints that are
5
 supported by the implementation and have default values specified; any user-supplied hints
6
 that were not ignored by the implementation; and any additional hints that were set by
7
 the implementation. If no such hints exist, a handle to a newly created info object is
8
 returned that contains no key/value pair. The user is responsible for freeing info_used via
9
 MPI_INFO_FREE.
10
11
12
 MPI_SESSION_GET_PSET_INFO(session, pset_name, info)
13
 IN
 session
 session (handle)
14
 IN
 name of process set (string)
15
 pset_name
16
 OUT
 info object containing information about the given
 info
17
 process set (handle)
18
19
 C binding
20
 int MPI_Session_get_pset_info(MPI_Session session, const char *pset_name,
21
 MPI_Info *info)
22
23
 Fortran 2008 binding
24
 MPI_Session_get_pset_info(session, pset_name, info, ierror)
25
 TYPE(MPI_Session), INTENT(IN) :: session
26
 CHARACTER(LEN=*), INTENT(IN) :: pset_name
 TYPE(MPI_Info), INTENT(OUT) :: info
27
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
 Fortran binding
30
 MPI_SESSION_GET_PSET_INFO(SESSION, PSET_NAME, INFO, IERROR)
^{31}
 INTEGER SESSION, INFO, IERROR
32
 CHARACTER*(*) PSET_NAME
33
34
 This function is used to query properties of a specific process set. The returned info
35
 object can be queried with existing MPI info object query functions. One key/value pair
36
 must be defined, "mpi_size". The value of the "mpi_size" key specifies the number of MPI
37
 processes in the process set. The user is responsible for freeing the returned MPI_Info object.
38
39
 11.3.4 Sessions Model Examples
40
 This section presents several examples of how to use MPI Sessions to create MPI Groups
41
 and MPI Communicators.
42
43
 Example 11.8 Simple example illustrating creation of an MPI communicator using the
44
 Sessions Model.
45
46
47
48
```

```
1
#include <stdio.h>
 \mathbf{2}
#include <stdlib.h>
 3
#include <string.h>
#include "mpi.h"
 4
 5
 6
static MPI_Session lib_shandle = MPI_SESSION_NULL;
static MPI_Comm lib_comm = MPI_COMM_NULL;
 7
 9
int library_foo_init(void)
 10
{
 11
 int rc, flag, valuelen;
 int ret = 0;
 12
 const char pset_name[] = "mpi://WORLD";
 13
 14
 const char mt_key[] = "thread_level";
 15
 const char mt_value[] = "MPI_THREAD_MULTIPLE";
 16
 char out_value[100];
 /* large enough */
 17
 MPI_Group wgroup = MPI_GROUP_NULL;
 18
 MPI_Info sinfo = MPI_INFO_NULL;
 19
 MPI_Info tinfo = MPI_INFO_NULL;
 20
 21
 MPI_Info_create(&sinfo);
 MPI_Info_set(sinfo, mt_key, mt_value);
 22
 rc = MPI_Session_init(sinfo, MPI_ERRORS_RETURN,
 23
 ^{24}
 &lib_shandle);
 25
 if (rc != MPI_SUCCESS) {
 26
 ret = -1;
 27
 goto fn_exit;
 }
 28
 29
 30
 /*
 * check we got thread support level foo library needs
 31
 32
 */
 33
 rc = MPI_Session_get_info(lib_shandle, &tinfo);
 34
 if (rc != MPI_SUCCESS) {
 ret = -1;
 35
 36
 goto fn_exit;
 37
 }
 38
 39
 valuelen = sizeof(out_value);
 MPI_Info_get_string(tinfo, mt_key, &valuelen,
 40
 41
 out_value, &flag);
 42
 if (0 == flag) {
 printf("Could not find key %s\n", mt_key);
 43
 44
 ret = -1;
 45
 goto fn_exit;
 }
 46
 47
 48
 if (strcmp(out_value, mt_value)) {
```

```
1
 printf("Did not get thread multiple support, got %s\n",
\mathbf{2}
 out_value);
3
 ret = -1;
4
 goto fn_exit;
5
 }
6
7
 /*
8
 * create a group from the WORLD process set
9
 */
10
 rc = MPI_Group_from_session_pset(lib_shandle,
11
 pset_name,
12
 &wgroup);
13
 if (rc != MPI_SUCCESS) {
14
 ret = -1;
15
 goto fn_exit;
16
 }
17
18
 /*
19
 * get a communicator
20
 */
21
 rc = MPI_Comm_create_from_group(wgroup,
22
 "org.mpi-forum.mpi-v4_0.example-ex10_8",
23
 MPI_INFO_NULL,
24
 MPI_ERRORS_RETURN,
25
 &lib_comm);
26
 if (rc != MPI_SUCCESS) {
27
 ret = -1;
28
 goto fn_exit;
29
 }
30
^{31}
 /*
32
 * free group, library doesn't need it.
33
 */
34
35
 fn_exit:
 MPI_Group_free(&wgroup);
36
37
38
 if (sinfo != MPI_INFO_NULL) {
39
 MPI_Info_free(&sinfo);
40
 }
41
42
 if (tinfo != MPI_INFO_NULL) {
43
 MPI_Info_free(&tinfo);
44
 }
45
46
 if (ret != 0) {
47
 MPI_Session_finalize(&lib_shandle);
48
 }
```

```
return ret;
```

}

Example 11.8 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 11.9** 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"
int main(int argc, char *argv[])
{
 int i, n_psets, psetlen, rc, ret;
 int valuelen;
 int flag = 0;
 21
 char *pset_name = NULL;
 22
 char *info_val = NULL;
 23
 MPI_Session shandle = MPI_SESSION_NULL;
 MPI_Info sinfo = MPI_INFO_NULL;
 MPI_Group pgroup = MPI_GROUP_NULL;
 27
 if (argc < 2) {
 fprintf(stderr, "A process set name fragment is required\n");
 28
 29
 return -1;
 30
 }
 32
 rc = MPI_Session_init(MPI_INFO_NULL, MPI_ERRORS_RETURN, &shandle);
 33
 if (rc != MPI_SUCCESS) {
 34
 fprintf(stderr, "Could not initialize session, bailing out\n");
 35
 return -1;
 }
 36
 37
 MPI_Session_get_num_psets(shandle, MPI_INFO_NULL, &n_psets);
 for (i=0, pset_name=NULL; i<n_psets; i++) {</pre>
 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;
```

1  $\mathbf{2}$ 

3

4

5

6

7 8

9

10 11

1213

14

1516

17

18

19

20

 24 

25

26

31

38

39 40

41

42

43 44

45

46

47

```
1
 free(pset_name);
\mathbf{2}
 pset_name = NULL;
3
 }
4
5
 /*
6
 * get instance of an info object for this Session
7
 */
8
9
 MPI_Session_get_pset_info(shandle, pset_name, &sinfo);
10
 valuelen = 0;
11
 MPI_Info_get_string(sinfo, "mpi_size", &valuelen, NULL, &flag);
12
 if (flag) {
13
 info_val = (char *)malloc(valuelen);
14
 MPI_Info_get_string(sinfo, "mpi_size", &valuelen, info_val, &flag);
15
 free(info_val);
16
 }
17
18
 /*
19
 * create a group from the process set
20
 */
21
22
 rc = MPI_Group_from_session_pset(shandle, pset_name,
23
 &pgroup);
24
 ret = (rc == MPI_SUCCESS) ? 0 : -1;
25
26
 free(pset_name);
27
 MPI_Group_free(&pgroup);
28
 MPI_Info_free(&sinfo);
29
 MPI_Session_finalize(&shandle);
30
31
 fprintf(stderr, "Test completed ret = %d\n", ret);
32
 return ret;
33
34
 }
35
 Example 11.9 illustrates several aspects of the Sessions Model. First, the default error
36
```

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 11.10 A Fortran 2008 example illustrating how to obtain information about
 available process sets, create an MPI Group from a process set, and subsequently create an
 MPI Communicator.

```
 PROGRAM MAIN
 USE mpi_f08
 IMPLICIT NONE
 INTEGER :: pset_len, ierror, n_psets
 CHARACTER(LEN=:), ALLOCATABLE :: pset_name
```

37

38

39

```
1
 TYPE(MPI_Session) :: shandle
 \mathbf{2}
 TYPE(MPI_Group) :: pgroup
 TYPE(MPI_Comm) :: pcomm
 CALL MPI_Session_init(MPI_INFO_NULL, MPI_ERRORS_RETURN, &
 5
 6
 shandle, ierror)
 IF (ierror .NE. MPI_SUCCESS) THEN
 WRITE(*,*) "MPI_Session_init failed"
 9
 ERROR STOP
 10
 END IF
 11
 CALL MPI_Session_get_num_psets(shandle, MPI_INFO_NULL, n_psets)
 12
 13
 IF (n_psets .LT. 2) THEN
 14
 WRITE(*,*) "MPI_Session_get_num_psets didn't return at least 2 psets"
 15
 ERROR STOP
 16
 END IF
 17
 18
!
 19
!
 Just get the second pset's length and name
!
 Note that index values are zero-based, even in Fortran
 20
 21
!
 22
 23
 pset_len = 0
 CALL MPI_Session_get_nth_pset(shandle, MPI_INFO_NULL, 1,
 ^{24}
 &
 25
 pset_len, pset_name)
 26
 ALLOCATE(CHARACTER(LEN=pset_len)::pset_name)
 CALL MPI_Session_get_nth_pset(shandle, MPI_INFO_NULL, 1,
 27
 &
 28
 pset_len, pset_name)
 29
 30
!
 31
!
 create a group from the pset
 32
!
 33
 CALL MPI_Group_from_session_pset(shandle, pset_name, pgroup)
 34
ļ
!
 free the buffer used for the pset name
 35
 36
!
 37
 DEALLOCATE(pset_name)
 38
 39
!
 40
!
 create a MPI communicator from the group
 41
i
 42
 CALL MPI_Comm_create_from_group(pgroup, "session_example",
 &
 MPI_INFO_NULL,
 &
 43
 44
 MPI_ERRORS_RETURN,
 X.
 45
 pcomm)
 46
 47
 CALL MPI_Barrier(pcomm, ierror)
 48
 IF (ierror .NE. MPI_SUCCESS) THEN
```

### 508 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

```
1
 WRITE(*,*) "Barrier call on communicator failed"
2
 ERROR STOP
3
 END IF
4
5
 CALL MPI_Comm_free(pcomm)
6
 CALL MPI_Group_free(pgroup)
7
 CALL MPI_Session_finalize(shandle, ierror)
8
9
 END PROGRAM MAIN
10
11
 Note in this example that the call to MPI_SESSION_FINALIZE may block in order
12
 to ensure that the calling MPI process has completed its involvement in the preceding
13
 MPI_BARRIER operation. If MPI_COMM_DISCONNECT had been used instead of
14
 MPI_COMM_FREE, the example would have blocked in MPI_COMM_DISCONNECT rather
15
 than MPI_SESSION_FINALIZE.
16
17
18
 Common Elements of Both Process Models
 11.4
19
20
 11.4.1
 MPI Functionality that is Always Available
21
 Some MPI functions may be invoked at any time, including prior to calling MPI_INIT or
22
 MPI_SESSION_INIT, and following MPI finalization, independent of whether the World
23
 Model, Sessions Model, or both are used. These functions can be called concurrently by
24
 multiple threads within an MPI Process. Table 11.1 lists the applicable MPI functions.
25
 In addition to the functions listed in Table 11.1, any function with the prefix MPI_T_
26
 (within the constraints for functions with this prefix listed in Section 15.3.4) may also be
27
 called prior to MPI initialization and after MPI finalization.
28
29
 11.4.2 Aborting MPI Processes
30
^{31}
32
 MPI_ABORT(comm, errorcode)
33
34
 IN
 comm
 communicator of tasks to abort (handle)
35
 IN
 errorcode
 error code to return to invoking environment
36
 (integer)
37
38
39
 C binding
 int MPI_Abort(MPI_Comm comm, int errorcode)
40
41
 Fortran 2008 binding
42
 MPI_Abort(comm, errorcode, ierror)
43
 TYPE(MPI_Comm), INTENT(IN) :: comm
44
 INTEGER, INTENT(IN) :: errorcode
45
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
 Fortran binding
 MPI_ABORT(COMM, ERRORCODE, IERROR)
48
```

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
MPI_INFO_GET_VALUELEN
MPI_INFO_GET_NKEYS
MPI_INFO_GET_NTHKEY
MPI_INFO_DUP
MPI_INFO_FREE
MPI_INFO_F2C
MPI_INFO_C2F
MPI_SESSION_CREATE_ERRHANDLER
MPI_SESSION_CALL_ERRHANDLER
MPI_ERRHANDLER_FREE
MPI_ERRHANDLER_F2C
MPI_ERRHANDLER_C2F
MPI_ERROR_STRING
MPI_ERROR_CLASS

Table 11.1: List of MPI Functions that can be called at any time within an MPI program, including prior to MPI initialization and following MPI finalization

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 non-aborted processes. As an example, if the user changes the error handler for MPI_COMM_WORLD to MPI_ERRORS_RETURN or a

### Unofficial Draft for Comment Only

# 510 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

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.)

10 11 12

13

14

15 16 17

18 19

20

1

2

3

4

5

6

7

8

9

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

21 22

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

39 40 41

46

47

48

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 the 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	<maxp< th=""><th>procs&gt;</th></maxp<>	procs>
-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:

As with MPI_COMM_SPAWN, all the arguments are optional. (Even the  $-n \ge argument$  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  $\langle \texttt{filename} \rangle$  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 11.11** Start 16 instances of myprog on the current or default machine:

mpiexec -n 16 myprog

**Example 11.12** Start 10 instances of myprog on the machine called ferrari:

mpiexec -n 10 -host ferrari myprog

### Unofficial Draft for Comment Only

 31 

1	Example 11.13 Start 3 instances of the same program myprog with different com-
2	mand-line arguments:
3	
4	<pre>mpiexec myprog infile1 : myprog infile2 : myprog infile3</pre>
5	
6	
7	<b>Example 11.14</b> Start 5 instances of the ocean program on x86_64 hosts and 10
8	instances of the atmos program on Power9 hosts (Form B):
9	
10	mpiexec -n 5 -arch x86_64 ocean : -n 10 -arch power9 atmos
11	
12	It is assumed that the implementation in this case has a method for choosing hosts of
13	the appropriate type. Their ranks are in the order specified.
14	
15	Example 11.15 Start the ocean program on five Suns and the atmos program on
16	10 RS/6000's (Form B):
17	
18	mpiexec -configfile myfile
19	
20	where myfile contains
21	
22	-n 5 -arch sun ocean
23	-n 10 -arch rs6000 atmos
24	
25	(End of advice to implementors.)
26	
27	

#### 11.6 MPI and Threads

This section specifies the interaction between MPI calls and threads. Although thread com-pliance is not required, the standard specifies how threads are to work if they are provided.  31 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 im-plemented 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 Ta-ble 11.1. When a thread is executing one of these routines, if another concurrently running thread also makes an MPI call, the outcome will be as if the calls executed in some order. 

This section generally assumes a thread package similar to POSIX threads [44], but the syntax and semantics of thread calls are not specified here—these are beyond the scope of this document.

11.6.1 General

In a thread-compliant implementation, an MPI process is a process that may be multi-threaded. Each thread can issue MPI calls; however, threads are not separately addressable: a rank in a send or receive call identifies a process, not a thread. A message sent to a process can be received by any thread in this process. 

This model corresponds to the POSIX model of interprocess commu-Rationale. nication: the fact that a process is multithreaded, rather than single-threaded, does not affect the external interface of this process. MPI implementations in which MPI 'processes' are POSIX threads inside a single POSIX process are not thread-compliant by this definition (indeed, their "processes" are single-threaded). (End of rationale.)

Advice to users. It is the user's responsibility to prevent races when threads within the same application post conflicting communication calls. The user can make sure that two threads in the same process will not issue conflicting communication calls by using distinct communicators at each thread. (End of advice to users.)

The two main requirements for a thread-compliant implementation are listed below.

- 1. All MPI calls are *thread-safe*, i.e., two concurrently running threads may make MPI calls and the outcome will be as if the calls executed in some order, even if their execution is interleaved.
- 2. Blocking MPI calls will block the calling thread only, allowing another thread to execute, if available. The calling thread will be blocked until the event on which it is waiting occurs. Once the blocked communication is enabled and can proceed, then the call will complete and the thread will be marked runnable, within a finite time. A blocked thread will not prevent progress of other runnable threads on the same process, and will not prevent them from executing MPI calls.

**Example 11.16** Process 0 consists of two threads. The first thread executes a blocking send call MPI_Send(buff1, count, type, 0, 0, comm), whereas the second thread executes a blocking receive call MPI_Recv(buff2, count, type, 0, 0, comm, &status), i.e., the first 27thread sends a message that is received by the second thread. This communication should always succeed. According to the first requirement, the execution will correspond to some 29interleaving of the two calls. According to the second requirement, a call can only block 30 the calling thread and cannot prevent progress of the other thread. If the send call went ahead of the receive call, then the sending thread may block, but this will not prevent the receiving thread from executing. Thus, the receive call will occur. Once both calls occur, the communication is enabled and both calls will complete. On the other hand, a 34 single-threaded process that posts a send, followed by a matching receive, may deadlock. 35 The progress requirement for multithreaded implementations is stronger, as a blocked call 36 cannot prevent progress in other threads. 37

Advice to implementors. MPI calls can be made thread-safe by executing only one at a time, e.g., by protecting MPI code with one process-global lock. However, blocked operations cannot hold the lock, as this would prevent progress of other threads in the process. The lock is held only for the duration of an atomic, locally-completing suboperation such as posting a send or completing a send, and is released in between. Finer locks can provide more concurrency, at the expense of higher locking overheads. Concurrency can also be achieved by having some of the MPI protocol executed by separate server threads. (End of advice to implementors.)

4647

48

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 6  $\overline{7}$ 

8

9

10 11

1213

14

15

1617

18

19

20

21

22

23 24 

25

26

28

31

32

33

38

39

40

41

42

43

44

# 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.

6 7 8

1

 $\mathbf{2}$ 

3

4

5

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

9 Threads and the Sessions Model The Sessions Model provides a finer-grain approach to 10 controlling the interaction between MPI calls and threads. When using this model, the 11desired level of thread support is specified at Session initialization time. See Section 11.3. 12Thus it is possible for communicators and other MPI objects derived from one Session 13 to provide a different level of thread support than those created from another Session 14for which a different level of thread support was requested. Depending on the level of 15thread support requested at Session initialization time, different threads in a MPI process 16can make concurrent calls to MPI when using MPI objects derived from different session 17handles. Note that the requested and provided level of thread support when creating a 18 Session may influence the granted level of thread support in a subsequent invocation of 19MPI_SESSION_INIT. Likewise, if the application at some point calls

²⁰ MPI_INIT_THREAD, the requested and granted level of thread support may influence the ²¹ granted level of thread support for subsequent calls to MPI_SESSION_INIT. Similarly, if the ²² application calls MPI_INIT_THREAD after a call to MPI_SESSION_INIT, the level of thread ²³ support returned from MPI_INIT_THREAD may be similarly influenced by the requested ²⁴ 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.*)

44 45 46

47

48

37

38

39

40

41

42

43

Probe A receive call that uses source and tag values returned by a preceding call toMPI_PROBE or MPI_IPROBE will receive the message matched by the probe call only

if there was no other matching receive after the probe and before that receive. In a multithreaded environment, it is up to the user to enforce this condition using suitable mutual exclusion logic. This can be enforced by making sure that each communicator is used by only one thread on each process. Alternatively, MPI_MPROBE or MPI_IMPROBE can be used.

Collective calls Matching of collective calls on a communicator, window, or file handle is done according to the order in which the calls are issued at each process. If concurrent threads issue such calls on the same communicator, window or file handle, it is up to the user to make sure the calls are correctly ordered, using interthread synchronization.

Advice to users. With three concurrent threads in each MPI process of a communicator comm, it is allowed that thread A in each MPI process calls a collective operation on comm, thread B calls a file operation on an existing filehandle that was formerly opened on comm, and thread C invokes one-sided operations on an existing window handle that was also formerly created on comm. (*End of advice to users.*)

*Rationale.* As specified in MPI_FILE_OPEN and MPI_WIN_CREATE, a file handle and a window handle inherit only the group of processes of the underlying communicator, but not the communicator itself. Accesses to communicators, window handles and file handles cannot affect one another. (*End of rationale.*)

Advice to implementors. If the implementation of file or window operations internally uses MPI communication then a duplicated communicator may be cached on the file or window object. (*End of advice to implementors.*)

**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-cancelsafe" or "async-signal-safe"). (*End of rationale.*)

Advice to users. Users can catch signals in separate, non-MPI threads (e.g., by ⁴⁵ masking signals on MPI calling threads, and unmasking them in one or more non-MPI ⁴⁶ threads). A good programming practice is to have a distinct thread blocked in a ⁴⁷ call to sigwait for each user expected signal that may occur. Users must not catch ⁴⁸

# Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

5

6 7

8

9

10

11

12

13

14

15

16 17

18

19

20

21 22

23

 24 

25 26

27

28

29 30

 31 

32

33

34 35

36

37

38

39 40

41

42

43

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

¹⁰ The dynamic process model allows for the creation and cooperative termination of processes ¹¹ after an MPI application has started. It provides a mechanism to establish communication ¹² between the newly created processes and the existing MPI application. It also provides a ¹³ mechanism to establish communication between two existing MPI applications, even when ¹⁴ one did not "start" the other.

15 16

17

26

28

1

2

3

4

5

6 7 8

9

## 11.7.1 Starting Processes

¹⁸ MPI applications may start new processes through an interface to an external process man ¹⁹ ager.

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.

²⁴ MPI uses the group abstraction to represent processes. A process is identified by a ²⁵ (group, rank) pair.

# ²⁷ 11.7.2 The Runtime Environment

The MPI_COMM_SPAWN and MPI_COMM_SPAWN_MULTIPLE routines provide an interface between MPI and the *runtime environment* of an MPI application. The difficulty is that there is an enormous range of runtime environments and application requirements, and MPI must not be tailored to any particular one.

MPI assumes, implicitly, the existence of an environment in which an application runs. It does not provide "operating system" services, such as a general ability to query what processes are running, to kill arbitrary processes, to find out properties of the runtime environment (how many processors, how much memory, etc.). Complex interaction of an MPI application with its runtime environment should be done through an environmentspecific API.

At some low level, obviously, MPI must be able to interact with the runtime system, but the interaction is not visible at the application level and the details of the interaction are not specified by the MPI standard.

In many cases, it is impossible to keep environment-specific information out of the MPI interface without seriously compromising MPI functionality. To permit applications to take advantage of environment-specific functionality, many MPI routines take an info argument that allows an application to specify environment-specific information. There is a tradeoff between functionality and portability: applications that make use of environment-specific info are not portable.

MPI does not require the existence of an underlying "virtual machine" model, in which there is a consistent global view of an MPI application and an implicit "operating system" managing resources and processes. For instance, processes spawned by one task may not be visible to another; additional hosts added to the runtime environment by one process may not be visible in another process; tasks spawned by different processes may not be automatically distributed over available resources.

Interaction between MPI and the runtime environment is limited to the following areas:

- A process may start new processes with MPI_COMM_SPAWN and MPI_COMM_SPAWN_MULTIPLE.
- When a process spawns a child process, it may optionally use an info argument to tell the runtime environment where or how to start the process. This extra information may be opaque to MPI.
- An attribute MPI_UNIVERSE_SIZE (See Section 11.10.1) on MPI_COMM_WORLD tells a program how "large" the initial runtime environment is, namely how many processes can usefully be started in all. One can subtract the size of MPI_COMM_WORLD from this value to find out how many processes might usefully be started in addition to those already running.

# 11.8 Process Manager Interface

## 11.8.1 Processes in MPI

A process is represented in MPI by a (group, rank) pair. A (group, rank) pair specifies a unique process but a process does not determine a unique (group, rank) pair, since a process may belong to several groups.

# 11.8.2 Starting Processes and Establishing Communication

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

Advice to users. It is possible in MPI to start an SPMD or MPMD application with a fixed number of processes after initialization by first starting one process and having that process start its siblings with MPI_COMM_SPAWN. This practice is discouraged primarily for reasons of performance. If possible, it is preferable to start all processes at once, as a single MPI application. (*End of advice to users.*)

 31 

1 2	MPI_COMM_SPAWN(command, argv, maxprocs, info, root, comm, intercomm, array_of_errcodes)		
3 4 5	IN	command	name of program to be spawned (string, significant only at root)
6 7	IN	argv	arguments to <b>command</b> (array of strings, significant only at root)
8 9	IN	maxprocs	maximum number of processes to start (integer, significant only at root)
10 11 12 13	IN	info	a set of key-value pairs telling the runtime system where and how to start the processes (handle, significant only at root)
14 15	IN	root	rank of process in which previous arguments are examined (integer)
16 17	IN	comm	intra-communicator containing group of spawning processes (handle)
18 19 20	OUT	intercomm	inter-communicator between original group and the newly spawned group (handle)
21 22	OUT	array_of_errcodes	one code per process (array of integer)
23 24 25 26 27 28 29 30 31 32 33 34 35 36	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(IN) :: intercomm		
37 38	INTEGER, OPTIONAL, INTENT(OUT) :: ierror		
<ul> <li>39</li> <li>40</li> <li>41</li> <li>42</li> <li>43</li> </ul>	<ul> <li>MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES, IERROR)</li> <li>CHARACTER*(*) COMMAND, ARGV(*)</li> <li>INTEGER MAXPROCS, INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*),</li> </ul>		
44 45 46 47 48	⁴⁵ MPI_COMM_SPAWN tries to start maxprocs identical copies of the MPI program spec- ⁴⁶ ified by command, establishing communication with them and returning an inter-commu- ⁴⁷ nicator. The spawned processes are referred to as children. The children have their own MPI_COMM_WORLD, which is separate from that of the parents_MPI_COMM_SPAWN is		

collective over comm, and also may not return until MPI_INIT has been called in the children. Similarly, MPI_INIT in the children may not return until all parents have called MPI_COMM_SPAWN. In this sense, MPI_COMM_SPAWN in the parents and MPI_INIT in the children form a collective operation over the union of parent and child processes. The inter-communicator returned by MPI_COMM_SPAWN contains the parent processes in the local group and the child processes in the remote group. The ordering of processes in the local and remote groups is the same as the ordering of the group of the comm in the parents and of MPI_COMM_WORLD of the children, respectively. This inter-communicator can be obtained in the children through the function MPI_COMM_GET_PARENT.

An implementation may automatically establish communication Advice to users. before MPI_INIT is called by the children. Thus, completion of MPI_COMM_SPAWN in the parent does not necessarily mean that MPI_INIT has been called in the children (although the returned inter-communicator can be used immediately). (End of advice to users.)

The command argument The command argument is a string containing the name of a program to be spawned. The string is null-terminated in C. In Fortran, leading and trailing spaces are stripped. MPI does not specify how to find the executable or how the working directory is determined. These rules are implementation-dependent and should be appropriate for the runtime environment.

The implementation should use a natural rule for finding Advice to implementors. executables and determining working directories. For instance, a homogeneous system with a global file system might look first in the working directory of the spawning process, or might search the directories in a PATH environment variable as do Unix shells. An implementation should document its rules for finding executables and determining working directories, and a high-quality implementation should give the user some control over these rules. (End of advice to implementors.)

If the program named in command does not call MPI_INIT, but instead forks a process that calls MPI_INIT, the results are undefined. Implementations may allow this case to work but are not required to.

MPI does not say what happens if the program you start is a Advice to users. shell script and that shell script starts a program that calls MPI_INIT. Though some implementations may allow you to do this, they may also have restrictions, such as requiring that arguments supplied to the shell script be supplied to the program, or requiring that certain parts of the environment not be changed. (End of advice to users.)

The argv argument argv is an array of strings containing arguments that are passed to 42the program. The first element of argv is the first argument passed to command, not, as 43 is conventional in some contexts, the command itself. The argument list is terminated by 44NULL in C and an empty string in Fortran. In Fortran, leading and trailing spaces are always stripped, so that a string consisting of all spaces is considered an empty string. The constant MPI_ARGV_NULL may be used in C and Fortran to indicate an empty argument list. In C this constant is the same as NULL.

## **Unofficial Draft for Comment Only**

1  $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6  $\overline{7}$ 

8 9

10

11

12

13

14

151617

18

19

20

21

22

23

 24 

25

26

27

28

2930

 31 

32

33 34

35

36

37

38

39

40 41

45

46

47

```
1
 Example 11.17 Examples of argv in C and Fortran
\mathbf{2}
 To run the program "ocean" with arguments "-gridfile" and "ocean1.grd" in C:
3
 char command[] = "ocean";
4
 char *argv[] = {"-gridfile", "ocean1.grd", NULL};
5
 MPI_Comm_spawn(command, argv, ...);
6
7
 or, if not everything is known at compile time:
8
9
 char *command;
10
 char **argv;
11
 command = "ocean";
 argv=(char **)malloc(3 * sizeof(char *));
12
 argv[0] = "-gridfile";
13
14
 argv[1] = "ocean1.grd";
15
 argv[2] = NULL;
 MPI_Comm_spawn(command, argv, ...);
16
17
 In Fortran:
18
19
 CHARACTER*25 command, argv(3)
20
 command = 'ocean'
21
 argv(1) = '-gridfile'
22
 argv(2) = 'ocean1.grd'
23
 argv(3) = ', '
^{24}
 call MPI_COMM_SPAWN(command, argv, ...
25
26
 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 27respects. First, it is shifted by one element. Specifically, argv[0] of main is provided by the 28implementation and conventionally contains the name of the program (given by command). 29argv[1] of main corresponds to argv[0] in MPI_COMM_SPAWN, argv[2] of main to argv[1] 30 of MPI_COMM_SPAWN, etc. Passing an argv of MPI_ARGV_NULL to MPI_COMM_SPAWN  31 results in main receiving argc of 1 and an argv whose element 0 is (conventionally) the 32 name of the program. Second, argv of MPI_COMM_SPAWN must be null-terminated, so 33 34 that its length can be determined.

³⁵ If a Fortran implementation supplies routines that allow a program to obtain its ar-³⁶ guments, the arguments may be available through that mechanism. In C, if the operating ³⁷ system does not support arguments appearing in argv of main(), the MPI implementation ³⁸ may add the arguments to the argv that is passed to MPI_INIT.

39

The maxprocs argument MPI tries to spawn maxprocs processes. If it is unable to spawn maxprocs processes, it raises an error of class MPI_ERR_SPAWN.

⁴² An implementation may allow the info argument to change the default behavior, such ⁴³ that if the implementation is unable to spawn all maxprocs processes, it may spawn a ⁴⁴ smaller number of processes instead of raising an error. In principle, the info argument ⁴⁵ may specify an arbitrary set  $\{m_i : 0 \le m_i \le \text{maxprocs}\}$  of allowed values for the number ⁴⁶ of processes spawned. The set  $\{m_i\}$  does not necessarily include the value maxprocs. If ⁴⁷ an implementation is able to spawn one of these allowed numbers of processes,

⁴⁸ 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  $\{0, \ldots, N\}$ . However, this is not completely portable, as implementations are not required to support soft spawning. (End of advice to users.)

The info argument The info argument to all of the routines in this chapter is an opaque handle of type MPI_Info in C and Fortran with the mpi_f08 module and INTEGER in Fortran with the mpi module or the include file mpif.h. It is a container for a number of user-specified (key,value) pairs. key and value are strings (null-terminated char* in C, character*(*) in Fortran). Routines to create and manipulate the info argument are described in Chapter 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.

The root argument All arguments before the root argument are examined only on the process whose rank in comm is equal to root. The value of these arguments on other processes is ignored.

The array_of_errcodes argument The array_of_errcodes is an array of length maxprocs in which MPI reports the status of each process that MPI was requested to start. If all maxprocs processes were spawned, array_of_errcodes is filled in with the value MPI_SUCCESS. If only m ( $0 \le m < \text{maxprocs}$ ) 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.

### Unofficial Draft for Comment Only

 $\mathbf{5}$ 

1 2 3 4 5	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.)		-	
6 7	MPI_COM	M_GET_PARENT(par	rent)	
8 9	OUT	parent	the parent communicator (handle)	
10 11	C binding	omm_get_parent(MP]	I_Comm *parent)	
12 13 14 15 16	MPI_Comm_; TYPE()	008 binding get_parent(parent, MPI_Comm), INTENT( ER, OPTIONAL, INTE		
17 18 19		inding GET_PARENT(PARENT, ER PARENT, IERROR		
20 21 22 23 24 25 26 27	MPI_COM cess. This same inter- If the p After t	M_GET_PARENT ret parent inter-communicator return process was not spawn	MPI_COMM_SPAWN or MPI_COMM_SPAWN_MULTurns the "parent" inter-communicator of the current nicator is created implicitly inside of MPI_INIT and ned by SPAWN in the parents. ned, MPI_COMM_GET_PARENT returns MPI_COMM_ cator is freed or disconnected, MPI_COMM_GET_PA	t pro- is the NULL.
28 29 30 31 32 33 34	comm to the or MF invali	unicator. Calling M e same inter-communi PI_COMM_FREE will	COMM_GET_PARENT returns a handle to a single IPI_COMM_GET_PARENT a second time returns a licator. Freeing the handle with MPI_COMM_DISCON cause other references to the inter-communicator to b hat calling MPI_COMM_FREE on the parent commun- ice to users.)	nandle NECT ecome
35 36 37 38 39	canno	COMM_PARENT simil	of the Forum was to create a constant lar to MPI_COMM_WORLD. Unfortunately such a con- ally) as an argument to MPI_COMM_DISCONNECT, <i>l of rationale.</i> )	
40 41	11.8.3 St	arting Multiple Exect	utables and Establishing Communication	
42 43 44 45 46 47 48	of multiple ing routine	binaries, or of the s spawns multiple bin	sufficient for most cases, it does not allow the spa same binary with multiple sets of arguments. The f naries or the same binary with multiple sets of argun h them and placing them in the same MPI_COMM_W	follow- nents,

MPI_COM	IM_SPAWN_MULTIPLE(count	, array_of_commands, array_of_argv,	1
	array_of_maxprocs, array	_of_info, root, comm, intercomm,	2
	array_of_errcodes)		3
IN	count	number of commands (positive integer, significant	4
		only at root)	5
IN	array_of_commands	programs to be executed (array of strings, significant	6 7
		only at root)	8
IN	array_of_argv	arguments for <b>commands</b> (array of array of strings, significant only at root)	9 10
IN	array_of_maxprocs	maximum number of processes to start for each	11
	,	command (array of integers, significant only at root)	12 13
IN	array_of_info	info objects telling the runtime system where and	14
		how to start processes (array of handles, significant	15
		only at root)	16
IN	root	rank of process in which previous arguments are	17
		examined (integer)	18
IN	comm	intra-communicator containing group of spawning	19
		processes (handle)	20
OUT	intercomm	inter-communicator between original group and the	21
001	Intercomm	newly spawned group (handle)	22
			23 24
OUT	array_of_errcodes	one error code per process (array of integers)	24
a			26
C bindin	0		27
int MPI_C		ount, char *array_of_commands[],	28
		<pre>[], const int array_of_maxprocs[],</pre>	29
		<pre>v_of_info[], int root, MPI_Comm comm, int root, [])</pre>	30
	MP1_Comm *intercomm,	int array_of_errcodes[])	31
Fortran 2	2008 binding		32
MPI_Comm_	_spawn_multiple(count, ar	<pre>ray_of_commands, array_of_argv,</pre>	33
	array_of_maxprocs, a	array_of_info, root, comm, intercomm,	34
	array_of_errcodes, i		35
		<pre>array_of_maxprocs(*), root</pre>	36
CHARA	ACTER(LEN=*), INTENT(IN)	·	37
	array_of_argv(count		38
TYPE	(MPI_Info), INTENT(IN) ::	array_of_info(*)	39

**Unofficial Draft for Comment Only** 

ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,

MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,

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

INTEGER :: array_of_errcodes(*)

Fortran binding

TYPE(MPI_Comm), INTENT(OUT) :: intercomm

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

ARRAY_OF_ERRCODES, IERROR)

 41 

INTEGER COUNT, ARRAY_OF_MAXPROCS(*), ARRAY_OF_INFO(*), ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)

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

MPI_COMM_SPAWN_MULTIPLE is identical to MPI_COMM_SPAWN except that there are multiple executable specifications. The first argument, count, gives the number of specifications. Each of the next four arguments are simply arrays of the corresponding arguments in MPI_COMM_SPAWN. For the Fortran version of array_of_argv, the element array_of_argv(i,j) is the j-th argument to command number i.

Rationale. This may seem backwards to Fortran programmers who are familiar with Fortran's column-major ordering. However, it is necessary to do it this way to allow MPI_COMM_SPAWN to sort out arguments. Note that the leading dimension of array_of_argv must be the same as count. Also note that Fortran rules for sequence association allow a different value in the first dimension; in this case, the sequence of array elements is interpreted by MPI_COMM_SPAWN_MULTIPLE as if the sequence is stored in an array defined with the first dimension set to count. This Fortran feature allows an implementor to define MPI_ARGVS_NULL (see below) with fixed dimensions, e.g., (1,1), or only with one dimension, e.g., (1). (End of rationale.)

Advice to users. The argument count is interpreted by MPI only at the root, as is array_of_argv. Since the leading dimension of array_of_argv is count, a non-positive value of count at a non-root node could theoretically cause a runtime bounds check error, even though array_of_argv should be ignored by the subroutine. If this happens, you should explicitly supply a reasonable value of count on the non-root nodes. (End of advice to users.)

2627In 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. 2829The effect of setting individual elements of array_of_argv to MPI_ARGV_NULL is not defined. 30 To specify arguments for some commands but not others, the commands without arguments  31 should have a corresponding argy whose first element is null ((char *)0 in C and empty 32 string in Fortran). In Fortran at non-root processes, the count argument must be set to 33 a value that is consistent with the provided array_of_argv although the content of these 34 arguments has no meaning for this operation.

³⁵ All of the spawned processes have the same MPI_COMM_WORLD. Their ranks in ³⁶ MPI_COMM_WORLD correspond directly to the order in which the commands are specified ³⁷ in MPI_COMM_SPAWN_MULTIPLE. Assume that  $m_1$  processes are generated by the first ³⁸ command,  $m_2$  by the second, etc. The processes corresponding to the first command have ³⁹ ranks 0, 1, ...,  $m_1$ -1. The processes in the second command have ranks  $m_1, m_1$ +1, ...,  $m_1$ + ⁴⁰  $m_2$ -1. The processes in the third have ranks  $m_1$ + $m_2$ +1, ...,  $m_1$ + $m_2$ + $m_3$ -1, ⁴¹ etc.

Advice to users. Calling MPI_COMM_SPAWN multiple times would create many
 sets of children with different MPI_COMM_WORLDs whereas

MPI_COMM_SPAWN_MULTIPLE creates children with a single MPI_COMM_WORLD,
 so the two methods are not completely equivalent. There are also two performance related reasons why, if you need to spawn multiple executables, you may want to
 use MPI_COMM_SPAWN_MULTIPLE instead of calling MPI_COMM_SPAWN several

times. First, spawning several things at once may be faster than spawning them sequentially. Second, in some implementations, communication between processes spawned at the same time may be faster than communication between processes spawned separately. (End of advice to users.)

The array_of_errcodes argument is a 1-dimensional array of size  $\sum_{i=1}^{count} n_i$ , where  $n_i$  is the *i*-th element of array_of_maxprocs. Command number *i* corresponds to the  $n_i$  contiguous slots in this array from element  $\sum_{j=1}^{i-1} n_j$  to  $\left|\sum_{j=1}^{i} n_j\right| - 1$ . Error codes are treated as for MPI_COMM_SPAWN.

```
Example 11.18 Examples of array_of_argv in C and Fortran
To run the program "ocean" with arguments "-gridfile" and "ocean1.grd" and the program
"atmos" with argument "atmos.grd" in C:
 char *array_of_commands[2] = {"ocean", "atmos"};
 char **array_of_argv[2];
 char *argv0[] = {"-gridfile", "ocean1.grd", (char *)0};
 char *argv1[] = {"atmos.grd", (char *)0};
 array_of_argv[0] = argv0;
 array_of_argv[1] = argv1;
 MPI_Comm_spawn_multiple(2, array_of_commands, array_of_argv, ...);
 CHARACTER*25 commands(2), array_of_argv(2, 3)
 commands(1) = 'ocean'
 array_of_argv(1, 1) = '-gridfile'
 array_of_argv(1, 2) = 'ocean1.grd'
 array_of_argv(1, 3) = ' '
```

Here is how you do it in Fortran:

commands(2) = 'atmos'

```
array_of_argv(2, 1) = 'atmos.grd'
array_of_argv(2, 2) = ', '
```

call MPI_COMM_SPAWN_MULTIPLE(2, commands, array_of_argv, ...)

#### 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) 46execute(s). This directory is made the working directory of the executing process(es). 47The format of the directory name is determined by the implementation. 48

### Unofficial Draft for Comment Only

1  $\mathbf{2}$ 

3

4 5

6

 $\overline{7}$ 

8 9

10

11

12

13 14

15

16

17

18

19

20

2122

23

 24 

25

26

27

2829

30

31

32 33

34 35

36 37

38

39 40

41

4243

44

1 2	" <b>path</b> " Value is a directory or set of directories where the implementation should look for the executable. The format of " <b>path</b> " is determined by the implementation.
3 4 5	"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.
6 7 8 9 10 11 12 13	"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, non-standard values may be supported by the implementation, which should document the resultant behavior.
14 15 16 17 18 19 20	"soft" Value specifies a set of numbers which are allowed values for the number of processes that MPI_COMM_SPAWN (et al.) may create. The format of the value is a comma-separated list of Fortran-90 triplets each of which specifies a set of integers and which together specify the set formed by the union of these sets. Negative values in this set and values greater than maxprocs are ignored. MPI will spawn the largest number of processes it can, consistent with some number in the set. The order in which triplets are given is not significant.
21 22	By Fortran-90 triplets, we mean:
23	1. a means $a$
24	2. a:b means $a, a + 1, a + 2,, b$
25 26 27 28	3. a:b:c means $a, a + c, a + 2c,, a + ck$ , where for $c > 0$ , k is the largest integer for which $a + ck \le b$ and for $c < 0$ , k is the largest integer for which $a + ck \ge b$ . If $b > a$ then c must be positive. If $b < a$ then c must be negative.
29 30	Examples:
31	1. <b>a:b</b> gives a range between $a$ and $b$
32	2. 0:N gives full "soft" functionality
33 34 35	3. 1,2,4,8,16,32,64,128,256,512,1024,2048,4096 allows a power-of-two number of processes.
36	4. 2:10000:2 allows an even number of processes.
37	5. 2:10:2,7 allows 2, 4, 6, 7, 8, or 10 processes.
38	
39 40	11.8.5 Spawn Example
41 42	Example 11.19 Manager-worker Example Using MPI_COMM_SPAWN
43 44	
45	
46	
47 48	

```
1
/* manager */
 2
#include "mpi.h"
int main(int argc, char *argv[])
Ł
 int world_size, universe_size, *universe_sizep, flag;
 5
 6
 MPI_Comm everyone;
 /* inter-communicator */
 char worker_program[100];
 a
 MPI_Init(&argc, &argv);
 10
 MPI_Comm_size(MPI_COMM_WORLD, &world_size);
 11
 if (world_size != 1)
 error("Top heavy with management");
 12
 13
 14
 MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,
 15
 &universe_sizep, &flag);
 16
 if (!flag) {
 17
 printf("This MPI does not support UNIVERSE_SIZE. How many\n
 18
processes total?");
 19
 scanf("%d", &universe_size);
 } else universe_size = *universe_sizep;
 20
 21
 if (universe_size == 1) error("No room to start workers");
 22
 /*
 23
 24
 * Now spawn the workers. Note that there is a run-time determination
 25
 * of what type of worker to spawn, and presumably this calculation must
 26
 * be done at run time and cannot be calculated before starting
 * the program. If everything is known when the application is
 27
 * first started, it is generally better to start them all at once
 28
 29
 * in a single MPI_COMM_WORLD.
 */
 30
 31
 choose_worker_program(worker_program);
 32
 33
 MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, universe_size-1,
 MPI_INFO_NULL, 0, MPI_COMM_SELF, &everyone,
 34
 MPI_ERRCODES_IGNORE);
 35
 36
 /*
 37
 * Parallel code here. The communicator "everyone" can be used
 38
 * to communicate with the spawned processes, which have ranks 0,...
 39
 * MPI_UNIVERSE_SIZE-1 in the remote group of the inter-communicator
 40
 * "everyone".
 41
 */
 42
 MPI_Finalize();
 43
 44
 return 0;
}
 45
 46
/* worker */
 47
 48
```

```
1
 #include "mpi.h"
\mathbf{2}
 int main(int argc, char *argv[])
3
 ſ
4
 int size;
5
 MPI_Comm parent;
6
 MPI_Init(&argc, &argv);
7
 MPI_Comm_get_parent(&parent);
8
 if (parent == MPI_COMM_NULL) error("No parent!");
9
 MPI_Comm_remote_size(parent, &size);
10
 if (size != 1) error("Something's wrong with the parent");
11
12
 /*
13
 * Parallel code here.
14
 * The manager is represented as the process with rank 0 in (the remote
15
 * group of) the parent communicator. If the workers need to communicate
16
 * among themselves, they can use MPI_COMM_WORLD.
17
 */
18
19
 MPI_Finalize();
20
 return 0;
21
 }
22
23
24
25
 11.9
 Establishing Communication
26
27
 This section provides functions that establish communication between two sets of MPI
28
 processes that do not share a communicator.
29
 Some situations in which these functions are useful are:
30
31
 1. Two parts of an application that are started independently need to communicate.
32
 2. A visualization tool wants to attach to a running process.
33
34
 3. A server wants to accept connections from multiple clients. Both clients and server
35
 may be parallel programs.
36
37
 In each of these situations, MPI must establish communication channels where none ex-
38
 isted before, and there is no parent/child relationship. The routines described in this
39
 section establish communication between the two sets of processes by creating an MPI
40
 inter-communicator, where the two groups of the inter-communicator are the original sets
41
 of processes.
42
 Establishing contact between two groups of processes that do not share an existing
43
 communicator is a collective but asymmetric process. One group of processes indicates its
44
 willingness to accept connections from other groups of processes. We will call this group
45
 the (parallel) server, even if this is not a client/server type of application. The other group
46
 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.

### Unofficial Draft for Comment Only

 24 

```
1
 2. Applications that use the MPI_PUBLISH_NAME mechanism are completely portable
\mathbf{2}
 among implementations that provide this service. To be portable among all imple-
3
 mentations, these applications should have a fall-back mechanism that can be used
4
 when names are not published.
5
 3. Applications may ignore MPI's name publishing functionality and use their own mech-
6
 anism (possibly system-supplied) to publish names. This allows arbitrary flexibility
7
 but is not portable.
8
9
 11.9.2 Server Routines
10
11
 A server makes itself available with two routines. First it must call MPI_OPEN_PORT to
12
 establish a port at which it may be contacted. Secondly it must call MPI_COMM_ACCEPT
13
 to accept connections from clients.
14
15
16
 MPI_OPEN_PORT(info, port_name)
17
 implementation-specific information on how to
 IN
 info
18
 establish an address (handle)
19
 OUT
 newly established port (string)
 port_name
20
21
22
 C binding
23
 int MPI_Open_port(MPI_Info info, char *port_name)
^{24}
 Fortran 2008 binding
25
 MPI_Open_port(info, port_name, ierror)
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
30
 Fortran binding
^{31}
 MPI_OPEN_PORT(INFO, PORT_NAME, IERROR)
32
 INTEGER INFO, IERROR
33
 CHARACTER*(*) PORT_NAME
34
 This function establishes a network address, encoded in the port_name string, at which
35
 the server will be able to accept connections from clients. port_name is supplied by the
36
 system, possibly using information in the info argument.
37
 MPI copies a system-supplied port name into port_name. port_name identifies the newly
38
 opened port and can be used by a client to contact the server. The maximum size string
39
 that may be supplied by the system is MPI_MAX_PORT_NAME.
40
41
```

41 42

43

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_CL	OSE_PORT(port_name)		19
IN	port_name	a port (string)	20 21
			21
C bind	ing		22
		ar *port_name)	24
	-		25
	n 2008 binding		26
	<pre>ose_port(port_name, ie ARACTER(LEN=*), INTENT</pre>		27
	EGER, OPTIONAL, INTEN	-	28
1111	LOLI, OI HONAL, INILI		29
	n binding		30
	DSE_PORT(PORT_NAME, IN	ERROR)	31
	RACTER*(*) PORT_NAME		32
INT	EGER IERROR		33
Thi	s function releases the ne	etwork address represented by port_name.	34
		······································	35
			36
			37
			38
			39
			40
			41
			42
			43
			44 45
			45 46
			40

 $\mathbf{2}$ 

## 532 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

```
1
 MPI_COMM_ACCEPT(port_name, info, root, comm, newcomm)
2
 IN
 port name (string, significant only at root)
 port_name
3
 IN
 info
 implementation-dependent information (handle,
4
 significant only at root)
5
6
 IN
 rank in comm of root node (integer)
 root
7
 intra-communicator over which call is collective
 IN
 comm
8
 (handle)
9
 OUT
 inter-communicator with client as remote group
 newcomm
10
 (handle)
11
12
 C binding
13
14
 int MPI_Comm_accept(const char *port_name, MPI_Info info, int root,
 MPI_Comm comm, MPI_Comm *newcomm)
15
16
 Fortran 2008 binding
17
 MPI_Comm_accept(port_name, info, root, comm, newcomm, ierror)
18
 CHARACTER(LEN=*), INTENT(IN) :: port_name
19
 TYPE(MPI_Info), INTENT(IN) :: info
20
 INTEGER, INTENT(IN) :: root
21
 TYPE(MPI_Comm), INTENT(IN) :: comm
22
 TYPE(MPI_Comm), INTENT(OUT) :: newcomm
23
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{24}
25
 Fortran binding
26
 MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
27
 CHARACTER*(*) PORT_NAME
 INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
28
29
 MPI_COMM_ACCEPT establishes communication with a client. It is collective over the
30
 calling communicator. It returns an inter-communicator that allows communication with
^{31}
 the client.
32
 The port_name must have been established through a call to MPI_OPEN_PORT.
33
 info can be used to provide directives that may influence the behavior of the ACCEPT
34
 call.
35
36
 11.9.3 Client Routines
37
38
 There is only one routine on the client side.
39
40
41
42
43
44
45
46
47
48
```

	(i		
IN	port_name	network address (string, significant only at root)	24
IN	info	implementation-dependent information (handle,	4
		significant only at root)	Ę
IN	root	rank in comm of root node (integer)	6
IN	comm	intra-communicator over which call is collective	7
		(handle)	8
OUT	newcomm	inter-communicator with server as remote group	1
001	newcomm	(handle)	1
			T

### MPI_COMM_CONNECT(port_name, info, root, comm, newcomm)

### C binding

### 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

### Unofficial Draft for Comment Only

2

6

13

14

15 16

17

18

19

20

21

22

23

24 25

26

27

28 29

30

 31 

32

33

34

35

36

37

38 39

40

41

42

43

44

 $45 \\ 46$ 

47

```
1
 from other connection attempts may prevent a particular connection attempt from being
\mathbf{2}
 satisfied.
3
 port_name is the address of the server. It must be the same as the name returned
4
 by MPI_OPEN_PORT on the server. Some freedom is allowed here. If there are equivalent
\mathbf{5}
 forms of port_name, an implementation may accept them as well. For instance, if port_name
6
 is (hostname:port), an implementation may accept (ip_address:port) as well.
7
8
 Name Publishing
 11.9.4
9
 The routines in this section provide a mechanism for publishing names. A (service_name,
10
 port_name) pair is published by the server, and may be retrieved by a client using the
11
 service_name only. An MPI implementation defines the scope of the service_name, that
12
 is, the domain over which the service_name can be retrieved. If the domain is the empty
13
 set, that is, if no client can retrieve the information, then we say that name publishing
14
 is not supported. Implementations should document how the scope is determined. High-
15
 quality implementations will give some control to users through the info arguments to name
16
 publishing functions. Examples are given in the descriptions of individual functions.
17
18
19
 MPI_PUBLISH_NAME(service_name, info, port_name)
20
 a service name to associate with the port (string)
 IN
 service_name
21
 implementation-specific information (handle)
22
 IN
 info
23
 IN
 a port name (string)
 port_name
^{24}
25
 C binding
26
 int MPI_Publish_name(const char *service_name, MPI_Info info,
27
 const char *port_name)
28
 Fortran 2008 binding
29
30
 MPI_Publish_name(service_name, info, port_name, ierror)
 CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name
31
 TYPE(MPI_Info), INTENT(IN) :: info
32
33
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
 Fortran binding
35
 MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
36
 CHARACTER*(*) SERVICE_NAME, PORT_NAME
37
 INTEGER INFO, IERROR
38
39
 This routine publishes the pair (port_name, service_name) so that an application may
40
 retrieve a system-supplied port_name using a well-known service_name.
^{41}
 The implementation must define the scope of a published service name, that is, the
42
 domain over which the service name is unique, and conversely, the domain over which the
43
 (port name, service name) pair may be retrieved. For instance, a service name may be
^{44}
 unique to a job (where job is defined by a distributed operating system or batch scheduler),
45
 unique to a machine, or unique to a Kerberos realm. The scope may depend on the info
46
 argument to MPI_PUBLISH_NAME.
```

⁴⁷ MPI permits publishing more than one service_name for a single port_name. On the ⁴⁸ 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)			
	29		
IN service_name a service name (strin	ig) 30		
IN info implementation-spec	ific information (handle) 31		
IN port_name a port name (string)	32		
	33		
C binding	34		
int MPI_Unpublish_name(const char *service_name, mPI_Unpublish_nam	PT Info info		
const char *port_name)	36 36		
const char *port_name)	37		
Fortran 2008 binding	38		
<pre>MPI_Unpublish_name(service_name, info, port_name, info)</pre>	ierror) 39		
CHARACTER(LEN=*), INTENT(IN) :: service_name, p	port_name 40		
TYPE(MPI_Info), INTENT(IN) :: info	41		
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	42		
Fortron binding	43		
Fortran binding	44		
MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, I	LERRUR) 45		
CHARACTER*(*) SERVICE_NAME, PORT_NAME	46		
INTEGER INFO, IERROR	47		
	15		

# 536 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

```
1
 This routine unpublishes a service name that has been previously published. Attempt-
\mathbf{2}
 ing to unpublish a name that has not been published or has already been unpublished is
3
 erroneous and is indicated by the error class MPI_ERR_SERVICE.
4
 All published names must be unpublished before the corresponding port is closed and
\mathbf{5}
 before the publishing process exits. The behavior of MPI_UNPUBLISH_NAME is implemen-
6
 tation dependent when a process tries to unpublish a name that it did not publish.
7
 If the info argument was used with MPI_PUBLISH_NAME to tell the implementation
8
 how to publish names, the implementation may require that info passed to
9
 MPI_UNPUBLISH_NAME contain information to tell the implementation how to unpublish
10
 a name.
11
12
 MPI_LOOKUP_NAME(service_name, info, port_name)
13
14
 IN
 service_name
 a service name (string)
15
 IN
 info
 implementation-specific information (handle)
16
 OUT
 port_name
 a port name (string)
17
18
19
 C binding
 int MPI_Lookup_name(const char *service_name, MPI_Info info,
20
21
 char *port_name)
22
 Fortran 2008 binding
23
 MPI_Lookup_name(service_name, info, port_name, ierror)
^{24}
 CHARACTER(LEN=*), INTENT(IN) :: service_name
25
 TYPE(MPI_Info), INTENT(IN) :: info
26
 CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name
27
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
 Fortran binding
29
 MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
30
^{31}
 CHARACTER*(*) SERVICE_NAME, PORT_NAME
32
 INTEGER INFO, IERROR
33
 This function retrieves a port_name published by MPI_PUBLISH_NAME with
34
 service_name. If service_name has not been published, it raises an error in the error class
35
 MPI_ERR_NAME. The application must supply a port_name buffer large enough to hold the
36
 largest possible port name (see discussion above under MPI_OPEN_PORT).
37
 If an implementation allows multiple entries with the same service_name within the
38
 same scope, a particular port_name is chosen in a way determined by the implementation.
39
 If the info argument was used with MPI_PUBLISH_NAME to tell the implementation
40
 how to publish names, a similar info argument may be required for MPI_LOOKUP_NAME.
41
42
 11.9.5
 Reserved Key Values
43
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 11.20** 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 11.21 Ocean/Atmosphere—Relies on Name Publishing

In this example, the "ocean" application is the "server" side of a coupled oceanatmosphere climate model. It assumes that the MPI implementation publishes names.

```
MPI_Open_port(MPI_INFO_NULL, port_name);
 37
 MPI_Publish_name("ocean", MPI_INFO_NULL, port_name);
 38
 39
 MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
 40
 /* do something with intercomm */
 41
 MPI_Unpublish_name("ocean", MPI_INFO_NULL, port_name);
 42
 43
 44
On the client side:
 45
 46
```

### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

45

6

9

10

11 12 13

14

15

16

17

18 19

20

21

22

23

24 25

26

27

28

29

30 31

32

33

34 35 36

47

# 538 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

```
1
 Example 11.22 Simple Client-Server Example
\mathbf{2}
 This is a simple example; the server accepts only a single connection at a time and serves
3
 that connection until the client requests to be disconnected. The server is a single process.
4
 Here is the server. It accepts a single connection and then processes data until it
5
 receives a message with tag 1. A message with tag 0 tells the server to exit.
6
 #include "mpi.h"
7
 int main(int argc, char *argv[])
8
 {
9
 MPI_Comm client;
10
 MPI_Status status;
11
 char port_name[MPI_MAX_PORT_NAME];
12
 double buf[MAX_DATA];
13
 int
 size, again;
14
15
 MPI_Init(&argc, &argv);
16
 MPI_Comm_size(MPI_COMM_WORLD, &size);
17
 if (size != 1) error(FATAL, "Server too big");
18
 MPI_Open_port(MPI_INFO_NULL, port_name);
19
 printf("server available at %s\n", port_name);
20
 while (1) \{
21
 MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
22
 &client);
23
 again = 1;
24
 while (again) {
25
 MPI_Recv(buf, MAX_DATA, MPI_DOUBLE,
26
 MPI_ANY_SOURCE, MPI_ANY_TAG, client, &status);
27
 switch (status.MPI_TAG) {
28
 case 0: MPI_Comm_free(&client);
29
 MPI_Close_port(port_name);
30
 MPI_Finalize();
31
 return 0:
32
 case 1: MPI_Comm_disconnect(&client);
33
 again = 0;
34
 break;
35
 case 2: /* do something */
36
37
 default:
38
 /* Unexpected message type */
39
 MPI_Abort(MPI_COMM_WORLD, 1);
40
 }
41
 }
42
 }
43
 }
44
45
 Here is the client.
46
47
48
```

```
#include "mpi.h"
int main(int argc, char **argv)
{
 MPI_Comm server;
 double buf [MAX_DATA];
 char port_name[MPI_MAX_PORT_NAME];
 MPI_Init(&argc, &argv);
 strcpy(port_name, argv[1]);/* assume server's name is cmd-line arg */
 MPI_Comm_connect(port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
 &server);
 while (!done) {
 tag = 2; /* Action to perform */
 MPI_Send(buf, n, MPI_DOUBLE, 0, tag, server);
 /* etc */
 }
 MPI_Send(buf, 0, MPI_DOUBLE, 0, 1, server);
 MPI_Comm_disconnect(&server);
 MPI_Finalize();
 return 0;
}
```

# 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 a user (or possibly a batch system) runs one of these quasi-static applications, she will usually specify a number of processes to start and a total number of processes that are expected. An application simply needs to know how many slots there are, i.e., how many processes it should spawn.

MPI provides an attribute on MPI_COMM_WORLD, MPI_UNIVERSE_SIZE, that allows the 35application to obtain this information in a portable manner. This attribute indicates the 36 total number of processes that are expected. In Fortran, the attribute is the integer value. 37 In C, the attribute is a pointer to the integer value. An application typically subtracts 38 the size of MPI_COMM_WORLD from MPI_UNIVERSE_SIZE to find out how many processes it 39 should spawn. MPI_UNIVERSE_SIZE is initialized in MPI_INIT and is not changed by MPI. If 40 defined, it has the same value on all processes of MPI_COMM_WORLD. MPI_UNIVERSE_SIZE 41 is determined by the application startup mechanism in a way not specified by MPI. (The 42size of MPI_COMM_WORLD is another example of such a parameter.) 43

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. 48

# Unofficial Draft for Comment Only

2

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

44 45

• An environment variable set by the user.

• Extra information passed to MPI_COMM_SPAWN through the info argument.

An implementation must document how MPI_UNIVERSE_SIZE is set. An implementation may not support the ability to set MPI_UNIVERSE_SIZE, in which case the attribute MPI_UNIVERSE_SIZE is not set.

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

¹¹ MPI_UNIVERSE_SIZE is assumed to have been specified when an application was started, ¹² and is in essence a portable mechanism to allow the user to pass to the application (through ¹³ the MPI process startup mechanism, such as mpiexec) a piece of critical runtime informa-¹⁴ tion. Note that no interaction with the runtime environment is required. If the runtime ¹⁵ environment changes size while an application is running, MPI_UNIVERSE_SIZE is not up-¹⁶ dated, and the application must find out about the change through direct communication ¹⁷ with the runtime system.

19

20

1

 $\mathbf{2}$ 

3 4

5

6

7

8

9

10

# 11.10.2 Singleton MPI Initialization

A high-quality implementation will allow any process (including those not started with a "parallel application" mechanism) to become an MPI process by calling MPI_INIT,

MPI_INIT_THREAD, or MPI_SESSION_INIT. Such a process can then connect to other MPI processes using the MPI_COMM_ACCEPT and MPI_COMM_CONNECT routines, or spawn other MPI processes. MPI does not mandate this behavior, but strongly encourages it where technically feasible.

27 28

29

30

31

32

33 34

35

36

37

38

Advice to implementors. Special coordination is required to start MPI processes belonging to the same MPI_COMM_WORLD in the case of the World Model, or the same "mpi://WORLD" process set in the Sessions Model. The processes must be started at the "same" time, they must have a mechanism to establish communication, etc. Either the user or the operating system must take special steps beyond simply starting processes.

Considering the World Model, when an application enters MPI_INIT, clearly it must be able to determine if these special steps were taken. If a process enters MPI_INIT and determines that no special steps were taken (i.e., it has not been given the information to form an MPI_COMM_WORLD with other processes) it succeeds and forms a singleton MPI program, that is, one in which MPI_COMM_WORLD has size 1.

In some implementations, MPI may not be able to function without an "MPI environment." For example, MPI may require that daemons be running or MPI may not be able to work at all on the front-end of an MPP. In this case, an MPI implementation may either

- 1. Create the environment (e.g., start a daemon) or
- 2. Raise an error if it cannot create the environment and the environment has not been started independently.
- 47 48

44

45

A high-quality implementation will try to create a singleton MPI process and not raise an error.

(End of advice to implementors.)

#### 11.10.3 MPI_APPNUM

There is a predefined attribute MPI_APPNUM of MPI_COMM_WORLD. In Fortran, the attribute is an integer value. In C, the attribute is a pointer to an integer value. If a process was spawned with MPI_COMM_SPAWN_MULTIPLE, MPI_APPNUM is the command number that generated the current process. Numbering starts from zero. If a process was spawned with MPI_COMM_SPAWN, it will have MPI_APPNUM equal to zero.

Additionally, if the process was not started by a spawn call, but by an implementationspecific startup mechanism that can handle multiple process specifications, MPI_APPNUM should be set to the number of the corresponding process specification. In particular, if it is started with

mpiexec spec0 [: spec1 : spec2 : ...]

MPI_APPNUM should be set to the number of the corresponding specification.

If an application was not spawned with MPI_COMM_SPAWN or

MPI_COMM_SPAWN_MULTIPLE, and MPI_APPNUM does not make sense in the context of the implementation-specific startup mechanism, MPI_APPNUM is not set.

MPI implementations may optionally provide a mechanism to override the value of MPI_APPNUM through the info argument. MPI reserves the following key for all SPAWN calls.

"appnum" Value contains an integer that overrides the default value for MPI_APPNUM in the child.

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

- Two processes are **connected** if there is a communication path (direct or indirect) between them. More precisely:
  - 1. Two processes are connected if

 $\frac{4}{5}$ 

 31 

 $46 \\ 47$ 

# 542 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

1	(a) they both belong to the same communicator (inter- or intra-, including
2	MPI_COMM_WORLD) or
3 4	(b) they have previously belonged to a communicator that was freed with
5	MPI_COMM_FREE instead of MPI_COMM_DISCONNECT or
6	(c) they both belong to the group of the same window or filehandle.
7	2. If A is connected to B and B to C, then A is connected to C.
8 9	• Two processes are <b>disconnected</b> (also <b>independent</b> ) if they are not connected.
10	• By the above definitions, connectivity is a transitive property, and divides the uni-
11	verse of MPI processes into disconnected (independent) sets (equivalence classes) of
12	processes.
13	
14	• Processes which are connected, but do not share the same MPI_COMM_WORLD, may
15	become disconnected (independent) if the communication path between them is bro-
16	ken by using MPI_COMM_DISCONNECT.
17 18	The following additional rules apply to MPI routines in other chapters:
19	• MPI_FINALIZE is collective over a set of connected processes.
20	
21	• MPI_ABORT does not abort independent processes. It may abort all processes in
22	the caller's MPI_COMM_WORLD (ignoring its comm argument). Additionally, it may
23	abort connected processes as well, though it makes a "best attempt" to abort only
24	the processes in comm.
25 26	• If a process terminates without calling MPI_FINALIZE, independent processes are not
20	affected but the effect on connected processes is not defined.
28	
29	Advice to implementors. In practice, it may be difficult to distinguish between an
30	MPI process failure and an erroneous program that terminates without calling an
31	MPI finalization function: an implementation that defines semantics for process fail-
32	ure management may have to exhibit the behavior defined for MPI process failures with such erroneous programs. A high quality implementation should exhibit a dif-
33	ferent behavior for erroneous programs and MPI process failures. ( <i>End of advice to</i>
34	implementors.)
35	
36	
37	
38	MPI_COMM_DISCONNECT(comm)
39 40	INOUT comm communicator (handle)
40	
42	C binding
43	<pre>int MPI_Comm_disconnect(MPI_Comm *comm)</pre>
44	Fortran 2008 binding
45	MPI_Comm_disconnect(comm, ierror)
46	TYPE(MPI_Comm), INTENT(INOUT) :: comm
47	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48	

Forman binding	1
	2 3
	4
This function waits for all pending communication on <b>comm</b> to complete internally,	5
collective energies	6 7
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	' 8 9 .0
same as for MPI_FINALIZE.	.1
waits for pending communication to finish internally and enables the guarantee about the	.3
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	.7
to disconnect several communicators (or to free several windows or files) before two	.8
processes are completely independent. (End of advice to users.) $($	.9
	21
function explicitly does not wait for pending communication to complete. (Ena of	22 23
rationale.)	24
11.10.5 Another Way to Establish MPI Communication	25
2	26
	27
MPL COMM IOIN(fd intercomm)	28
	29 80
	31
OUT     intercomm     new inter-communicator (handle)	2
C 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2	3
C binding	4
<pre>int MPI_Comm_join(int fd, MPI_Comm *intercomm) 3</pre>	5
Fortran 2008 binding	86
MPI_Comm_join(fd, intercomm, ierror) 3	7
INTEGER, INTENT(IN) :: fd	8
TYPE(MPI_Comm), INTENT(OUT) :: intercomm	
	10
Fortran binding	11
MPT COMM JOIN (FD. INTERCOMM, IERROR)	12 13
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 environ-	

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 47 and should return MPI_COMM_NULL. 48

# 544 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

This call creates an inter-communicator from the union of two MPI processes which are  $\mathbf{2}$ connected by a socket. MPI_COMM_JOIN should normally succeed if the local and remote 3 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.)

141516

1

4

5

6

7

8

9 10

11

12

13

fd is a file descriptor representing a socket of type SOCK_STREAM (a two-way reliable byte-stream connection). Nonblocking I/O and asynchronous notification via SIGIO must 17not be enabled for the socket. The socket must be in a connected state. The socket must 18 be quiescent when MPI_COMM_JOIN is called (see below). It is the responsibility of the 19application to create the socket using standard socket API calls. 20

MPI_COMM_JOIN must be called by the process at each end of the socket. It does not 21return until both processes have called MPI_COMM_JOIN. The two processes are referred 22 to as the local and remote processes. 23

MPI uses the socket to bootstrap creation of the inter-communicator, and for nothing  24 else. Upon return from MPI_COMM_JOIN, the file descriptor will be open and quiescent 25(see below). 26

If MPI is unable to create an inter-communicator, but is able to leave the socket in its 27original state, with no pending communication, it succeeds and sets intercomm to 28MPI_COMM_NULL. 29

The socket must be quiescent before MPI_COMM_JOIN is called and after 30 MPI_COMM_JOIN returns. More specifically, on entry to MPI_COMM_JOIN, a read on the  31 socket will not read any data that was written to the socket before the remote process called 32 MPI_COMM_JOIN. On exit from MPI_COMM_JOIN, a read will not read any data that was 33 written to the socket before the remote process returned from MPI_COMM_JOIN. It is the 34responsibility of the application to ensure the first condition, and the responsibility of the 35 MPI implementation to ensure the second. In a multithreaded application, the application 36 must ensure that one thread does not access the socket while another is calling 37 MPI_COMM_JOIN, or call MPI_COMM_JOIN concurrently. 38

- 39
- 40 41

42

Advice to implementors. MPI is free to use any available communication path(s) for MPI messages in the new communicator; the socket is only used for the initial handshaking. (End of advice to implementors.)

43 MPI_COMM_JOIN uses non-MPI communication to do its work. The interaction of non-MPI communication with pending MPI communication is not defined. Therefore, the 4445result of calling MPI_COMM_JOIN on two connected processes (see Section 11.10.4 for the 46definition of connected) is undefined.

47The returned communicator may be used to establish MPI communication with addi-48tional 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(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.

9

15 16

17

18

19

20

21

22

23

 24 

25

26

27

28

29

30

31

32

33

34

35

36

37 38

39 40

41

42

43 44

 $45 \\ 46$ 

47

1 MPI supports two fundamentally different *memory models*: separate and *unified*. The  $\mathbf{2}$ separate model makes no assumption about memory consistency and is highly portable. 3 This model is similar to that of weakly coherent memory systems: the user must impose 4 correct ordering of memory accesses through synchronization calls. The unified model can  $\mathbf{5}$ exploit cache-coherent hardware and hardware-accelerated, one-sided operations that are 6 commonly available in high-performance systems. The two different models are discussed 7in detail in Section 12.4. Both models support several synchronization calls to support 8 different synchronization styles.

⁹ The design of the RMA functions allows implementors to take advantage of fast or ¹⁰ asynchronous communication mechanisms provided by various platforms, such as coherent ¹¹ or noncoherent shared memory, DMA engines, hardware-supported put/get operations, and ¹² communication coprocessors. The most frequently used RMA communication mechanisms ¹³ can be layered on top of message-passing. However, certain RMA functions might need ¹⁴ support for asynchronous communication agents in software (handlers, threads, etc.) in a ¹⁵ 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.

23 24

25

12.2 Initialization

²⁶ MPI provides the following window initialization functions: MPI_WIN_CREATE,

²¹ MPI_WIN_ALLOCATE, MPI_WIN_ALLOCATE_SHARED, and

²⁸ MPI_WIN_CREATE_DYNAMIC, which are collective on an intra-communicator.

²⁹ MPI_WIN_CREATE allows each process to specify a "window" in its memory that is made ³⁰ accessible to accesses by remote processes. The call returns an opaque object that represents ³¹ the group of processes that own and access the set of windows, and the attributes of each ³² window, as specified by the initialization call. MPI_WIN_ALLOCATE differs from ³³ MPI_WIN_CREATE is allowed by the initialization call.

³⁴ MPI_WIN_CREATE in that the user does not pass allocated memory;

³⁷ MPI_WIN_ALLOCATE returns a pointer to memory allocated by the MPI implementation. ³⁵ MPI_WIN_ALLOCATE_SHARED differs from MPI_WIN_ALLOCATE in that the allocated ³⁶ memory can be accessed from all processes in the window's group with direct load/store ³⁷ instructions. Some restrictions may apply to the specified communicator.

³⁰ MPI_WIN_CREATE_DYNAMIC creates a window that allows the user to dynamically control ³⁰ which memory is exposed by the window.

- 41
- 42

- 44
- 45
- 46 47
- 48

12.2.1	Window Creation		1
			2 3
MPI_V	VIN_CREATE(base, size, d	lisp_unit, info, comm, win)	4
IN	base	initial address of window (choice)	5
IN	size		6
		size of window in bytes (non-negative integer)	7 8
IN	disp_unit	local unit size for displacements, in bytes (positive integer)	9 10
IN	info	info argument (handle)	10
IN	comm	intra-communicator (handle)	12
Ουτ	win	window object (handle)	13
			14
C bin	ding		15
int MI	PI_Win_create(void *ba	ase, MPI_Aint size, int disp_unit, MPI_Info info,	16 17
	MPI_Comm comm	, MPI_Win *win)	18
int MI	PI_Win_create_c(void >	*base, MPI_Aint size, MPI_Aint disp_unit,	19
		, MPI_Comm comm, MPI_Win *win)	20
Fortr	an 2008 binding		21
	•	disp_unit, info, comm, win, ierror)	22
TYPE(*). DIMENSION(). ASYNCHRONOUS :: base			23
INTEGER(KIND=MPI ADDRESS KIND). INTENT(IN) :: size			24 25
INTEGER, INTENT(IN) :: disp unit			25 26
TYPE(MPI_Info), INTENT(IN) :: info			27
	TYPE(MPI_Comm), INTENT(IN) :: comm		
	PE(MPI_Win), INTENT(		29
11	NTEGER, OPTIONAL, INT	ENI(UUI) :: lerror	30
		disp_unit, info, comm, win, ierror)	31
		, ASYNCHRONOUS :: base	32
		SS_KIND), INTENT(IN) :: size, disp_unit	33 34
	TYPE(MPI_Info), INTENT(IN) :: info TYPE(MPI_Comm), INTENT(IN) :: comm		
	YPE(MPI_Win), INTENT(		36
	VTEGER, OPTIONAL, INTI		37
Fonta	n hinding		38
	an binding	DISP_UNIT, INFO, COMM, WIN, IERROR)	39
	type> BASE(*)	DIGI_GMIT, INLO, COINT, WIN, ILIULOID/	40
	INTEGER (KIND=MPI ADDRESS KIND) SIZE		
	TEGER DISP_UNIT, INF		42 43
			40

44This is a collective call executed by all processes in the group of comm. It returns 45a window object that can be used by these processes to perform RMA operations. Each 46process specifies a window of existing memory that it exposes to RMA accesses by the processes in the group of comm. The window consists of size bytes, starting at address 4748base. In C, base is the starting address of a memory region. In Fortran, one can pass the

1	first element of a memory region or a whole array, which must be 'simply contiguous' (for
2	'simply contiguous,' see also Section 19.1.12). A process may elect to expose no memory
3	by specifying size $= 0$ .
4	The displacement unit argument is provided to facilitate address arithmetic in RMA
5	operations: the target displacement argument of an RMA operation is scaled by the factor
6	disp_unit specified by the target process, at window creation.
7	
8	Rationale. The window size is specified using an address-sized integer, rather than a
9	basic integer type, to allow windows that span more memory than can be described
10	with a basic integer type. ( <i>End of rationale.</i> )
11	
12	Advice to users. Common choices for disp_unit are 1 (no scaling), and (in C syntax)
13	sizeof(type), for a window that consists of an array of elements of type type. The
14	latter choice will allow one to use array indices in RMA calls, and have those scaled
15	correctly to byte displacements, even in a heterogeneous environment. (End of advice
16	to users.)
17	
18	The info argument provides optimization hints to the runtime about the expected usage
19	pattern of the window. The following info keys are predefined:
20	
21	"no_locks"—if set to true, then the implementation may assume that passive target synchro-
22	nization (i.e., MPI_WIN_LOCK, MPI_WIN_LOCK_ALL) will not be used on the given
23	window. This implies that this window is not used for 3-party communication, and
24	RMA can be implemented with no (less) asynchronous agent activity at this process.
25	"accumulate_ordering"—controls the ordering of accumulate operations at the target. See
26	Section 12.7.2 for details.
27	
28	"accumulate_ops"—if set to "same_op", the implementation will assume that all concurrent
29	accumulate calls to the same target address will use the same operation. If set to
30	"same_op_no_op", then the implementation will assume that all concurrent accumulate
31	calls to the same target address will use the same operation or MPI_NO_OP. This can
32	eliminate the need to protect access for certain operation types where the hardware
33	can guarantee atomicity. The default is "same_op_no_op".
34	
35	"same_size"—if set to true, then the implementation may assume that the argument size is
36	identical on all processes, and that all processes have provided this info key with the
37	same value.
38	"same_disp_unit"—if set to true, then the implementation may assume that the argument
39	disp_unit is identical on all processes, and that all processes have provided this info
40	key with the same value.
41	key with the same value.
42	Advice to users. The info query mechanism described in Section 12.2.7 can be used
43	to query the specified info arguments for windows that have been passed to a library.
44	It is recommended that libraries check attached info keys for each passed window.
45	(End of advice to users.)
46	
47	The various processes in the group of <b>comm</b> may specify completely different target
48	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.*)

1

 $\mathbf{2}$ 

3

4

5

6

 $\overline{7}$ 

8

9

10

11

12

13 14

15

16

17

18

19 20

21

22

23

24

25

26

27

28

29

30 31

32 33

34

35

```
550
 CHAPTER 12. ONE-SIDED COMMUNICATIONS
1
 12.2.2 Window That Allocates Memory
\mathbf{2}
3
4
 MPI_WIN_ALLOCATE(size, disp_unit, info, comm, baseptr, win)
5
 IN
 size
 size of window in bytes (non-negative integer)
6
 IN
 disp_unit
 local unit size for displacements, in bytes (positive
7
8
 integer)
9
 IN
 info
 info argument (handle)
10
 IN
 comm
 intra-communicator (handle)
11
 initial address of window (choice)
 OUT
 baseptr
12
13
 OUT
 win
 window object returned by call (handle)
14
15
 C binding
16
 int MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info,
17
 MPI_Comm comm, void *baseptr, MPI_Win *win)
18
19
 int MPI_Win_allocate_c(MPI_Aint size, MPI_Aint disp_unit, MPI_Info info,
 MPI_Comm comm, void *baseptr, MPI_Win *win)
20
21
 Fortran 2008 binding
22
 MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror)
23
 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
24
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
25
 INTEGER, INTENT(IN) :: disp_unit
26
 TYPE(MPI_Info), INTENT(IN) :: info
27
 TYPE(MPI_Comm), INTENT(IN) :: comm
28
 TYPE(C_PTR), INTENT(OUT) :: baseptr
29
 TYPE(MPI_Win), INTENT(OUT) :: win
30
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32
 MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror)
33
 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
34
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size, disp_unit
 TYPE(MPI_Info), INTENT(IN) :: info
35
 TYPE(MPI_Comm), INTENT(IN) :: comm
36
37
 TYPE(C_PTR), INTENT(OUT) :: baseptr
 TYPE(MPI_Win), INTENT(OUT) :: win
38
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
 Fortran binding
41
 MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
42
 INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
43
 INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
44
45
 This is a collective call executed by all processes in the group of comm. On each
46
 process, it allocates memory of at least size bytes, returns a pointer to it, and returns a
47
 window object that can be used by all processes in comm to perform RMA operations. The
```

returned memory consists of size bytes local to each process, starting at address baseptr

and is associated with the window as if the user called MPI_WIN_CREATE on existing memory. The size argument may be different at each process and size = 0 is valid; however, a library might allocate and expose more memory in order to create a fast, globally symmetric allocation. The discussion of and rationales for MPI_ALLOC_MEM and MPI_FREE_MEM in Section 9.2 also apply to MPI_WIN_ALLOCATE; in particular, see the rationale in Section 9.2 for an explanation of the type used for **baseptr**.

If the Fortran compiler provides TYPE(C_PTR), then the following generic interface must be provided in the mpi module and should be provided in mpif.h through overloading, i.e., with the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND) **BASEPTR**, but with a different specific procedure name:

```
INTERFACE MPI_WIN_ALLOCATE
 SUBROUTINE MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
 WIN, IERROR)
 IMPORT :: MPI_ADDRESS_KIND
 INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
 INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
 END SUBROUTINE
 SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
 WIN, IERROR)
 USE, INTRINSIC ::
 ISO_C_BINDING, ONLY : C_PTR
 IMPORT :: MPI_ADDRESS_KIND
 INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
 INTEGER(KIND=MPI_ADDRESS_KIND) ::
 SIZE
 TYPE(C_PTR) ::
 BASEPTR
 END SUBROUTINE
END INTERFACE
```

The base procedure name of this overloaded function is MPI_WIN_ALLOCATE_CPTR. The implied specific procedure names are described in Section 19.1.5.

Rationale. By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with remote direct memory access. This also permits the collective allocation of memory and supports what is sometimes called the "symmetric allocation" model that can be more scalable (for example, the implementation can arrange to return an address for the allocated memory that is the same on all processes). (End of rationale.)

The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE and MPI_ALLOC_MEM.

The default memory alignment requirements and the "mpi_minimum_memory_alignment" info key described for MPI_ALLOC_MEM in Section 9.2 apply to all processes with non-zero size argument.

1

2

3

4

56

 $\overline{7}$ 

8

9 10

11

12

13

14

15

16

17

18

19

20

21

22

23

 24 

25

26

2728

29

30 31

32

33

34

35

36

37 38

39

40

41

42

	552		CHAPTER 12. ONE-SIDED COMMUNICATIONS
1 2 3	12.2.3 W	/indow That Allocates Sha	ared Memory
4	MPI_WIN_	_ALLOCATE_SHARED(size	, disp_unit, info, comm, baseptr, win)
5 6	IN	size	size of local window in bytes (non-negative integer)
7 8	IN	disp_unit	local unit size for displacements, in bytes (positive integer)
9	IN	info	info argument (handle)
10 11	IN	comm	intra-communicator (handle)
12	OUT	baseptr	address of local allocated window segment (choice)
13	OUT	win	window object returned by the call (handle)
15 16 17 18	C binding int MPI_Win_allocate_shared(MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, void *baseptr, MPI_Win *win)		
19 20 21	int MPI_W		IPI_Aint size, MPI_Aint disp_unit, I_Comm comm, void *baseptr, MPI_Win *win)
22 23 24 25 26 27 28 29 30 31	<pre>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 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</pre>		
32 33 34 35 36 37 38 39	<pre>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, 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</pre>		
40 41 42 43 44	INTEG INTEG	ALLOCATE_SHARED(SIZE, E ER(KIND=MPI_ADDRESS_KI ER DISP_UNIT, INFO, CO	MM, WIN, IERROR
45 46 47 48	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		

1 target of load/store accesses by remote processes; the base pointers for other processes  $\mathbf{2}$ can be queried using the function MPI_WIN_SHARED_QUERY. The call also returns a 3 window object that can be used by all processes in comm to perform RMA operations. 4 The size argument may be different at each process and size = 0 is valid. It is the user's responsibility to ensure that the communicator **comm** represents a group of processes that 5can create a shared memory segment that can be accessed by all processes in the group. 6  $\overline{7}$ 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 8 9 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 10 11process 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 1213 calculate remote address offsets with local information only.

If the Fortran compiler provides TYPE(C_PTR), then the following generic interface must be provided in the mpi module and should be provided in mpif.h through overloading, i.e., with the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR, but with a different specific procedure name:

INTERFACE MPI_WIN_ALLOCATE_SHARED
SUBROUTINE MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, &
BASEPTR, WIN, IERROR)
IMPORT :: MPI_ADDRESS_KIND
INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
END SUBROUTINE
SUBROUTINE MPI_WIN_ALLOCATE_SHARED_CPTR(SIZE, DISP_UNIT, INFO, COMM, &
BASEPTR, WIN, IERROR)
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
IMPORT :: MPI_ADDRESS_KIND
INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
TYPE(C_PTR) :: BASEPTR
END SUBROUTINE
END INTERFACE

The base procedure name of this overloaded function is MPI_WIN_ALLOCATE_SHARED_CPTR. The implied specific procedure names are described in Section 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.)

## Unofficial Draft for Comment Only

14

15

16 17

18

34 35

36

37

38

39

40

41

Advice to implementors. If the user sets the info key "alloc_shared_noncontig" to true, the implementation can allocate the memory requested by each process in a location that is close to this process. This can be achieved by padding or allocating memory in special memory segments. Both techniques may make the address space across consecutive ranks noncontiguous. (*End of advice to implementors.*)

For contiguous shared memory allocations, the default alignment requirements outlined for MPI_ALLOC_MEM in Section 9.2 and the "mpi_minimum_memory_alignment" info key apply to the start of the contiguous memory that is returned in baseptr to the first process with non-zero size argument. For noncontiguous memory allocations, the default alignment requirements and the "mpi_minimum_memory_alignment" info key apply to all processes with non-zero size argument.

Advice to users. If the info key "alloc_shared_noncontig" is not set to true (or ignored by the MPI implementation), the alignment of the memory returned in baseptr to all but the first process with non-zero size argument depends on the value of the size argument provided by other processes. It is thus the user's responsibility to control the alignment of contiguous memory allocated for these processes by ensuring that each process provides a size argument that is an integral multiple of the alignment required for the application. (End of advice to users.)

The consistency of load/store accesses from/to the shared memory as observed by the user program depends on the architecture. A consistent view can be created in the *unified memory model* (see Section 12.4) by utilizing the window synchronization functions (see Section 12.5) or explicitly completing outstanding store accesses (e.g., by calling MPI_WIN_FLUSH). MPI does not define semantics for accessing shared memory windows in the *separate memory model*.

MPI_WIN_SHARED_QUERY(win, rank, size, disp_unit, baseptr)

30		•	
31	IN	win	shared memory window object (handle)
32	IN	rank	rank in the group of window win or
33			$MPI_PROC_NULL\ (\mathrm{non-negative\ integer})$
34	OUT	size	size of the window segment (non-negative integer)
35 36 37	OUT	disp_unit	local unit size for displacements, in bytes (positive integer)
38 39 40	OUT	baseptr	address for load/store access to window segment (choice)
40	C bindi	ıg	
42 43	int MPI_	Win_shared_query(MPI_Win int *disp_unit, void	win, int rank, MPI_Aint *size, 1 *baseptr)
44 45 46	int MPI_Win_shared_query_c(MPI_Win win, int rank, MPI_Aint *size, MPI_Aint *disp_unit, void *baseptr)		
47 48		2008 binding shared_query(win, rank, s	size, disp_unit, baseptr, ierror)

1

2

3

4

5

6

7

8

9

10

11

12 13

14

15

16

17

18

19

20

USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR TYPE(MPI_Win), INTENT(IN) :: win INTEGER, INTENT(IN) :: rank INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size INTEGER, INTENT(OUT) :: disp_unit TYPE(C_PTR), INTENT(OUT) :: baseptr INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Win_shared_query(win, rank, size, disp_unit, baseptr, ierror) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR TYPE(MPI_Win), INTENT(IN) :: win INTEGER, INTENT(IN) :: rank INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size, disp_unit TYPE(C_PTR), INTENT(OUT) :: baseptr INTEGER, OPTIONAL, INTENT(OUT) :: ierror

#### Fortran binding

MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, BASEPTR, IERROR) INTEGER WIN, RANK, DISP_UNIT, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR

This function queries the process-local address for remote memory segments created with MPI_WIN_ALLOCATE_SHARED. This function can return different process-local addresses for the same physical memory on different processes. The returned memory can be used for load/store accesses subject to the constraints defined in Section 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, &	36 37
BASEPTR, IERROR)	37
IMPORT :: MPI_ADDRESS_KIND	38 39
INTEGER WIN, RANK, DISP_UNIT, IERROR	39 40
INTEGER (KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR	40
END SUBROUTINE	41 42
SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT,	
BASEPTR, IERROR)	40
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	45
IMPORT :: MPI_ADDRESS_KIND	46
INTEGER :: WIN, RANK, DISP_UNIT, IERROR	47
INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE	48

1

2

5 6

9

10

11

12

13

14

1516

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

1	TYPE(C_PTR) :: BASEPTR		
2	END SUBROUTINE		
3	END INTERFACE		
4 5	The base procedure name of this overloaded function is		
6	MPI_WIN_SHARED_QUERY_CPTR. The implied specific procedure names are described in		
7	Section 19.1.5.		
8			
9 10	12.2.4 Window of Dynamically Attached Memory		
11	The MPI-2 RMA model requires the user to identify the local memory that may be a		
12	target of RMA calls at the time the window is created. This has advantages for both		
13	the programmer (only this memory can be updated by one-sided operations and provides		
14	greater safety) and the MPI implementation (special steps may be taken to make one-		
15	sided access to such memory more efficient). However, consider implementing a modifiable		
16	linked list using RMA operations; as new items are added to the list, memory must be		
17	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		
18	amount of memory and then implement routines for allocating memory from within the		
19	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		
20 21	predefined amount of memory turns out to be inadequate. To support this model, the		
21	routine MPI_WIN_CREATE_DYNAMIC creates a window that makes it possible to expose		
23	memory without remote synchronization. It must be used in combination with the local		
24	routines MPI_WIN_ATTACH and MPI_WIN_DETACH.		
25			
26	MPI_WIN_CREATE_DYNAMIC(info, comm, win)		
27 28	IN info info info argument (handle)		
29	IN comm intra-communicator (handle)		
30	OUT win window object returned by the call (handle)		
31	window object returned by the can (nandle)		
32	C binding		
33	int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win)		
34			
35	Fortran 2008 binding		
36	MPI_Win_create_dynamic(info, comm, win, ierror)		
37	TYPE(MPI_Info), INTENT(IN) :: info		
38 39	TYPE(MPI_Comm), INTENT(IN) :: comm		
39 40	TYPE(MPI_Win), INTENT(OUT) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror		
41	INIEGER, UFILUMAE, INIENI(UUI) :: LETIOF		
42	Fortran binding		
43	MPI_WIN_CREATE_DYNAMIC(INFO, COMM, WIN, IERROR)		
44	INTEGER INFO, COMM, WIN, IERROR		
45	This is a collective call executed by all processes in the group of comm. It returns		
46	a window win without memory attached. Existing process memory can be attached as		
47	described below. This routine returns a window object that can be used by these processes to		
48	• • • •		

perform RMA operations on attached memory. Because this window has special properties, it will sometimes be referred to as a *dynamic* window.

The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE.

In the case of a window created with MPI_WIN_CREATE_DYNAMIC, the target_disp for all RMA functions is the address at the target; i.e., the effective window_base is MPI_BOTTOM and the disp_unit is one. For dynamic windows, the target_disp argument to RMA communication operations is not restricted to non-negative values. Users should use MPI_GET_ADDRESS at the target process to determine the address of a target memory location and communicate this address to the origin process.

Advice to users. Users are cautioned that displacement arithmetic can overflow in variables of type MPI_Aint and result in unexpected values on some platforms. The MPI_AINT_ADD and MPI_AINT_DIFF functions can be used to safely perform address arithmetic with MPI_Aint displacements. (End of advice to users.)

Advice to implementors. In environments with heterogeneous data representations, care must be exercised in communicating addresses between processes. For example, it is possible that an address valid at the target process (for example, a 64-bit pointer) cannot be expressed as an address at the origin (for example, the origin uses 32-bit pointers). For this reason, a portable MPI implementation should ensure that the type MPI_AINT (see Table 3.3) is able to store addresses from any process. (End of advice to implementors.)

Memory at the target cannot be accessed with this window until that memory has been attached using the function MPI_WIN_ATTACH. That is, in addition to using MPI_WIN_CREATE_DYNAMIC to create an MPI window, the user must use MPI_WIN_ATTACH before any local memory may be the target of an MPI RMA operation. Only memory that is currently accessible may be attached.

MPI_WIN_ATTACH(win, base, size)

IN	win	window object (handle)	32	
IN	base	initial address of memory to be attached (choice)	33 34	
IN	size	size of memory to be attached in bytes (non-negative	35	
		integer)	36	
			37	
C bindi	ng		38	
int MPI_	_Win_attach(MPI_Win win, v	oid *base, MPI_Aint size)	39	
<b>F</b> (	4			
Fortran 2008 binding			41	
			42	
	E(MPI_Win), INTENT(IN) ::		43	
	TYPE(*), DIMENSION(), ASYNCHRONOUS :: base			
INTE	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size			
INTE	EGER, OPTIONAL, INTENT(OUT)	) :: ierror	46	
Fortran	Fortran binding 47			
	0	ERBOR)	48	

MPI_WIN_ATTACH(WIN, BASE, SIZE, IERROR)

**Unofficial Draft for Comment Only** 

2

3

4

5

6  $\overline{7}$ 

8 9

10

11

12

13

14

1516

17

18

19

20

21

22

23 24 

25

26

27

28

29 30

1	INTEGER WIN, IERROR		
2 3	<type> BASE(*)</type>		
4	INTEGER(KIND=MPI_ADDRESS_KIND) SIZE		
5	Attaches a local me	mory region beginning at <b>base</b> for remote access within the given	
6		egion specified must not contain any part that is already attached	
7		is, attaching overlapping memory concurrently within the same	
8		he argument win must be a window that was created with	
9		VAMIC. The local memory region attached to the window consists	
10		address base. In C, base is the starting address of a memory region. Is the first element of a memory region or a whole array, which	
11		ous' (for 'simply contiguous,' see Section 19.1.12). Multiple (but	
12		y regions may be attached to the same window.	
13	non overlæpping) memor	y regions may be accalled to the same window.	
14 15		iring that memory be explicitly attached before it is exposed to	
16	-	other processes can simplify implementations and improve perfor-	
17		to make memory available for RMA operations without requiring a	
18		<b>J_CREATE</b> call is needed for some one-sided programming models.	
19	(End of rationale.)		
20	Advice to users.	Attaching memory to a window may require the use of scarce	
21	, , ,	taching large regions of memory is not recommended in portable	
22		ing memory to a window may fail if sufficient resources are not	
23	available; this is sir	milar to the behavior of MPI_ALLOC_MEM.	
24		sponsible for ensuring that MPI_WIN_ATTACH at the target has	
25 26	returned before a p	process attempts to target that memory with an MPI RMA call.	
27		A operation to memory that has not been attached to a window	
28	created with $MPL$	WIN_CREATE_DYNAMIC is erroneous. ( <i>End of advice to users.</i> )	
29	Advice to impleme	<i>ntors.</i> A high-quality implementation will attempt to make as	
30	1	ilable for attaching as possible. Any limitations should be docu-	
31		lementor. (End of advice to implementors.)	
32 33			
33 34	0	is a local operation as defined by MPI, which means that the call pletes without requiring any MPI routine to be called in any other	
35		e detached with the routine MPI_WIN_DETACH. After memory has	
36		bt be the target of an MPI RMA operation on that window (unless	
37	,	ed with MPI_WIN_ATTACH).	
38	U U	,	
39			
40	MPI_WIN_DETACH(win,	base)	
41	IN win	window object (handle)	
42	IN base	initial address of memory to be detached (choice)	
43 44			
45	C binding		
46	<pre>int MPI_Win_detach(MF</pre>	PI_Win win, const void *base)	
47	Fortran 2008 binding		
48	MPI_Win_detach(win, base, ierror)		

TYPE(MPI_Win), INTENT(IN) :: win	1
TYPE(*), DIMENSION(), ASYNCHRONOUS :: base	2
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	3
Fortran binding	4
MPI_WIN_DETACH(WIN, BASE, IERROR)	5
INTEGER WIN, IERROR	6
<pre><type> BASE(*)</type></pre>	7
	8
Detaches a previously attached memory region beginning at base. The arguments base and win must match the arguments passed to a previous call to MPI_WIN_ATTACH.	9 10
	11
Advice to users. Detaching memory may permit the implementation to make more	12
efficient use of special memory or provide memory that may be needed by a subsequent	13
MPI_WIN_ATTACH. Users are encouraged to detach memory that is no longer needed. Memory should be detached before it is freed by the user. ( <i>End of advice to users.</i> )	14
Memory should be detached before it is need by the user. ( <i>Entu of dubice to users.</i> )	15 16
Memory becomes detached when the associated dynamic memory window is freed, see	16
Section 12.2.5.	18
	10
12.2.5 Window Destruction	20
	21
	22
MPI_WIN_FREE(win)	23
INOUT win window object (handle)	24
window object (nancie)	25
Chinding	26
C binding int MPI_Win_free(MPI_Win *win)	27
IIIC MFI_WIII_IIee(MFI_WIII *WIII)	28
Fortran 2008 binding	29
MPI_Win_free(win, ierror)	30
TYPE(MPI_Win), INTENT(INOUT) :: win	31
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	32
Fortran binding	33
MPI_WIN_FREE(WIN, IERROR)	34
INTEGER WIN, IERROR	35
	36 37
Frees the window object win and returns a null handle (equal to MPI_WIN_NULL).	38
This is a collective call executed by all processes in the group associated with win.	39
MPI_WIN_FREE(win) can be invoked by a process only after it has completed its involvement in RMA communications on window win: e.g., the process has called	40
MPI_WIN_FENCE, or called MPI_WIN_WAIT to match a previous call to MPI_WIN_POST	41
or called MPI_WIN_COMPLETE to match a previous call to MPI_WIN_START or called	42
MPI_WIN_UNLOCK to match a previous call to MPI_WIN_LOCK. The memory associated	43

MPI_WIN_UNLOCK to match a previous call to MPI_WIN_LOCK. The memory associated
 with windows created by a call to MPI_WIN_CREATE may be freed after the call returns. If
 the window was created with MPI_WIN_ALLOCATE, MPI_WIN_FREE will free the window
 memory that was allocated in MPI_WIN_ALLOCATE. If the window was created with
 MPI_WIN_ALLOCATE_SHARED, MPI_WIN_FREE will free the window memory that was
 allocated in MPI_WIN_ALLOCATE_SHARED.

Freeing a window that was created with a call to MPI_WIN_CREATE_DYNAMIC detaches all associated memory; i.e., it has the same effect as if all attached memory was detached by calls to MPI_WIN_DETACH.

Advice to implementors. MPI_WIN_FREE requires a barrier synchronization: no process can return from free until all processes in the group of win call free. This ensures that no process will attempt to access a remote window (e.g., with lock/unlock) after it was freed. The only exception to this rule is when the user sets the "no_locks" info key to "true" when creating the window. In that case, an MPI implementation may free the local window without barrier synchronization. (End of advice to implementors.)

12

# 12.2.6 Window Attributes

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

15	MPI_WIN_BASE	window base address.
16	WIFT_WIN_DASE	· · · · · · · · · · · · · · · · · · ·
17	MPI_WIN_SIZE	window size, in bytes.
	MPI_WIN_DISP_UNIT	displacement unit associated with the window.
18	MPI_WIN_CREATE_FLAVOR	how the window was created.
19	MPI_WIN_MODEL	memory model for window.
20		J
21	In C, calls to MPI_Win_get_attr(wi	n, MPI_WIN_BASE, &base, &flag),
22	MPI_Win_get_attr(win, MPI_WIN_SIZE,	, &size, &flag),

- 22MPI_Win_get_attr(win, MPI_WIN_DISP_UNIT, &disp_unit, &flag),
- 23MPI_Win_get_attr(win, MPI_WIN_CREATE_FLAVOR, &create_kind, &flag), and
- 24 MPI_Win_get_attr(win, MPI_WIN_MODEL, & memory_model, & flag) will return in base a 25pointer to the start of the window win, and will return in size, disp_unit, create_kind, and 26memory_model pointers to the size, displacement unit of the window, the kind of routine 27used to create the window, and the memory model, respectively. A detailed listing of the 28type of the pointer in the attribute value argument to MPI_WIN_GET_ATTR and 29 MPI_WIN_SET_ATTR is shown in Table 12.1. 30

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 *

39 Table 12.1: C types of attribute value argument to MPI_WIN_GET_ATTR and 40MPI_WIN_SET_ATTR 41

- 42In Fortran, calls to MPI_WIN_GET_ATTR(win, MPI_WIN_BASE, base, flag, ierror),
- 43MPI_WIN_GET_ATTR(win, MPI_WIN_SIZE, size, flag, ierror),
- 44MPI_WIN_GET_ATTR(win, MPI_WIN_DISP_UNIT, disp_unit, flag, ierror),
- 45MPI_WIN_GET_ATTR(win, MPI_WIN_CREATE_FLAVOR, create_kind, flag, ierror), and
- 46MPI_WIN_GET_ATTR(win, MPI_WIN_MODEL, memory_model, flag, ierror) will return in
- 47base, size, disp_unit, create_kind, and memory_model the (integer representation of) the 48

560

1

 $\mathbf{2}$ 

3

4

5

6

7

8

9

10 11

13

14

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)		22
IN	win	window object (handle)	23
0.UT			24
OUT	group	group of processes which share access to the window	25
		(handle)	26
			27
C bindin	5		28
int MPI_N	<pre>Min_get_group(MPI_Win win;</pre>	, MPI_Group *group)	29
Fortran ⁴	2008 binding		30
	get_group(win, group, ier)	ror	31
	(MPI_Win), INTENT(IN) :: v		32
	(MPI_Group), INTENT(OUT)		33
	• ·	<b>5</b> I	34
INTE	GER, OPTIONAL, INTENT(OUT)	) :: lerror	35
Fortran	binding		36
MPI_WIN_	GET_GROUP(WIN, GROUP, IER	ROR)	37
INTE	GER WIN, GROUP, IERROR		38
			39

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 

## Unofficial Draft for Comment Only

 $41 \\ 42$ 

1

56

7

8

9

10 11

1213 14

15

21

2526

30

38

39

40

41

42

43

a restriction on the behavior of the application. Hints are specified on a per window basis,  $\mathbf{2}$ in window creation functions and MPI_WIN_SET_INFO, via the opaque info object. When 3 an info object that specifies a subset of valid hints is passed to MPI_WIN_SET_INFO there 4 will be no effect on previously set or default hints that the info does not specify.

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
 window object (handle)
16
 win
17
 IN
 info
 info argument (handle)
18
19
 C binding
20
```

```
int MPI_Win_set_info(MPI_Win win, MPI_Info info)
```

```
Fortran 2008 binding
22
```

```
MPI_Win_set_info(win, info, ierror)
23
```

```
TYPE(MPI_Win), INTENT(IN) :: win
^{24}
```

```
TYPE(MPI_Info), INTENT(IN) :: info
```

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

#### 27Fortran binding 28

```
MPI_WIN_SET_INFO(WIN, INFO, IERROR)
29
```

```
INTEGER WIN, INFO, IERROR
```

 31 MPI_WIN_SET_INFO updates the hints of the window associated with win using the 32 hints provided in info. This operation has no effect on previously set or defaulted hints 33 that are not specified by info. It also has no effect on previously set or defaulted hints that 34are specified by info, but are ignored by the MPI implementation in this call to MPI_WIN_SET_INFO. The call is collective on the group of win. The info object may be 35 36 different on each process, but any info entries that an implementation requires to be the 37 same on all processes must appear with the same value in each process's info object.

Advice to users. Some info items that an implementation can use when it creates a window cannot easily be changed once the window has been created. Thus, an implementation may ignore hints issued in this call that it would have accepted in a creation call. An implementation may also be unable to update certain info hints in a call to MPI_WIN_SET_INFO. MPI_WIN_GET_INFO can be used to determine whether info changes were ignored by the implementation. (End of advice to users.)

MPI_WIN_GET_INFO(win, info_used)

IN	win	window object (handle)
OUT	info_used	new info object (handle)

C binding

int MPI_Win_get_info(MPI_Win win, MPI_Info *info_used)

#### Fortran 2008 binding

MPI_Win_get_info(win, info_used, ierror)
 TYPE(MPI_Win), INTENT(IN) :: win
 TYPE(MPI_Info), INTENT(OUT) :: info_used
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

#### Fortran binding

MPI_WIN_GET_INFO(WIN, INFO_USED, IERROR) INTEGER WIN, INFO_USED, IERROR

MPI_WIN_GET_INFO returns a new info object containing the hints of the window associated with win. The current setting of all hints related to this window is returned in info_used. An MPI implementation is required to return all hints that are supported by the implementation and have default values specified; any user-supplied hints that were not ignored by the implementation; and any additional hints that were set by the implementation. If no such hints exist, a handle to a newly created info object is returned that contains no key/value pair. The user is responsible for freeing info_used via MPI_INFO_FREE.

# 12.3 Communication Calls

MPI supports the following RMA communication calls: MPI_PUT and MPI_RPUT transfer data from the caller memory (origin) to the target memory; MPI_GET and MPI_RGET transfer data from the target memory to the caller memory; MPI_ACCUMULATE and MPI_RACCUMULATE update locations in the target memory, e.g., by adding to these locations values sent from the caller memory; MPI_GET_ACCUMULATE,

MPI_RGET_ACCUMULATE, and MPI_FETCH_AND_OP perform atomic read-modify-write and return the data before the accumulate operation; and MPI_COMPARE_AND_SWAP performs a remote atomic compare and swap operation. These operations are *nonblocking*: the call initiates the transfer, but the transfer may continue after the call returns. The transfer is completed, at the origin or both the origin and the target, when a subsequent *synchronization* call is issued by the caller on the involved window object. These synchronization calls are described in Section 12.5. Transfers can also be completed with calls to flush routines; see Section 12.5.4 for details. For the MPI_RPUT, MPI_RGET, MPI_RACCUMULATE, and MPI_RGET_ACCUMULATE calls, the transfer can be locally completed by using the MPI test or wait operations described in Section 3.7.3.

The local communication buffer of an RMA call should not be updated, and the local communication buffer of a get call should not be accessed after the RMA call until the operation completes at the origin.

The resulting data values, or outcome, of concurrent conflicting accesses to the same 46 memory locations is undefined; if a location is updated by a put or accumulate operation, 47 then the outcome of loads or other RMA operations is undefined until the updating operation 48

#### **Unofficial Draft for Comment Only**

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.

¹⁰ For all RMA calls, the target process may be identical with the origin process; i.e., a ¹¹ process may use an RMA operation to move data in its memory.

*Rationale.* The choice of supporting "self-communication" is the same as for messagepassing. 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.

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

30			
31	IN	origin_addr	initial address of origin buffer (choice)
32 33	IN	origin_count	number of entries in origin buffer (non-negative integer)
34	IN	origin_datatype	datatype of each entry in origin buffer (handle)
35 36	IN	target_rank	rank of target (non-negative integer)
37 38	IN	target_disp	displacement from start of window to target buffer (non-negative integer)
39 40	IN	target_count	number of entries in target buffer (non-negative integer)
41 42	IN	target_datatype	datatype of each entry in target buffer (handle)
43	IN	win	window object used for communication (handle)
44			
45	C bind	ing	
46	int MP	I_Put(const void *orig	gin_addr, int origin_count,

```
<sup>47</sup> MPI_Datatype origin_datatype, int target_rank,
```

12

13

14

15 16

21

22

27 28

```
1
 MPI_Aint target_disp, int target_count,
 \mathbf{2}
 MPI_Datatype target_datatype, MPI_Win win)
 3
int MPI_Put_c(const void *origin_addr, MPI_Count origin_count,
 4
 MPI_Datatype origin_datatype, int target_rank,
 5
 MPI_Aint target_disp, MPI_Count target_count,
 6
 MPI_Datatype target_datatype, MPI_Win win)
 7
Fortran 2008 binding
 9
MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
 10
 target_disp, target_count, target_datatype, win, ierror)
 11
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
 INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
 12
 13
 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
 14
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
 15
 TYPE(MPI_Win), INTENT(IN) :: win
 16
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 17
MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
 18
 target_disp, target_count, target_datatype, win, ierror)
 19
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
 20
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
 21
 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
 22
 INTEGER, INTENT(IN) :: target_rank
 23
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
 ^{24}
 TYPE(MPI_Win), INTENT(IN) :: win
 25
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 26
 27
Fortran binding
 28
MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
 29
 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
 30
 <type> ORIGIN_ADDR(*)
 31
 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
 32
 TARGET_DATATYPE, WIN, IERROR
 33
 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
 34
 Transfers origin_count successive entries of the type specified by the origin_datatype,
 35
starting at address origin_addr on the origin node, to the target node specified by the win,
 36
target_rank pair. The data are written in the target buffer at address target_addr =
 37
window_base + target_disp \times disp_unit, where window_base and disp_unit are the base address
 38
and window displacement unit specified at window initialization, by the target process.
 39
```

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 47 communication. The target_datatype may not specify overlapping entries in the target 48

#### **Unofficial Draft for Comment Only**

buffer. The message sent must fit, without truncation, in the target buffer. Furthermore,  $\mathbf{2}$ 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  $\overline{7}$ 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.)
  - A high-quality implementation will attempt to prevent Advice to implementors. remote accesses to memory outside the window that was exposed by the process. This is important both for debugging purposes and for protection with client-server codes that use RMA. That is, a high-quality implementation will check, if possible, window bounds on each RMA call, and raise an error at the origin call if an out-ofbound situation occurs. Note that the condition can be checked at the origin. Of course, the added safety achieved by such checks has to be weighed against the added cost of such checks. (End of advice to implementors.)

12.3.2	Get		1
			2 3
MPI_0	ET(origin_addr, origin_count, target_datatype, wii	origin_datatype, target_rank, target_disp, target_count, 1)	4
OUT	origin_addr	initial address of origin buffer (choice)	6 7
IN	origin_count	number of entries in origin buffer (non-negative integer)	8 9
IN	origin_datatype	datatype of each entry in origin buffer (handle)	10
IN	target_rank	rank of target (non-negative integer)	11 12
IN	target_disp	displacement from window start to the beginning of the target buffer (non-negative integer)	13 14
IN	target_count	number of entries in target buffer (non-negative integer)	15 16
IN	target_datatype	datatype of each entry in target buffer (handle)	17 18
IN	win	window object used for communication (handle)	19
			20
C bin	0		21
int M	PI_Get(void *origin_addr		22 23
		gin_datatype, int target_rank,	23
	-	disp, int target_count,	25
	MPI_Datatype tar	get_datatype, MPI_Win win)	26
int M	_	dr, MPI_Count origin_count,	27
		<pre>gin_datatype, int target_rank,</pre>	28
		disp, MPI_Count target_count,	29
	MPI_Datatype tar	get_datatype, MPI_Win win)	30
Fortra	an 2008 binding		31
MPI_G	et(origin_addr, origin_c	ount, origin_datatype, target_rank,	32
		get_count, target_datatype, win, ierror)	33
		SYNCHRONOUS :: origin_addr	34 35
		igin_count, target_rank, target_count	36
		T(IN) :: origin_datatype, target_datatype	37
	YPE(MPI_Win), INTENT(IN)	KIND), INTENT(IN) :: target_disp	38
	NTEGER, OPTIONAL, INTENT		39
			40
MPI_G		ount, origin_datatype, target_rank,	41
_	<b>U 1</b>	get_count, target_datatype, win, ierror)	42
		SYNCHRONOUS :: origin_addr	43
		ND), INTENT(IN) :: origin_count, target_count T(IN) :: origin_datatype, target_datatype	44
	NTEGER, INTENT(IN) :: ta:		45 46
		KIND), INTENT(IN) :: target_disp	40 47
	YPE(MPI_Win), INTENT(IN)		48

```
1
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
 Fortran binding
3
 MPI_GET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
4
 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
5
 <type> ORIGIN_ADDR(*)
6
 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
7
 TARGET_DATATYPE, WIN, IERROR
8
 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
9
10
 Similar to MPI_PUT, except that the direction of data transfer is reversed. Data
11
 are copied from the target memory to the origin. The origin_datatype may not specify
12
 overlapping entries in the origin buffer. The target buffer must be contained within the
13
 target window or within attached memory in a dynamic window, and the copied data must
14
 fit, without truncation, in the origin buffer.
15
16
 12.3.3 Examples for Communication Calls
17
 These examples show the use of the MPI_GET function. As all MPI RMA communication
18
 functions are nonblocking, they must be completed. In the following, this is accomplished
19
 with the routine MPI_WIN_FENCE, introduced in Section 12.5.
20
21
 Example 12.1 We show how to implement the generic indirect assignment A = B(map),
22
 where A, B, and map have the same distribution, and map is a permutation. To simplify, we
23
 assume a block distribution with equal size blocks.
24
25
 SUBROUTINE MAPVALS(A, B, map, m, comm, p)
26
 USE MPI
27
 INTEGER m, map(m), comm, p
28
 REAL A(m), B(m)
29
30
 INTEGER otype(p), oindex(m),
 & ! used to construct origin datatypes
31
 ttype(p), tindex(m),
 & ! used to construct target datatypes
32
 count(p), total(p),
 &
33
 disp_int, win, ierr
34
 INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint
35
36
 ! This part does the work that depends on the locations of B.
37
 ! Can be reused while this does not change
38
39
 CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
40
 disp_int = realextent
41
 size = m * realextent
42
 CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL,
 &
43
 comm, win, ierr)
44
45
 ! This part does the work that depends on the value of map and
46
 ! the locations of the arrays.
47
 ! Can be reused while these do not change
48
```

```
\mathbf{2}
! Compute number of entries to be received from each process
 3
DO i=1,p
 4
 count(i) = 0
 5
 6
END DO
DO i=1,m
 7
 8
 j = map(i)/m+1
 count(j) = count(j)+1
 9
 10
END DO
 11
total(1) = 0
 12
 13
DO i=2,p
 total(i) = total(i-1) + count(i-1)
 14
 15
END DO
 16
 17
DO i=1,p
 18
 count(i) = 0
 19
END DO
 20
 21
! compute origin and target indices of entries.
! entry i at current process is received from location
 22
 23
! k at process (j-1), where map(i) = (j-1)*m + (k-1),
 ^{24}
! j = 1...p and k = 1...m
 25
 26
DO i=1,m
 j = map(i)/m+1
 27
 k = MOD(map(i), m) + 1
 28
 29
 count(j) = count(j)+1
 30
 oindex(total(j) + count(j)) = i
 tindex(total(j) + count(j)) = k
 31
END DO
 32
 33
 34
! create origin and target datatypes for each get operation
DO i=1,p
 35
 CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, &
 36
 37
 oindex(total(i)+1:total(i)+count(i)), &
 38
 MPI_REAL, otype(i), ierr)
 39
 CALL MPI_TYPE_COMMIT(otype(i), ierr)
 CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, &
 40
 41
 tindex(total(i)+1:total(i)+count(i)), &
 42
 MPI_REAL, ttype(i), ierr)
 CALL MPI_TYPE_COMMIT(ttype(i), ierr)
 43
 44
END DO
 45
 46
! this part does the assignment itself
 47
CALL MPI_WIN_FENCE(0, win, ierr)
 48
disp_aint = 0
```

```
1
 DO i=1,p
\mathbf{2}
 CALL MPI_GET(A, 1, otype(i), i-1, disp_aint, 1, ttype(i), win, ierr)
3
 END DO
4
 CALL MPI_WIN_FENCE(0, win, ierr)
5
6
 CALL MPI_WIN_FREE(win, ierr)
\overline{7}
 DO i=1,p
8
 CALL MPI_TYPE_FREE(otype(i), ierr)
9
 CALL MPI_TYPE_FREE(ttype(i), ierr)
10
 END DO
11
 RETURN
12
 END
13
14
 Example 12.2 A simpler version can be written that does not require that a datatype
15
 be built for the target buffer. But, one then needs a separate get call for each entry, as
16
 illustrated below. This code is much simpler, but usually much less efficient, for large arrays.
17
18
 SUBROUTINE MAPVALS(A, B, map, m, comm, p)
19
 USE MPI
20
 INTEGER m, map(m), comm, p
21
 REAL A(m), B(m)
22
 INTEGER disp_int, win, ierr
23
 INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint
24
25
 CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
26
 disp_int = realextent
27
 size = m * realextent
28
 CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL,
 &
29
 comm, win, ierr)
30
^{31}
 CALL MPI_WIN_FENCE(0, win, ierr)
32
 DO i=1,m
33
 j = map(i)/m
34
 disp_aint = MOD(map(i),m)
35
 CALL MPI_GET(A(i), 1, MPI_REAL, j, disp_aint, 1, MPI_REAL, win, ierr)
36
 END DO
37
 CALL MPI_WIN_FENCE(0, win, ierr)
38
 CALL MPI_WIN_FREE(win, ierr)
39
 RETURN
40
 END
41
42
 12.3.4 Accumulate Functions
43
44
 It is often useful in a put operation to combine the data moved to the target process with the
45
```

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 variable in the memory of one process.

semantics with respect to overlapping data accesses than the put and get functions; see Section 12.7 for details.

#### Accumulate Function

# MPI_ACCUMULATE(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, op, win)

	10.800_000.00, 10.800_0000	-jpe; ep;)	9
IN	origin_addr	initial address of buffer (choice)	10
IN	origin_count	number of entries in buffer (non-negative integer)	11
IN	origin_datatype	datatype of each entry (handle)	12
IN	target_rank	rank of target (non-negative integer)	13 14
IN	target_disp	displacement from start of window to beginning of	15
		target buffer (non-negative integer)	16
IN	target_count	number of entries in target buffer (non-negative	17
		integer)	18 19
IN	target_datatype	datatype of each entry in target buffer (handle)	20
IN	ор	reduce operation (handle)	21
IN	win	window object (handle)	22
			23
binding			24

## C binding

C binding	25
<pre>int MPI_Accumulate(const void *origin_addr, int origin_count,</pre>	26
MPI_Datatype origin_datatype, int target_rank,	27
MPI_Aint target_disp, int target_count,	28
MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)	29
int MPI_Accumulate_c(const 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_Op op, MPI_Win win)	33
	34
Fortran 2008 binding	35
MPI_Accumulate(origin_addr, origin_count, origin_datatype, target_rank,	36
<pre>target_disp, target_count, target_datatype, op, win, ierror)</pre>	37
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr	38
INTEGER, INTENT(IN) :: origin_count, target_rank, target_count	39
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype	40
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp	41
TYPE(MPI_Op), INTENT(IN) :: op	42
TYPE(MPI_Win), INTENT(IN) :: win	43
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	44
MDI Accumulate (anigin addr. anigin count anigin datatuma target work	45
MPI_Accumulate(origin_addr, origin_count, origin_datatype, target_rank,	46
target_disp, target_count, target_datatype, op, win, ierror)	40
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr	
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count	48

 $\mathbf{2}$ 

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_Op), INTENT(IN) :: op
5	TYPE(MPI_Win), INTENT(IN) :: win
6	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7	
8	Fortran binding
9	MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
10	TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
11	<type> ORIGIN_ADDR(*)</type>
12	INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
13	TARGET_DATATYPE, OP, WIN, IERROR
14	INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
15	Accumulate the contents of the origin buffer (as defined by origin_addr, origin_count, and
16	origin_datatype) to the buffer specified by arguments target_count and target_datatype, at
17	
18	offset target_disp, in the target window specified by target_rank and win, using the operation
19	op. This is like MPI_PUT except that data is combined into the target area instead of
20	overwriting it.
21	Any of the predefined operations for MPI_REDUCE can be used. User-defined functions
21	cannot be used. For example, if op is MPI_SUM, each element of the origin buffer is added
22	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
24	all basic components are of the same predefined datatype. Both datatype arguments must
25	be constructed from the same predefined datatype. The operation <b>op</b> applies to elements of
26	that predefined type. The parameter target_datatype must not specify overlapping entries,
27	and the target buffer must fit in the target window.
28	A new predefined operation, MPI_REPLACE, is defined. It corresponds to the associative
29	function $f(a,b) = b$ ; i.e., the current value in the target memory is replaced by the value
30	supplied by the origin.
31	MPI_REPLACE can be used only in MPI_ACCUMULATE, MPI_RACCUMULATE,
32	MPI_GET_ACCUMULATE, MPI_FETCH_AND_OP, and MPI_RGET_ACCUMULATE, but not
33	in collective reduction operations such as MPI_REDUCE.
34	
35	Advice to users. MPI_PUT is a special case of MPI_ACCUMULATE, with the op-
36	eration MPI_REPLACE. Note, however, that MPI_PUT and MPI_ACCUMULATE have
37	different constraints on concurrent updates. (End of advice to users.)
38	
39	<b>Example 19.9</b> We must be example $P(i)$ $\sum P(i)$ The event $A$ $P$ and $m$
40	<b>Example 12.3</b> We want to compute $B(j) = \sum_{map(i)=j} A(i)$ . The arrays A, B, and map
41	are distributed in the same manner. We write the simple version.
42	
43	SUBROUTINE SUM(A, B, map, m, comm, p)
44	USE MPI
45	INTEGER m, map(m), comm, p, win, ierr, disp_int
46	REAL A(m), B(m)
47	INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint
48	

```
CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
size = m * realextent
disp_int = realextent
CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL,
 &
 comm, win, ierr)
CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,m
 j = map(i)/m
 disp_aint = MOD(map(i),m)
 CALL MPI_ACCUMULATE(A(i), 1, MPI_REAL, j, disp_aint, 1, MPI_REAL,
 &
 MPI_SUM, win, ierr)
END DO
CALL MPI_WIN_FENCE(0, win, ierr)
CALL MPI_WIN_FREE(win, ierr)
RETURN
END
```

This code is identical to the code in Example 12.2, except that a call to get has been replaced by a call to accumulate. (Note that, if map is one-to-one, the code computes  $B = A(map^{-1})$ , which is the reverse assignment to the one computed in that previous example.) In a similar manner, we can replace in Example 12.1, the call to get by a call to accumulate, thus performing the computation with only one communication between any two processes.

#### Get Accumulate Function

It is often useful to have fetch-and-accumulate semantics such that the remote data is returned to the caller before the sent data is accumulated into the remote data. The get and accumulate steps are executed atomically for each basic element in the datatype (see Section 12.7 for details). The predefined operation MPI_REPLACE provides fetch-and-set behavior.

 $\mathbf{2}$ 

 24 

 31 

```
1
 MPI_GET_ACCUMULATE(origin_addr, origin_count, origin_datatype, result_addr,
\mathbf{2}
 result_count, result_datatype, target_rank, target_disp, target_count,
3
 target_datatype, op, win)
4
 IN
 origin_addr
 initial address of buffer (choice)
5
 origin_count
 IN
 number of entries in origin buffer (non-negative
6
 integer)
7
8
 IN
 origin_datatype
 datatype of each entry in origin buffer (handle)
9
 OUT
 result_addr
 initial address of result buffer (choice)
10
 result_count
 number of entries in result buffer (non-negative
 IN
11
 integer)
12
 IN
 result_datatype
 datatype of each entry in result buffer (handle)
13
14
 IN
 target_rank
 rank of target (non-negative integer)
15
 IN
 target_disp
 displacement from start of window to beginning of
16
 target buffer (non-negative integer)
17
 IN
 target_count
 number of entries in target buffer (non-negative
18
 integer)
19
20
 IN
 target_datatype
 datatype of each entry in target buffer (handle)
21
 IN
 reduce operation (handle)
 ор
22
 IN
 win
 window object (handle)
23
24
25
 C binding
26
 int MPI_Get_accumulate(const void *origin_addr, int origin_count,
27
 MPI_Datatype origin_datatype, void *result_addr,
28
 int result_count, MPI_Datatype result_datatype,
 int target_rank, MPI_Aint target_disp, int target_count,
29
30
 MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
^{31}
 int MPI_Get_accumulate_c(const void *origin_addr, MPI_Count origin_count,
32
 MPI_Datatype origin_datatype, void *result_addr,
33
 MPI_Count result_count, MPI_Datatype result_datatype,
34
 int target_rank, MPI_Aint target_disp, MPI_Count target_count,
35
 MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
36
37
 Fortran 2008 binding
38
 MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
39
 result_count, result_datatype, target_rank, target_disp,
40
 target_count, target_datatype, op, win, ierror)
41
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
42
 INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
43
 target_count
44
 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
45
 target_datatype
46
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
47
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
48
 TYPE(MPI_Op), INTENT(IN) :: op
```

```
1
 TYPE(MPI_Win), INTENT(IN) :: win
 2
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 3
MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
 4
 result_count, result_datatype, target_rank, target_disp,
 5
 target_count, target_datatype, op, win, ierror)
 6
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
 7
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, result_count,
 8
 target_count
 9
 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
 10
 target_datatype
 11
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
 12
 INTEGER, INTENT(IN) :: target_rank
 13
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
 14
 TYPE(MPI_Op), INTENT(IN) :: op
 15
 TYPE(MPI_Win), INTENT(IN) :: win
 16
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 17
 18
Fortran binding
 19
MPI_GET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_ADDR,
 RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, TARGET_DISP,
 20
 21
 TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
 22
 <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
 23
 ^{24}
 TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR
 25
 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
 26
```

Accumulate origin_count elements of type origin_datatype from the origin buffer ( origin_addr) to the buffer at offset target_disp, in the target window specified by target_rank and win, using the operation op and return in the result buffer result_addr the content of the target buffer before the accumulation, specified by target_disp, target_count, and target_datatype. The data transferred from origin to target must fit, without truncation, in the target buffer. Likewise, the data copied from target to origin must fit, without truncation, in the result buffer.

The origin and result buffers (origin_addr and result_addr) must be disjoint. Each datatype argument must be a predefined datatype or a derived datatype where all basic components are of the same predefined datatype. All datatype arguments must be constructed from the same predefined datatype. The operation op applies to elements of that predefined type. target_datatype must not specify overlapping entries, and the target buffer must fit in the target window or in attached memory in a dynamic window. The operation is executed atomically for each basic datatype; see Section 12.7 for details.

Any of the predefined operations for MPI_REDUCE, as well as MPI_NO_OP or MPI_REPLACE can be specified as op. User-defined functions cannot be used. A new predefined operation, MPI_NO_OP, is defined. It corresponds to the associative function f(a, b) = a; i.e., the current value in the target memory is returned in the result buffer at the origin and no operation is performed on the target buffer. When MPI_NO_OP is specified as the operation, the origin_addr, origin_count, and origin_datatype arguments are ignored. MPI_NO_OP can be used only in MPI_GET_ACCUMULATE, MPI_RGET_ACCUMULATE, 41 42 42 42 43 44 44 45 4647

27

28

29

30

31

32

33

34

35

36

37

38

39

40

	576	СН	APTER 12. ONE-SIDED COMMUNICATIONS					
1 2 3	and MPI_FETCH_AND_OP. MPI_NO_OP cannot be used in MPI_ACCUMULATE, MPI_RACCUMULATE, or collective reduction operations, such as MPI_REDUCE and others.							
4 5 6 7	Advice to users. MPI_GET is similar to MPI_GET_ACCUMULATE, with the opera- tion MPI_NO_OP. Note, however, that MPI_GET and MPI_GET_ACCUMULATE have different constraints on concurrent updates. ( <i>End of advice to users.</i> )							
8	Fetch and C	Op Function						
9 10 11 12 13 14	The generic functionality of MPI_GET_ACCUMULATE might limit the performance of fetch- and-increment or fetch-and-add calls that might be supported by special hardware oper- ations. MPI_FETCH_AND_OP thus allows for a fast implementation of a commonly used subset of the functionality of MPI_GET_ACCUMULATE.							
15 16	MPI_FETC	H_AND_OP(origin_addr, result	t_addr, datatype, target_rank, target_disp, op, win)					
17								
18	IN	origin_addr	initial address of buffer (choice)					
19 20	OUT	result_addr	initial address of result buffer (choice)					
21 22	IN	datatype	datatype of the entry in origin, result, and target buffers (handle)					
23	IN	target_rank	rank of target (non-negative integer)					
24 25	IN	target_disp	displacement from start of window to beginning of target buffer (non-negative integer)					
26 27	IN	ор	reduce operation (handle)					
28	IN	win	window object (handle)					
29 30 31 32 33	C binding int MPI_F	etch_and_op(const void *o	rigin_addr, void *result_addr, e, int target_rank, MPI_Aint target_disp, in)					
34	Fortran 2	008 binding						
35 36 37	<pre>MPI_Fetch_and_op(origin_addr, result_addr, datatype, target_rank, target_disp, op, win, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr</pre>							
38 39	TYPE(*), DIMENSION(), ASYNCHRONOUS :: result_addr							
40	TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(IN) :: target_rank							
41	INTEGER, INTENI(IN) :: target_rank INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp							
42 43		MPI_Op), INTENT(IN) :: op						
43 44 45		MPI_Win), INTENT(IN) :: w ER, OPTIONAL, INTENT(OUT)						
45 46	Fortran b	inding						
47		•	LT_ADDR, DATATYPE, TARGET_RANK,					
48	TARGET_DISP, OP, WIN, IERROR)							

			1		
<pre><type> ORIGIN_ADDR(*), RESULT_ADDR(*) INTEGED_DATATYDETARGET_DANK_ODUINIEDDOD</type></pre>					
INTEGER DATATYPE, TARGET_RANK, OP, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP					
		atatype from the origin buffer (origin_addr) to the	5		
	• • • •	et window specified by target_rank and win, using	6		
-		t buffer result_addr the content of the target buffer	7		
	accumulation.		8		
		_addr and result_addr) must be disjoint. Any of the	9		
-	-	annot be used. The datatype argument must be a	10		
-	datatype. The operation is ex		11		
predenned	datatype. The operation is ea	xecuted atomicany.	12		
Compare a	nd Swap Function		13		
			14		
		ompare and swap where the value at the origin is	15 16		
-		nich is atomically replaced by a third value only if	10		
the values	at origin and target are equal		18		
			19		
MPI_COM	PARE_AND_SWAP(origin_add	r, compare_addr, result_addr, datatype,	20		
	target_rank, target_disp,	win)	21		
IN	origin_addr	initial address of buffer (choice)	22		
IN	compare_addr	initial address of compare buffer (choice)	23		
OUT	result_addr	initial address of result buffer (choice)	24 25		
			26		
IN	datatype	datatype of the element in all buffers (handle)	27		
IN	target_rank	rank of target (non-negative integer)	28		
IN	target_disp	displacement from start of window to beginning of target buffer (non-negative integer)	29 30		
IN	win	window object (handle)	31		
			32		
C binding	g		33		
int MPI_C		id *origin_addr, const void *compare_addr,	34 35		
	void *result_addr, M	PI_Datatype datatype, int target_rank,	36		
	MPI_Aint target_disp	, MPI_Win win)	37		
Fortran 2	2008 binding		38		
		compare_addr, result_addr, datatype,	39		
-	target_rank, target_		40		
TYPE(	*), DIMENSION(), INTENT	Γ(IN), ASYNCHRONOUS :: origin_addr,	41		
	compare_addr		42		
	(*), DIMENSION(), ASYNCH		43		
	MPI_Datatype), INTENT(IN)		44		
	ER, INTENT(IN) :: target_		45 46		
	ER(KIND=MP1_ADDRESS_KIND) MPI_Win), INTENT(IN) :: w	), INTENT(IN) :: target_disp	46 47		
			48		
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 48					

Unofficial Draft for Comment Only

1	Fortran binding
2	MPI_COMPARE_AND_SWAP(ORIGIN_ADDR, COMPARE_ADDR, RESULT_ADDR, DATATYPE,
3	TARGET_RANK, TARGET_DISP, WIN, IERROR)
4	<type> ORIGIN_ADDR(*), COMPARE_ADDR(*), RESULT_ADDR(*)</type>
5	INTEGER DATATYPE, TARGET_RANK, WIN, IERROR
6	INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP

7 This function compares one element of type datatype in the compare buffer 8 compare_addr with the buffer at offset target_disp in the target window specified by 9 target_rank and win and replaces the value at the target with the value in the origin buffer 10 origin_addr if the compare buffer and the target buffer are identical. The original value at 11 the target is returned in the buffer result_addr. The parameter datatype must belong to 12one of the following categories of predefined datatypes: C integer, Fortran integer, Logical, 13 Multi-language types, or Byte as specified in Section 6.9.2. The origin and result buffers 14(origin_addr and result_addr) must be disjoint. 15

16

#### 17 12.3.5 Request-based RMA Communication Operations

¹⁸ Request-based RMA communication operations allow the user to associate a request handle ¹⁹ with the RMA operations and test or wait for the completion of these requests using the ²⁰ functions described in Section 3.7.3. Request-based RMA operations are only valid within ²¹ a passive target epoch (see Section 12.5).

²² Upon returning from a completion call in which an RMA operation completes, all fields ²³ of the status object, if any, and the results of status query functions (e.g.,

²⁴ MPI_GET_COUNT) are undefined with the exception of MPI_ERROR if appropriate (see ²⁵ Section 3.2.5). It is valid to mix different request types (e.g., any combination of RMA ²⁶ requests, collective requests, I/O requests, generalized requests, or point-to-point requests) ²⁷ in functions that enable multiple completions (e.g., MPI_WAITALL). It is erroneous to call ²⁸ MPI_REQUEST_FREE or MPI_CANCEL for a request associated with an RMA operation. ²⁹ RMA requests are not persistent.

³⁰ The end of the epoch, or explicit bulk synchronization using

³¹ MPI_WIN_FLUSH, MPI_WIN_FLUSH_ALL, MPI_WIN_FLUSH_LOCAL, or

MPI_WIN_FLUSH_LOCAL_ALL, also indicates completion of the RMA operations. How ever, users must still wait or test on the request handle to allow the MPI implementation to
 clean up any resources associated with these requests; in such cases the wait operation will
 complete locally.

MPI_RPU	T(origin_addr, origin_count, or target_count, target_dat	igin_datatype, target_rank, target_disp, atype, win, request)	$\frac{1}{2}$			
IN	origin_addr	initial address of origin buffer (choice)	3			
IN	origin_count	number of entries in origin buffer (non-negative integer)	4 5 6			
IN	origin_datatype	datatype of each entry in origin buffer (handle)	7			
IN	target_rank	rank of target (non-negative integer)	8 9			
IN	target_disp	displacement from start of window to target buffer (non-negative integer)	9 10 11			
IN	target_count	number of entries in target buffer (non-negative integer)	12 13			
IN	target_datatype	datatype of each entry in target buffer (handle)	14			
IN	win	window object used for communication (handle)	15 16			
OUT	request	RMA request (handle)	17			
			18			
C bindin	g		19			
int MPI_	Rput(const void *origin_a	J	20 21			
		datatype, int target_rank,	21			
	MPI_Aint target_disp		23			
		_datatype, MPI_Win win,	24			
MPI_Request *request)						
int MPI_Rput_c(const void *origin_addr, MPI_Count origin_count,						
	MPI_Datatype origin_	datatype, int target_rank,	27			
		o, MPI_Count target_count,	28			
		datatype, MPI_Win win,	29			
	MPI_Request *request		30			
Fortran	2008 binding		31			
	5	t, origin_datatype, target_rank,	32			
-		_count, target_datatype, win, request,	33			
	ierror)		34			
TYPE	(*), DIMENSION(), INTEN	T(IN), ASYNCHRONOUS :: origin_addr	35 36			
	Ū.	_count, target_rank, target_count	37			
		) :: origin_datatype, target_datatype	38			
		), INTENT(IN) :: target_disp	39			
	(MPI_Win), INTENT(IN) ::		40			
	(MPI_Request), INTENT(OUT GER, OPTIONAL, INTENT(OUT		41			
	GER, OPIIONAL, INIENI(UUI		42			
MPI_Rput	(origin_addr, origin_coun	t, origin_datatype, target_rank,	43			
		count, target_datatype, win, request,	44			
	ierror)	- /	45			
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr       46						
	<pre>INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count 47 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype 48</pre>					

1 2 3 4 5 6	<pre>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</pre>					
7 8 9 10 11 12 13 14 15	<pre>Fortran binding MPI_RPUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,</pre>					
16 17 18 19 20 21 22 23	MPI_RPUT is similar to MPI_PUT (Section 12.3.1), except that it allocates a commu- nication 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) in- dicates 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 re- quired, MPI_WIN_FLUSH, MPI_WIN_FLUSH_ALL, MPI_WIN_UNLOCK, or MPI_WIN_UNLOCK_ALL can be used.					
24 25	MPI_RGE		nt, origin_datatype, target_rank, target_disp, et_datatype, win, request)			
26 27	OUT	origin_addr	initial address of origin buffer (choice)			
28 29	IN	origin_count	number of entries in origin buffer (non-negative integer)			
30	IN	origin_datatype	datatype of each entry in origin buffer (handle)			
31	IN	target_rank	rank of target (non-negative integer)			
32 33 34	IN	target_disp	displacement from window start to the beginning of the target buffer (non-negative integer)			
35 36	IN	target_count	number of entries in target buffer (non-negative integer)			
37	IN	target_datatype	datatype of each entry in target buffer (handle)			
$\frac{38}{39}$	IN	win	window object used for communication (handle)			
40	OUT	request	RMA request (handle)			
41						
42 43	C bindin	•	la int onigin count			
43	IIIC MPI_	• •	<pre>lr, int origin_count, igin_datatype, int target_rank,</pre>			
45		• •	_disp, int target_count,			
46			rget_datatype, MPI_Win win,			
47		MPI_Request *re	quest)			
48						

```
1
int MPI_Rget_c(void *origin_addr, MPI_Count origin_count,
 \mathbf{2}
 MPI_Datatype origin_datatype, int target_rank,
 3
 MPI_Aint target_disp, MPI_Count target_count,
 MPI_Datatype target_datatype, MPI_Win win,
 4
 MPI_Request *request)
 5
 6
Fortran 2008 binding
MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank,
 target_disp, target_count, target_datatype, win, request,
 9
 ierror)
 10
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: origin_addr
 11
 INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
 12
 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
 13
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
 14
 TYPE(MPI_Win), INTENT(IN) :: win
 15
 TYPE(MPI_Request), INTENT(OUT) :: request
 16
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 17
 18
MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank,
 19
 target_disp, target_count, target_datatype, win, request,
 20
 ierror)
 21
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: origin_addr
 22
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
 23
 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
 24
 INTEGER, INTENT(IN) :: target_rank
 25
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
 26
 TYPE(MPI_Win), INTENT(IN) :: win
 27
 TYPE(MPI_Request), INTENT(OUT) :: request
 28
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 29
Fortran binding
 30
MPI_RGET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
 31
 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST,
 32
 IERROR)
 33
 <type> ORIGIN_ADDR(*)
 34
 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
 35
 TARGET_DATATYPE, WIN, REQUEST, IERROR
 36
 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
 37
 38
 MPI_RGET is similar to MPI_GET (Section 12.3.2), except that it allocates a commu-
 39
nication request object and associates it with the request handle (the argument request)
```

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.

> 44 45

40

41

42

43

```
582
 CHAPTER 12. ONE-SIDED COMMUNICATIONS
1
 MPI_RACCUMULATE(origin_addr, origin_count, origin_datatype, target_rank, target_disp,
\mathbf{2}
 target_count, target_datatype, op, win, request)
3
 IN
 origin_addr
 initial address of buffer (choice)
4
 IN
 origin_count
 number of entries in buffer (non-negative integer)
5
6
 origin_datatype
 IN
 datatype of each entry in origin buffer (handle)
7
 IN
 target_rank
 rank of target (non-negative integer)
8
 IN
 target_disp
 displacement from start of window to beginning of
9
 target buffer (non-negative integer)
10
11
 IN
 number of entries in target buffer (non-negative
 target_count
12
 integer)
13
 IN
 target_datatype
 datatype of each entry in target buffer (handle)
14
 reduce operation (handle)
 IN
 ор
15
16
 IN
 win
 window object (handle)
17
 OUT
 RMA request (handle)
 request
18
19
 C binding
20
 int MPI_Raccumulate(const void *origin_addr, int origin_count,
21
 MPI_Datatype origin_datatype, int target_rank,
22
 MPI_Aint target_disp, int target_count,
23
 MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
24
 MPI_Request *request)
25
26
 int MPI_Raccumulate_c(const void *origin_addr, MPI_Count origin_count,
 MPI_Datatype origin_datatype, int target_rank,
27
 MPI_Aint target_disp, MPI_Count target_count,
28
 MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
29
30
 MPI_Request *request)
^{31}
 Fortran 2008 binding
32
 MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
33
 target_disp, target_count, target_datatype, op, win, request,
34
 ierror)
35
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
36
 INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
37
 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
38
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
39
 TYPE(MPI_Op), INTENT(IN) :: op
40
 TYPE(MPI_Win), INTENT(IN) :: win
41
 TYPE(MPI_Request), INTENT(OUT) :: request
42
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44
 MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
45
 target_disp, target_count, target_datatype, op, win, request,
46
 ierror)
47
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
48
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
```

TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype 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 TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror

#### Fortran binding

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.

```
1
 MPI_RGET_ACCUMULATE(origin_addr, origin_count, origin_datatype, result_addr,
\mathbf{2}
 result_count, result_datatype, target_rank, target_disp, target_count,
3
 target_datatype, op, win, request)
4
 IN
 origin_addr
 initial address of buffer (choice)
5
 IN
 origin_count
 number of entries in origin buffer (non-negative
6
 integer)
7
8
 IN
 origin_datatype
 datatype of each entry in origin buffer (handle)
9
 OUT
 result addr
 initial address of result buffer (choice)
10
 result_count
 number of entries in result buffer (non-negative
 IN
11
 integer)
12
13
 IN
 result_datatype
 datatype of entries in result buffer (handle)
14
 IN
 target_rank
 rank of target (non-negative integer)
15
 IN
 target_disp
 displacement from start of window to beginning of
16
 target buffer (non-negative integer)
17
18
 number of entries in target buffer (non-negative
 IN
 target_count
19
 integer)
20
 IN
 target_datatype
 datatype of each entry in target buffer (handle)
21
 IN
 ор
 reduce operation (handle)
22
23
 IN
 window object (handle)
 win
24
 RMA request (handle)
 OUT
 request
25
26
 C binding
27
 int MPI_Rget_accumulate(const void *origin_addr, int origin_count,
28
 MPI_Datatype origin_datatype, void *result_addr,
29
 int result_count, MPI_Datatype result_datatype,
30
 int target_rank, MPI_Aint target_disp, int target_count,
^{31}
 MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
32
 MPI_Request *request)
33
34
 int MPI_Rget_accumulate_c(const void *origin_addr, MPI_Count origin_count,
35
 MPI_Datatype origin_datatype, void *result_addr,
36
 MPI_Count result_count, MPI_Datatype result_datatype,
37
 int target_rank, MPI_Aint target_disp, MPI_Count target_count,
38
 MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
39
 MPI_Request *request)
40
 Fortran 2008 binding
41
 MPI_Rget_accumulate(origin_addr, origin_count, origin_datatype,
42
 result_addr, result_count, result_datatype, target_rank,
43
 target_disp, target_count, target_datatype, op, win, request,
44
 ierror)
45
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
46
 INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
47
 target_count
```

```
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
 1
 2
 target_datatype
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
 TYPE(MPI_Op), INTENT(IN) :: op
 5
 TYPE(MPI_Win), INTENT(IN) :: win
 6
 TYPE(MPI_Request), INTENT(OUT) :: request
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Rget_accumulate(origin_addr, origin_count, origin_datatype,
 10
 result_addr, result_count, result_datatype, target_rank,
 11
 target_disp, target_count, target_datatype, op, win, request,
 12
 ierror)
 13
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
 14
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, result_count,
 15
 target_count
 16
 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
 17
 target_datatype
 18
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
 19
 INTEGER, INTENT(IN) :: target_rank
 20
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
 21
 TYPE(MPI_Op), INTENT(IN) :: op
 22
 TYPE(MPI_Win), INTENT(IN) :: win
 23
 TYPE(MPI_Request), INTENT(OUT) :: request
 24
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 25
 26
Fortran binding
MPI_RGET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE,
 27
 28
 RESULT_ADDR, RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK,
 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
 29
 30
 IERROR)
 31
 <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
 32
 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
 33
 TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
 34
 IERROR
 35
 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
 36
```

MPI_RGET_ACCUMULATE is similar to MPI_GET_ACCUMULATE (Section 12.3.4), except that it allocates a communication request object and associates it with the request handle (the argument request) that can be used to wait or test for completion. The completion of an MPI_RGET_ACCUMULATE operation indicates that the data is available in the result buffer and the origin buffer is free to be updated. It does not indicate that the operation has been completed at the target window.

# 12.4 Memory Model

The memory semantics of RMA are best understood by using the concept of *public* and private window copies. We assume that systems have a public memory region that is 47 addressable by all processes (e.g., the shared memory in shared memory machines or the 48

# Unofficial Draft for Comment Only

37

38

39

40

41

42 43

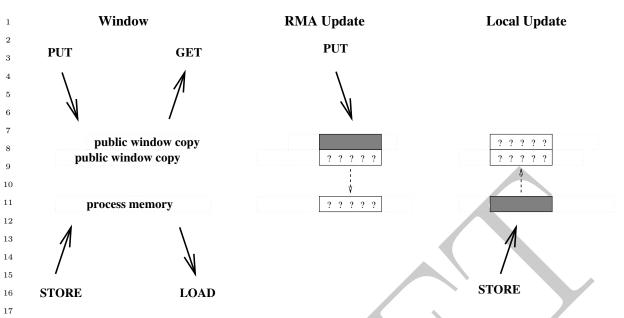


Figure 12.1: Schematic description of the public/private window operations in the MPI_WIN_SEPARATE memory model for two overlapping windows

2021exposed main memory in distributed memory machines). In addition, most machines have 22fast private buffers (e.g., transparent caches or explicit communication buffers) local to 23each process where copies of data elements from the main memory can be stored for faster  24 access. Such buffers are either coherent, i.e., all updates to main memory are reflected in 25all private copies consistently, or non-coherent, i.e., conflicting accesses to main memory 26need to be synchronized and updated in all private copies explicitly. Coherent systems 27allow direct updates to remote memory without any participation of the remote side. Non-28coherent systems, however, need to call RMA functions in order to reflect updates to the 29public window in their private memory. Thus, in coherent memory, the public and the 30 private window are identical while they remain logically separate in the non-coherent case.  31 MPI thus differentiates between two memory models called **RMA unified**, if public and 32 private window are logically identical, and **RMA** separate, otherwise. 33

In the RMA separate model, there is only one instance of each variable in process 34memory, but a distinct *public* copy of the variable for each window that contains it. A load 35 accesses the instance in process memory (this includes MPI sends). A local store accesses 36 and updates the instance in process memory (this includes MPI receives), but the update 37 may affect other public copies of the same locations. A get on a window accesses the public 38 copy of that window. A put or accumulate on a window accesses and updates the public 39 copy of that window, but the update may affect the private copy of the same locations 40in process memory, and public copies of other overlapping windows. This is illustrated in 41 Figure **12.1**.

⁴² In the RMA unified model, public and private copies are identical and updates via put ⁴³ or accumulate calls are eventually observed by load operations without additional RMA ⁴⁴ calls. A store access to a window is eventually visible to remote get or accumulate calls ⁴⁵ without additional RMA calls. These stronger semantics of the RMA unified model allow ⁴⁶ the user to omit some synchronization calls and potentially improve performance.

47 48

18

19

Advice to users. If accesses in the RMA unified model are not synchronized (with

#### Unofficial Draft for Comment Only

locks or flushes, see Section 12.5.3), load and store operations might observe changes to the memory while they are in progress. The order in which data is written is not specified unless further synchronization is used. This might lead to inconsistent views on memory and programs that assume that a transfer is complete by only checking parts of the message are erroneous. (*End of advice to users.*)

The memory model for a particular RMA window can be determined by accessing the attribute MPI_WIN_MODEL. If the memory model is the unified model, the value of this attribute is MPI_WIN_UNIFIED; otherwise, the value is MPI_WIN_SEPARATE.

# 12.5 Synchronization Calls

RMA communications fall in two categories:

- active target communication, where data is moved from the memory of one process to the memory of another, and both are explicitly involved in the communication. This communication pattern is similar to message passing, except that all the data transfer arguments are provided by one process, and the second process only participates in the synchronization.
- passive target communication, where data is moved from the memory of one process to the memory of another, and only the origin process is explicitly involved in the transfer. Thus, two origin processes may communicate by accessing the same location in a target window. The process that owns the target window may be distinct from the two communicating processes, in which case it does not participate explicitly in the communication. This communication paradigm is closest to a shared memory model, where shared data can be accessed by all processes, irrespective of location.

RMA communication calls with argument win must occur at a process only within an **access epoch** for win. Such an epoch starts with an RMA synchronization call on win; it proceeds with zero or more RMA communication calls (e.g., MPI_PUT, MPI_GET or MPI_ACCUMULATE) on win; it completes with another synchronization call on win. This allows users to amortize one synchronization with multiple data transfers and provide implementors more flexibility in the implementation of RMA operations.

Distinct access epochs for win at the same process must be disjoint. On the other hand, epochs pertaining to different win arguments may overlap. Local operations or other MPI calls may also occur during an epoch.

In active target communication, a target window can be accessed by RMA operations only within an **exposure epoch**. Such an epoch is started and completed by RMA synchronization calls executed by the target process. Distinct exposure epochs at a process on the same window must be disjoint, but such an exposure epoch may overlap with exposure epochs on other windows or with access epochs for the same or other win arguments. There is a one-to-one matching between access epochs at origin processes and exposure epochs on target processes: RMA operations issued by an origin process for a target window will access that target window during the same exposure epoch if and only if they were issued during the same access epoch.

In passive target communication the target process does not execute RMA synchronization calls, and there is no concept of an exposure epoch.

MPI provides three synchronization mechanisms:

#### **Unofficial Draft for Comment Only**

1 2

3

4

5

6

 $\overline{7}$ 

8

9 10 11

12 13

14

15

16

17

18

19 20

21

22

23

 24 

25

26

27

28 29

30

 31 

32 33

34

35

36 37

38

39

40 41

42

43 44

45

46

47

- 1. The MPI_WIN_FENCE collective synchronization call supports a simple 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.
- ²⁷ 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.

37 Figure 12.2 illustrates the general synchronization pattern for active target communi-38 cation. The synchronization between **post** and **start** ensures that the put call of the origin 39 process does not start until the target process exposes the window (with the **post** call); 40the target process will expose the window only after preceding local accesses to the window 41 have completed. The synchronization between complete and wait ensures that the put call 42of the origin process completes before the window is unexposed (with the wait call). The 43 target process will execute following local accesses to the target window only after the wait 44returned.

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 However, such strong synchronization is more than needed for

1

2

3

4

5

6

7

8

9

10

11

12

28

29 30

31

32

33

34

35

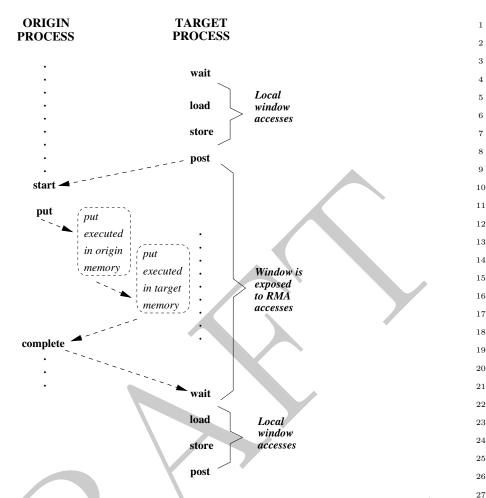


Figure 12.2: Active target communication. Dashed arrows represent synchronizations (ordering of events).

correct ordering of window accesses. The semantics of MPI calls allow **weak synchroniza-tion**, as illustrated in Figure 12.3. The access to the target window is delayed until the window is exposed, after the **post**. However the **start** may complete earlier; the **put** and **complete** may also terminate earlier, if put data is buffered by the implementation. The synchronization calls order correctly window accesses, but do not necessarily synchronize other operations. This weaker synchronization semantic allows for more efficient implementations.

Figure 12.4 illustrates the general synchronization pattern for passive target communication. The first origin process communicates data to the second origin process, through the memory of the target process; the target process is not explicitly involved in the communication. The lock and unlock calls ensure that the two RMA accesses do not occur concurrently. However, they do *not* ensure that the put by origin 1 will precede the get by origin 2.

*Rationale.* RMA does not define fine-grained mutexes in memory (only logical coarsegrained process locks). MPI provides the primitives (compare and swap, accumulate, send/receive, etc.) needed to implement high-level synchronization operations. (*End of rationale.*)

#### Unofficial Draft for Comment Only

28

29 30

 31 

32

33

34

35

36

37

38

39

40

41

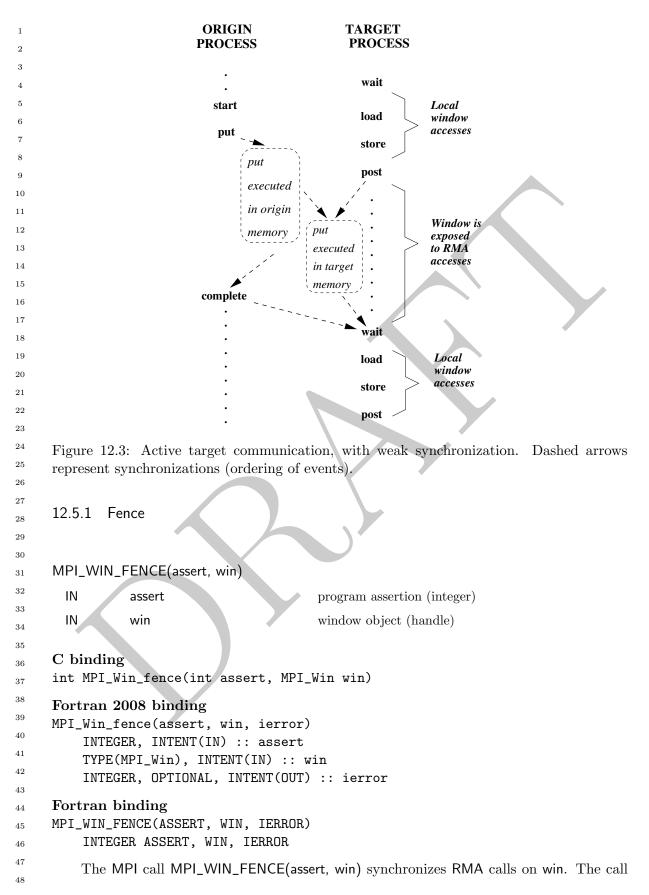
42

43 44

45

46

47



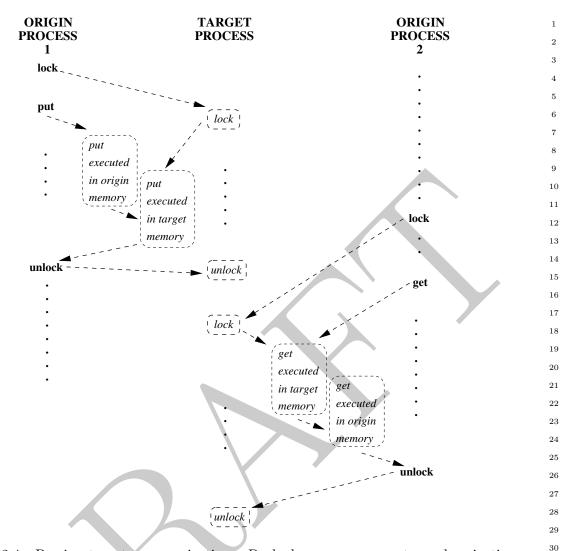


Figure 12.4: Passive target communication. Dashed arrows represent synchronizations (ordering of events).

is collective on the group of win. All RMA operations on win originating at a given process and started before the fence call will complete at that process before the fence call returns. They will be completed at their target before the fence call returns at the target. RMA operations on win started by a process after the fence call returns will access their target window only after MPI_WIN_FENCE has been called by the target process.

The call completes an RMA access epoch if it was preceded by another fence call and 39 the local process issued RMA communication calls on win between these two calls. The call 40 completes an RMA exposure epoch if it was preceded by another fence call and the local 41 window was the target of RMA accesses between these two calls. The call starts an RMA 42access epoch if it is followed by another fence call and by RMA communication calls issued 43 between these two fence calls. The call starts an exposure epoch if it is followed by another 44fence call and the local window is the target of RMA accesses between these two fence calls. 45Thus, the fence call is equivalent to calls to a subset of post, start, complete, wait. 46

A fence call usually entails a barrier synchronization: a process completes a call to MPI_WIN_FENCE only after all other processes in the group entered their matching call.

#### Unofficial Draft for Comment Only

 31 

32 33

34

35

36

37

# CHAPTER 12. ONE-SIDED COMMUNICATIONS

```
1
 However, a call to MPI_WIN_FENCE that is known not to end any epoch (in particular, a
\mathbf{2}
 call with assert equal to MPI_MODE_NOPRECEDE) does not necessarily act as a barrier.
3
 The assert argument is used to provide assertions on the context of the call that may
4
 be used for various optimizations. This is described in Section 12.5.5. A value of assert =
5
 0 is always valid.
6
 Advice to users. Calls to MPI_WIN_FENCE should both precede and follow calls to
7
 RMA communication functions that are synchronized with fence calls. (End of advice
8
 to users.)
9
10
 12.5.2 General Active Target Synchronization
11
12
13
14
 MPI_WIN_START(group, assert, win)
15
 IN
 group of target processes (handle)
 group
16
 IN
 assert
 program assertion (integer)
17
18
 IN
 window object (handle)
 win
19
20
 C binding
21
 int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
22
 Fortran 2008 binding
23
 MPI_Win_start(group, assert, win, ierror)
24
 TYPE(MPI_Group), INTENT(IN) :: group
25
 INTEGER, INTENT(IN) :: assert
26
 TYPE(MPI_Win), INTENT(IN) :: win
27
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
 Fortran binding
30
 MPI_WIN_START(GROUP, ASSERT, WIN, IERROR)
^{31}
 INTEGER GROUP, ASSERT, WIN, IERROR
32
 Starts an RMA access epoch for win. RMA calls issued on win during this epoch must
33
 access only windows at processes in group. Each process in group must issue a matching
34
 call to MPI_WIN_POST. RMA accesses to each target window will be delayed, if necessary,
35
 until the target process executed the matching call to MPI_WIN_POST. MPI_WIN_START
36
 is allowed to block until the corresponding MPI_WIN_POST calls are executed, but is not
37
 required to.
38
 The assert argument is used to provide assertions on the context of the call that may
39
 be used for various optimizations. This is described in Section 12.5.5. A value of assert =
40
 0 is always valid.
41
42
43
 MPI_WIN_COMPLETE(win)
44
 IN
 window object (handle)
 win
45
46
47
 C binding
48
 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_START is nonblocking, but the call to MPI_PUT blocks until the matching call to MPI_WIN_POST occurs; or implementations where the first two calls are nonblocking, but the call to MPI_WIN_COMPLETE blocks until the call to MPI_WIN_POST occurred; or even implementations where all three calls can complete before any target process has called MPI_WIN_POST—the data put must be buffered, in this last case, so as to allow the put to complete at the origin ahead of its completion at the target. However, once the call to MPI_WIN_POST is issued, the sequence above must complete, without further dependencies.

MPI_WIN_POST(group, assert, win)

IN	group	group of origin processes (handle)
IN	assert	program assertion (integer)
IN	win	window object (handle)

# C binding int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)

Fortran 2008 binding
MPI_Win_post(group, assert, win, ierror)
 TYPE(MPI_Group), INTENT(IN) :: group

Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

5

6

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

44

45

46

47

1	INTEGER, INTENT(IN) :: assert					
2	TYPE(MPI_Win), INTENT(IN) :: win					
3	INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
4	Fortran b	oinding				
5 6		•	SSERT, WIN, IERROR)			
			SERT, WIN, IERROR			
7 8						
9		-	sure epoch for the local window associated with win. Only processes			
10	• •		e window with RMA calls on win during this epoch. Each process			
11	in group m	ust issue a mat	tching call to MPI_WIN_START. MPI_WIN_POST does not block.			
12						
13	MPI_WIN_	WAIT(win)				
14			min dame abient (bere die)			
15	IN	win	window object (handle)			
16	~					
17	C binding	-				
18	int MPI_W	in_wait(MPI_	Win win)			
19	Fortran 2	008 binding				
20	MPI_Win_w	ait(win, ier	ror)			
21	TYPE(	MPI_Win), IN	TENT(IN) :: win			
22	INTEG	ER, OPTIONAL	., INTENT(OUT) :: ierror			
23	Fortran b	inding				
24		AIT(WIN, IER				
25		ER WIN, IERR				
26						
27	-		exposure epoch started by a call to MPI_WIN_POST on win. This			
28			_WIN_COMPLETE(win) issued by each of the origin processes that			
29	-		e window during this epoch. The call to MPI_WIN_WAIT will block			
30		-	o MPI_WIN_COMPLETE have occurred. This guarantees that all			
31 32	0		ve completed their RMA accesses to the local window. When the A accesses will have completed at the target window.			
33			es the use of these four functions. Process 0 puts data in the			
34		· · · · · · · · · · · · · · · · · · ·	and 2 and process 3 puts data in the window of process 2. Each			
35		-	f the processes whose windows will be accessed; each post call lists			
36			es that access the local window. The figure illustrates a possible			
37			iming strong synchronization; in a weak synchronization, the start,			
38	-		y occur ahead of the matching post calls.			
39	r	r	O I			
40						
41	MPI_WIN_	TEST(win, flag	g)			
42	IN	win	window object (handle)			
43	OUT	flag	success flag (logical)			
44		- 0				
45	C binding	<b>y</b>				
46		-	Win win, int *flag)			
47			,0,			
48						

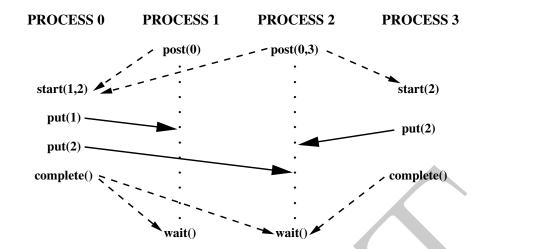


Figure 12.5: Active target communication. Dashed arrows represent synchronizations and solid arrows represent data transfer.

#### Fortran 2008 binding

```
MPI_Win_test(win, flag, ierror)
 TYPE(MPI_Win), INTENT(IN) :: win
 LOGICAL, INTENT(OUT) :: flag
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

#### Fortran binding

MPI_WIN_TEST(WIN, FLAG, IERROR) INTEGER WIN, IERROR LOGICAL FLAG

This is the nonblocking version of MPI_WIN_WAIT. It returns flag = true if all accesses to the local window by the group to which it was exposed by the corresponding MPI_WIN_POST call have been completed as signalled by matching MPI_WIN_COMPLETE calls, and flag = false otherwise. In the former case MPI_WIN_WAIT would have returned immediately. The effect of return of MPI_WIN_TEST with flag = true is the same as the effect of a return of MPI_WIN_WAIT. If flag = false is returned, then the call has no visible effect.

MPI_WIN_TEST should be invoked only where MPI_WIN_WAIT can be invoked. Once the call has returned flag = true, it must not be invoked anew, until the window is posted anew.

Assume that window win is associated with a "hidden" communicator wincomm, used for communication by the processes of win. The rules for matching of post and start calls and for matching complete and wait calls can be derived from the rules for matching sends and receives, by considering the following (partial) model implementation.

# MPI_WIN_POST(group,0,win) initiates a nonblocking send with tag tag0 to each process in group, using wincomm. There is no need to wait for the completion of these sends.

MPI_WIN_START(group,0,win) initiates a nonblocking receive with tag tag0 from each process in group, using wincomm. An RMA access to a window in target process i is delayed until the receive from i is completed.

#### Unofficial Draft for Comment Only

- **MPI_WIN_COMPLETE(win)** initiates a nonblocking send with tag tag1 to each process in the group of the preceding start call. No need to wait for the completion of these sends.
  - **MPI_WIN_WAIT(win)** initiates a nonblocking receive with tag **tag1** from each process in the group of the preceding post call. Wait for the completion of all receives.

No races can occur in a correct program: each of the sends matches a unique receive, and vice versa.

Rationale. The design for general active target synchronization requires the user to provide complete information on the communication pattern, at each end of a communication link: each origin specifies a list of targets, and each target specifies a list of origins. This provides maximum flexibility (hence, efficiency) for the implementor: each synchronization can be initiated by either side, since each "knows" the identity of the other. This also provides maximum protection from possible races. On the other hand, the design requires more information than RMA needs: in general, it is sufficient for the origin to know the rank of the target, but not vice versa. Users that want more "anonymous" communication will be required to use the fence or lock mechanisms. (*End of rationale.*)

Advice to users. Assume a communication pattern that is represented by a directed graph  $G = \langle V, E \rangle$ , where  $V = \{0, ..., n-1\}$  and  $ij \in E$  if origin process i accesses the window at target process j. Then each process i issues a call to MPI_WIN_POST( $ingroup_i, ...$ ), followed by a call to

²⁵ MPI_WIN_START( $outgroup_i,...$ ), where  $outgroup_i = \{j : ij \in E\}$  and  $ingroup_i = \{j : ji \in E\}$ . A call is a noop, and can be skipped, if the group argument is empty. ²⁷ After the communications calls, each process that issued a start will issue a complete. ²⁸ 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

34 35 36

30

31 32

33

MPI_WIN_LOCK(lock_type, rank, assert, win)

37 38 39	IN	lock_type	either MPI_LOCK_EXCLUSIVE or MPI_LOCK_SHARED (state)			
40	IN	rank	rank of locked window (non-negative integer)			
41	IN	assert	program assertion (integer)			
42 43	IN	win	window object (handle)			
44 45	C binding	5	nt rank, int assert, MPI Win win)			
46 47	<pre>int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win) Fortran 2008 binding</pre>					

⁴⁸ MPI_Win_lock(lock_type, rank, assert, win, ierror)

**Unofficial Draft for Comment Only** 

1

 $\mathbf{2}$ 

3

4

5

6 7

8

9 10

11

12

13

14

15

16

17

18

19

<pre>INTEGER, INTENT(IN) :: lock_type, rank, assert</pre>							
TY	PE(MPI_Win), INTENT(	(IN) :: win	2				
IN	INTEGER, OPTIONAL, INTENT(OUT) :: ierror						
Fortra	n binding		4				
	•	ANK, ASSERT, WIN, IERROR)	5				
		IK, ASSERT, WIN, IERROR	6				
			7 8				
	-	the window at the process with rank rank can be accessed	9				
-	_	r simultaneously; however, each access epoch must target a	10				
	t process.	simultaneously, nowever, each access epoch must target a	11				
umeren	t process.		12				
			13				
MPI_W	IN_LOCK_ALL(assert, v	vin)	14				
IN	assert	program assertion (integer)	15				
IN	win	window object (handle)	16				
	vviii	window object (indicie)	17				
C bind	ina		18				
		assert, MPI_Win win)	19				
			20 21				
	n 2008 binding		21				
	n_lock_all(assert, w		23				
	FEGER, INTENT(IN) ::		24				
TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror							
Fortran binding							
	MPI_WIN_LOCK_ALL(ASSERT, WIN, IERROR)						
IN	FEGER ASSERT, WIN, 1	IERROR	29				
Sta	rts an RMA access epo	och to all processes in win, with a lock type of	30				
		e epoch, the calling process can access the window memory on	31				
all proc	esses in win by using $RN$	A operations. A window locked with MPI_WIN_LOCK_ALL	32				
		VIN_UNLOCK_ALL. This routine is not collective—the ALL	33				
refers to	o a lock on all members	of the group of the window.	$\frac{34}{35}$				
A	dvice to users. There	may be additional overheads associated with using	36				
		I_WIN_LOCK_ALL concurrently on the same window. These	37				
		led by specifying the assertion MPI_MODE_NOCHECK when	38				
		(5.5). (End of advice to users.)	39				
-			40				
			41				
MPI_W	IN_UNLOCK(rank, win)		42				
IN	rank	rank of window (non-negative integer)	43				
			44				
IN	win	window object (handle)	45 46				
C bind			47 48				
int MPI_Win_unlock(int rank, MPI_Win win) 4							

**Unofficial Draft for Comment Only** 

```
1
 Fortran 2008 binding
\mathbf{2}
 MPI_Win_unlock(rank, win, ierror)
3
 INTEGER, INTENT(IN) :: rank
4
 TYPE(MPI_Win), INTENT(IN) :: win
5
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
 Fortran binding
7
 MPI_WIN_UNLOCK(RANK, WIN, IERROR)
8
 INTEGER RANK, WIN, IERROR
9
10
 Completes an RMA access epoch started by a call to MPI_WIN_LOCK on window win.
11
 RMA operations issued during this period will have completed both at the origin and at the
12
 target when the call returns.
13
14
 MPI_WIN_UNLOCK_ALL(win)
15
16
 IN
 window object (handle)
 win
17
18
 C binding
19
 int MPI_Win_unlock_all(MPI_Win win)
20
21
 Fortran 2008 binding
 MPI_Win_unlock_all(win, ierror)
22
 TYPE(MPI_Win), INTENT(IN) :: win
23
 INTEGER, OPTIONAL, INTENT(OUT) ::
^{24}
 ierror
25
 Fortran binding
26
 MPI_WIN_UNLOCK_ALL(WIN, IERROR)
27
 INTEGER WIN, IERROR
28
 Completes a shared RMA access epoch started by a call to MPI_WIN_LOCK_ALL on
29
 window win. RMA operations issued during this epoch will have completed both at the
30
 origin and at the target when the call returns.
^{31}
32
 Locks are used to protect accesses to the locked target window effected by RMA calls
33
 issued between the lock and unlock calls, and to protect load/store accesses to a locked local
34
 or shared memory window executed between the lock and unlock calls. Accesses that are
35
 protected by an exclusive lock will not be concurrent at the window site with other accesses
36
 to the same window that are lock protected. Accesses that are protected by a shared lock
37
 will not be concurrent at the window site with accesses protected by an exclusive lock to
38
 the same window.
39
 It is erroneous to have a window locked and exposed (in an exposure epoch) concur-
40
 rently. For example, a process may not call MPI_WIN_LOCK to lock a target window if
41
 the target process has called MPI_WIN_POST and has not yet called MPI_WIN_WAIT; it
42
 is erroneous to call MPI_WIN_POST while the local window is locked.
43
44
 Rationale.
 An alternative is to require MPI to enforce mutual exclusion between
45
 exposure epochs and locking periods. But this would entail additional overheads
46
 when locks or active target synchronization do not interact in support of those rare
47
 interactions between the two mechanisms. The programming style that we encourage
48
```

here is that a set of windows is used with only one synchronization mechanism at a time, with shifts from one mechanism to another being rare and involving global synchronization. (*End of rationale.*)

Advice to users. Users need to use explicit synchronization code in order to enforce mutual exclusion between locking periods and exposure epochs on a window. (*End of advice to users.*)

Implementors may restrict the use of RMA communication that is synchronized by lock calls to windows in memory allocated by MPI_ALLOC_MEM (Section 9.2), MPI_WIN_ALLOCATE (Section 12.2.2), MPI_WIN_ALLOCATE_SHARED (Section 12.2.3), or attached with MPI_WIN_ATTACH (Section 12.2.4). Locks can be used portably only in such memory.

*Rationale.* The implementation of passive target communication when memory is not shared may require an asynchronous software agent. Such an agent can be implemented more easily, and can achieve better performance, if restricted to specially allocated memory. It can be avoided altogether if shared memory is used. It seems natural to impose restrictions that allows one to use shared memory for third party communication in shared memory machines.

(End of rationale.)

Consider the sequence of calls in the example below.

Example 12.5	Use of MPL	_WIN_	LOCK	and	MPI.	_WIN_	UNLO	DCK.

```
MPI_Win_lock(MPI_LOCK_EXCLUSIVE, rank, assert, win);
MPI_Put(..., rank, ..., win);
MPI_Win_unlock(rank, win);
```

The call to MPI_WIN_UNLOCK will not return until the put transfer has completed at the origin and at the target. This still leaves much freedom to implementors. The call to MPI_WIN_LOCK may block until an exclusive lock on the window is acquired; or, the first two calls may not block, while MPI_WIN_UNLOCK blocks until a lock is acquired—the update of the target window is then postponed until the call to MPI_WIN_UNLOCK occurs. However, if the call to MPI_WIN_LOCK is used to lock a local window, then the call must block until the lock is acquired, since the lock may protect local load/store accesses to the window issued after the lock call returns.

# 12.5.4 Flush and SyncAll 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)

Unofficial Draft for Comment Only

 $\mathbf{2}$ 

```
1
 Fortran 2008 binding
\mathbf{2}
 MPI_Win_flush(rank, win, ierror)
3
 INTEGER, INTENT(IN) :: rank
4
 TYPE(MPI_Win), INTENT(IN) :: win
5
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
 Fortran binding
7
 MPI_WIN_FLUSH(RANK, WIN, IERROR)
8
 INTEGER RANK, WIN, IERROR
9
10
 MPI_WIN_FLUSH completes all outstanding RMA operations initiated by the calling
11
 process to the target rank on the specified window. The operations are completed both at
12
 the origin and at the target.
13
14
 MPI_WIN_FLUSH_ALL(win)
15
16
 window object (handle)
 IN
 win
17
18
 C binding
19
 int MPI_Win_flush_all(MPI_Win win)
20
21
 Fortran 2008 binding
 MPI_Win_flush_all(win, ierror)
22
 TYPE(MPI_Win), INTENT(IN) :: win
23
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
^{24}
25
 Fortran binding
26
 MPI_WIN_FLUSH_ALL(WIN, IERROR)
27
 INTEGER WIN, IERROR
28
 All RMA operations issued by the calling process to any target on the specified window
29
 prior to this call and in the specified window will have completed both at the origin and at
30
 the target when this call returns.
^{31}
32
33
 MPI_WIN_FLUSH_LOCAL(rank, win)
34
35
 IN
 rank
 rank of target window (non-negative integer)
36
 IN
 win
 window object (handle)
37
38
 C binding
39
 int MPI_Win_flush_local(int rank, MPI_Win win)
40
41
 Fortran 2008 binding
42
 MPI_Win_flush_local(rank, win, ierror)
43
 INTEGER, INTENT(IN) :: rank
44
 TYPE(MPI_Win), INTENT(IN) :: win
45
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
 Fortran binding
47
 MPI_WIN_FLUSH_LOCAL(RANK, WIN, IERROR)
48
```

INTEGER	RANK,	WIN,	IERROR
---------	-------	------	--------

Locally completes at the origin all outstanding RMA operations initiated by the calling process to the target process specified by rank on the specified window. For example, after this routine completes, the user may reuse any buffers provided to put, get, or accumulate operations.

MPI_WIN_FLUSH_LOCAL_ALL(win)		
IN win window object (handle)	9 10	
	11	
C binding	12	
int MPI_Win_flush_local_all(MPI_Win win)	13	
	14	
Fortran 2008 binding	15	
MPI_Win_flush_local_all(win, ierror)	16	
TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror	17	
INTEGER, OFITONAL, INTENT(001) TEITOI	18	
Fortran binding	19	
MPI_WIN_FLUSH_LOCAL_ALL(WIN, IERROR)	20	
INTEGER WIN, IERROR	21	
All RMA operations issued to any target prior to this call in this window will have	22	
completed at the origin when MPI_WIN_FLUSH_LOCAL_ALL returns.	23	
	24	
	25	
MPI_WIN_SYNC(win)	26	
IN win window object (handle)	27	
	28	
C binding	29	
int MPI_Win_sync(MPI_Win win)	30 31	
	32	
Fortran 2008 binding	33	
MPI_Win_sync(win, ierror)	34	
TYPE(MPI_Win), INTENT(IN) :: win	35	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	36	
Fortran binding	37	
MPI_WIN_SYNC(WIN, IERROR)	38	
INTEGER WIN, IERROR	39	
The call MPI_WIN_SYNC synchronizes the private and public window copies of win.	40	
For the purposes of synchronizing the private and public window, MPI_WIN_SYNC has the	41	
effect of ending and reopening an access and exposure epoch on the window (note that it	42	
does not actually end an epoch or complete any pending MPI RMA operations).	43	
does not actually one an epoch of complete any pending with think operations).	44	
12.5.5 Assertions	45	
	46	

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

#### Unofficial Draft for Comment Only

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,

¹⁸ MPI_MODE_NOPRECEDE, and MPI_MODE_NOSUCCEED. The significant options are listed ¹⁹ 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.

⁴⁸ MPI_WIN_FENCE:

5

6

7

8

9

10

11 12

13

14

15 16

17

21

22

23

 24 

25 26

27

28

29

30

31

32

33

34 35

36

37

38

39

40 41

42

43

44

45

46

- MPI_MODE_NOSTORE—the local window was not updated by stores (or local get or receive calls) since last synchronization.
- MPI_MODE_NOPUT—the local window will not be updated by put or accumulate calls after the fence call, until the ensuing (fence) synchronization.
- MPI_MODE_NOPRECEDE—the fence does not complete any sequence of locally issued RMA calls. If this assertion is given by any process in the window group, then it must be given by all processes in the group.
- MPI_MODE_NOSUCCEED—the fence does not start any sequence of locally issued RMA calls. If the assertion is given by any process in the window group, then it must be given by all processes in the group.

#### MPI_WIN_LOCK, MPI_WIN_LOCK_ALL:

MPI_MODE_NOCHECK—no other process holds, or will attempt to acquire, a conflicting lock, while the caller holds the window lock. This is useful when mutual exclusion is achieved by other means, but the coherence operations that may be attached to the lock and unlock calls are still required.

Advice to users. Note that the nostore and noprecede flags provide information on what happened before the call; the noput and nosucceed flags provide information on what will happen after the call. (End of advice to users.)

#### 12.5.6 Miscellaneous Clarifications

Once an RMA routine completes, it is safe to free any opaque objects passed as arguments to that routine. For example, the datatype argument of a MPI_PUT call can be freed as soon as the call returns, even though the communication may not be complete.

As in message-passing, datatypes must be committed before they can be used in RMA communication.

# 12.6 Error Handling

#### 12.6.1 Error Handlers

Errors occurring during calls to routines that create MPI windows (e.g., MPI_WIN_CREATE (...,comm,...)) cause the error handler currently associated with comm to be invoked. All other RMA calls have an input win argument. When an error occurs during such a call, the error handler currently associated with win is invoked.

The error handler MPI_ERRORS_ARE_FATAL is associated with win during its creation. Users may change this default by explicitly associating a new error handler with win (see Section 9.3).

#### 12.6.2 Error Classes

The error classes for one-sided communication are defined in Table 12.2. RMA routines may (and almost certainly will) use other MPI error classes, such as MPI_ERR_OP or MPI_ERR_RANK.

 24 

1	1	MPI_ERR_WIN	invalid win argument
2	2	MPI_ERR_BASE	invalid <b>base</b> argument
3	3	MPI_ERR_SIZE	invalid size argument
4	4	MPI_ERR_DISP	invalid disp argument
5	5	MPI_ERR_LOCKTYPE	invalid locktype argument
6	6	MPI_ERR_ASSERT	invalid assert argument
7	7	MPI_ERR_RMA_CONFLICT	conflicting accesses to window
٤	8	MPI_ERR_RMA_SYNC	invalid synchronization of RMA calls
ę	9	MPI_ERR_RMA_RANGE	target memory is not part of the window (in the case
1	0		of a window created with
1	1		MPI_WIN_CREATE_DYNAMIC, target memory is not
1	2		attached)
1	3	MPI_ERR_RMA_ATTACH	memory cannot be attached (e.g., because of resource
1	4		exhaustion)
1	5	MPI_ERR_RMA_SHARED	memory cannot be shared (e.g., some process in the
1	6		group of the specified communicator cannot expose
1	7		shared memory)
1	8	MPI_ERR_RMA_FLAVOR	passed window has the wrong flavor for the called
1	9		function
2	0		
2	1		

Table 12.2: Error classes in one-sided communication routines

# 12.7 Semantics and Correctness

The following rules specify the latest time at which an operation must complete at the origin or the target. The update performed by a get call in the origin process memory is visible when the get operation is complete at the origin (or earlier); the update performed by a put or accumulate call in the public copy of the target window is visible when the put or accumulate has completed at the target (or earlier). The rules also specify the latest time at which an update of one window copy becomes visible in another overlapping copy.

- An RMA operation is completed at the origin by the ensuing call to MPI_WIN_COMPLETE, MPI_WIN_FENCE, MPI_WIN_FLUSH, MPI_WIN_FLUSH_ALL, MPI_WIN_FLUSH_LOCAL, MPI_WIN_FLUSH_LOCAL_ALL, MPI_WIN_UNLOCK, or MPI_WIN_UNLOCK_ALL that synchronizes this access at the origin.
- 2. If an RMA operation is completed at the origin by a call to MPI_WIN_FENCE then the operation is completed at the target by the matching call to MPI_WIN_FENCE by the target process.
  - 3. If an RMA operation is completed at the origin by a call to MPI_WIN_COMPLETE then the operation is completed at the target by the matching call to MPI_WIN_WAIT by the target process.
- 4. If an RMA operation is completed at the origin by a call to MPI_WIN_UNLOCK, MPI_WIN_UNLOCK_ALL, MPI_WIN_FLUSH(rank=target), or MPI_WIN_FLUSH_ALL, then the operation is completed at the target by that same call.

22 23

24 25

26

27

28

29

30

31 32

33

34

35

36

37

38

39

40 41

42

43

- 5. An update of a location in a private window copy in process memory becomes visible in the public window copy at latest when an ensuing call to MPI_WIN_POST, MPI_WIN_FENCE, MPI_WIN_UNLOCK, MPI_WIN_UNLOCK_ALL, or MPI_WIN_SYNC is executed on that window by the window owner. In the RMA unified memory model, an update of a location in a private window in process memory becomes visible without additional RMA calls.
- 6. An update by a put or accumulate call to a public window copy becomes visible in the private copy in process memory at latest when an ensuing call to MPI_WIN_WAIT, MPI_WIN_FENCE, MPI_WIN_LOCK, MPI_WIN_LOCK_ALL, or MPI_WIN_SYNC is executed on that window by the window owner. In the RMA unified memory model, an update by a put or accumulate call to a public window copy eventually becomes visible in the private copy in process memory without additional RMA calls.

The MPI_WIN_FENCE or MPI_WIN_WAIT call that completes the transfer from public copy to private copy (6) is the same call that completes the put or accumulate operation in the window copy (2, 3). If a put or accumulate access was synchronized with a lock, then the update of the public window copy is complete as soon as the updating process executed MPI_WIN_UNLOCK or MPI_WIN_UNLOCK_ALL. In the RMA separate memory model, the update of a private copy in the process memory may be delayed until the target process executes a synchronization call on that window (6). Thus, updates to process memory can always be delayed in the RMA separate memory model until the process executes a suitable synchronization call, while they must complete in the RMA unified model without additional synchronization calls. If fence or post-start-complete-wait synchronization is used, updates to a public window copy can be delayed in both memory models until the window owner executes a synchronization call. When passive target synchronization is used, it is necessary to update the public window copy even if the window owner does not execute any related synchronization call.

The rules above also define, by implication, when an update to a public window copy becomes visible in another overlapping public window copy. Consider, for example, two overlapping windows, win1 and win2. A call to MPI_WIN_FENCE(0, win1) by the window owner makes visible in the process memory previous updates to window win1 by remote processes. A subsequent call to MPI_WIN_FENCE(0, win2) makes these updates visible in the public copy of win2.

The behavior of some MPI RMA operations may be *undefined* in certain situations. For example, the result of several origin processes performing concurrent MPI_PUT operations to the same target location is undefined. In addition, the result of a single origin process performing multiple MPI_PUT operations to the same target location within the same access epoch is also undefined. The result at the target may have all of the data from one of the MPI_PUT operations (the "last" one, in some sense), bytes from some of each of the operations, or something else. In MPI-2, such operations were *erroneous*. That meant that an MPI implementation was permitted to 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 ⁴⁶ be erroneous makes it difficult to use MPI RMA to implement programming models—⁴⁷ such as Unified Parallel C (UPC) or SHMEM—that permit these operations. Further, ⁴⁸

#### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6 7

8

9

10

11

12

13 14

15

16

17

18

19

20

21

22

23

 24 

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

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 the target window, until the update becomes visible in the public copy of the window. Conversely, a store to process memory to a location in a window must not start once a put or accumulate update to that target window has started, until the put or accumulate update becomes visible in process memory. In both cases, the restriction applies to operations even if they access disjoint locations in the window.
- Rationale. The last constraint on correct RMA accesses may seem unduly restric-32 tive, as it forbids concurrent accesses to nonoverlapping locations in a window. The 33 34 reason for this constraint is that, on some architectures, explicit coherence restoring operations may be needed at synchronization points. A different operation may 35 be needed for locations that were updated by stores and for locations that were re-36 motely updated by put or accumulate operations. Without this constraint, the MPI 37 library would have to track precisely which locations in a window were updated by a 38 put or accumulate call. The additional overhead of maintaining such information is 39 considered prohibitive. (*End of rationale.*) 40
- Note that MPI_WIN_SYNC may be used within a passive target epoch to synchronize
  the private and public window copies (that is, updates to one are made visible to the other).
  In the MPI_WIN_UNIFIED memory model, the rules are simpler because the public and
  private windows are the same. However, there are restrictions to avoid concurrent access
  to the same memory locations by different processes. The rules that a program with a
  well-defined outcome must obey in this case are:
- 48

41

1

2

3

4

5 6

7

8

9 10

11

12 13

14

15

16 17

18

19

20

21

22

23

- U1. A location in a window must not be accessed with load/store operations once an update to that location has started, until the update is complete, subject to the following special case.
- U2. Accessing a location in the window that is also the target of a remote update is valid (not erroneous) but the precise result will depend on the behavior of the implementation. Updates from a remote process will appear in the memory of the target, but there are no atomicity or ordering guarantees if more than one byte is updated. Updates are stable in the sense that once data appears in memory of the target, the data remains until replaced by another update. This permits polling on a location for a change from zero to non-zero or for a particular value, but not polling and comparing the relative magnitude of values. Users are cautioned that polling on one memory location and then accessing a different memory location has defined behavior only if the other rules given here and in this chapter are followed.

Advice to users. Some compiler optimizations can result in code that maintains the sequential semantics of the program, but violates this rule by introducing temporary values into locations in memory. Most compilers only apply such transformations under very high levels of optimization and users should be aware that such aggressive optimization may produce unexpected results. (*End of advice to users.*)

- U3. Updating a location in the window with a store operation that is also the target of a remote read (but not update) is valid (not erroneous) but the precise result will depend on the behavior of the implementation. Store updates will appear in memory, but there are no atomicity or ordering guarantees if more than one byte is updated. Updates are stable in the sense that once data appears in memory, the data remains until replaced by another update. This permits updates to memory with store operations without requiring an RMA epoch. Users are cautioned that remote accesses to a window that is updated by the local process has defined behavior only if the other rules given here and elsewhere in this chapter are followed.
- U4. A location in a window must not be accessed as a target of an RMA operation once an update to that location has started and until the update completes at the target. There is one exception to this rule: in the case where the same location is updated by two concurrent accumulates with the same predefined datatype on the same window. Additional restrictions on the operation apply; see the info key accumulate_ops in Section 12.2.1.
- U5. A put or accumulate must not access a target window once a store, put, or accumulate update to another (overlapping) target window has started on the same location in the target window and until the update completes at the target window. Conversely, a store operation to a location in a window must not start once a put or accumulate update to the same location in that target window has started and until the put or accumulate update completes at the target.

Advice to users. In the unified memory model, in the case where the window is in shared memory, MPI_WIN_SYNC can be used to order store operations and make store updates to the window visible to other processes and threads. Use of this

#### **Unofficial Draft for Comment Only**

1

 $\mathbf{2}$ 

3

4

5

6

 $\overline{7}$ 

8

9

10

11

12

13

14 15

16

17

18

19

20

21 22

23

 24 

25

26

27

28

29

30

31 32

33

34

35

36

37

38

39

40

41

42

43

44 45

46

47

1	routine is necessary to ensure portable behavior when point-to-point, collective, or
2	shared memory synchronization is used in place of an RMA synchronization routine.
3	MPI_WIN_SYNC should be called by the writer before the non-RMA synchroniza-
4	tion operation and by the reader after the non-RMA synchronization, as shown in
5	Example 12.21. (End of advice to users.)
6	Example 12.21. (Ena of advice to users.)
	A program that violates these rules has undefined behavior.
7	
8	Advice to users. A user can write correct programs by following the following rules:
9	fence: During each period between fence calls, each window is either updated by put
10	
11	or accumulate calls, or updated by stores, but not both. Locations updated by
12	put or accumulate calls should not be accessed during the same period (with
13	the exception of concurrent updates to the same location by accumulate calls).
14	Locations accessed by get calls should not be updated during the same period.
	<b>post-start-complete-wait:</b> A window should not be updated with store operations
15	while posted if it is being updated by put or accumulate calls. Locations updated
16	by put or accumulate calls should not be accessed while the window is posted
17	
18	(with the exception of concurrent updates to the same location by accumulate
19	calls). Locations accessed by get calls should not be updated while the window
20	is posted.
21	With the post-start synchronization, the target process can tell the origin process
22	that its window is now ready for RMA access; with the complete-wait synchro-
23	nization, the origin process can tell the target process that it has finished its
24	RMA accesses to the window.
	lock: Updates to the window are protected by exclusive locks if they may conflict.
25	
26	Nonconflicting accesses (such as read-only accesses or accumulate accesses) are
27	protected by shared locks, both for load/store accesses and for RMA accesses.
28	changing window or synchronization mode: One can change synchronization
29	mode, or change the window used to access a location that belongs to two over-
30	lapping windows, when the process memory and the window copy are guaranteed
31	to have the same values. This is true after a local call to MPI_WIN_FENCE, if
32	RMA accesses to the window are synchronized with fences; after a local call
33	to MPI_WIN_WAIT, if the accesses are synchronized with post-start-complete-
34	
	wait; after the call at the origin (local or remote) to MPI_WIN_UNLOCK or
35	MPI_WIN_UNLOCK_ALL if the accesses are synchronized with locks.
36	In addition, a process should not access the local buffer of a get operation until the
37	operation is complete, and should not update the local buffer of a put or accumulate
38	operation in complete, and should not update the local buller of a put of accumulate operation until that operation is complete.
39	
40	The RMA synchronization operations define when updates are guaranteed to become
41	visible in public and private windows. Updates may become visible earlier, but such
42	behavior is implementation dependent. (End of advice to users.)
43	
44	The semantics are illustrated by the following examples:
45	<b>Example 12.6</b> The following example demonstrates updating a memory location inside
46	
47	a window for the separate memory model, according to Rule 5. The MPI_WIN_LOCK
48	and MPI_WIN_UNLOCK calls around the store to X in process B are necessary to ensure

⁴⁸ consistency between the public and private copies of the window.

# **Unofficial Draft for Comment Only**

Process A:	Process B:	1			
	window location X	2			
		3			
	MPI_Win_lock(EXCLUSIVE, B)	4			
	store X /* local update to private copy of B */	5			
	MPI_Win_unlock(B)	6			
	<pre>/* now visible in public window copy */</pre>	7			
		8			
MPI_Barrier	MPI_Barrier	9			
		10			
MPI_Win_lock(EXCLUSIVE,	B)	11			
<pre>MPI_Get(X) /* ok, read</pre>	from public window */	12			
MPI_Win_unlock(B)		13			
		14			
Example 12.7 In the RM	A unified model, although the public and private copies of the	15			
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.		
					20
Process A:	Process B:	21			
	window location X	22			
		23			
	store X /* update to private & public copy of B */	24			
MPI_Barrier	MPI_Barrier	25			
MPI_Win_lock_all		26			
<pre>MPI_Get(X) /* ok, read</pre>	from window */	27			
MPI_Win_flush_local(B)		28			
/* read value in X */		29			
MPI_Win_unlock_all		30			
		31 32			
MPI_BARRIER provides process synchronization, but not memory synchronization.		32			
	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		35 36			
result.					
		37			

**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.

		10
Process A:	Process B:	44
	window location X	45
		46
MPI_Win_lock(EXCLUSIVE,	B)	47
MPI_Put(X) /* update to	<pre>public window */</pre>	48

38

39 40

 41 

42

1 2	MPI_Win_unlock(B)	
3	MPI_Barrier	MDT Demnion
4	MP1_Barrier	MPI_Barrier
5		MPI_Win_lock(EXCLUSIVE, B)
6		/* now visible in private copy of B */
7		load X
8		MPI_Win_unlock(B)
9		In 1_win_unioek(D)
10	Note that in this example, the	e barrier is not critical to the semantic correctness. The
11	use of exclusive locks guarante	es a remote process will not modify the public copy after
12	MPI_WIN_LOCK synchronizes	the private and public copies. A polling implementation
13		ess B would be semantically correct. The barrier is required
14	to ensure that process A perform	ms the put operation before process B performs the load of
15	Х.	
16		
17	-	nple 12.7, the following example is unsafe even in the unified
18	,	n not be guaranteed to occur after the MPI_BARRIER. While
19		plicitly synchronize the public and private copies through
20		PUT will update both the public and private copies of the
21		bad could result in old values of X being returned. Compiler
22	*	s could ensure the load occurs after the data is updated, or
23	explicit one-sided synchronization	on calls can be used to ensure the proper result.
24	Process A:	Process B:
25	TIOCESS A.	window location X
26	MPI_Win_lock_all	window ideation x
27	MPI_Put(X) /* update to win	ndow */
28	MPI_Win_flush(B)	
29		
30	MPI_Barrier	MPI_Barrier
31		load X
32	MPI_Win_unlock_all	
33		
34		*
35	-	ag example further clarifies Rule 5. MPI_WIN_LOCK and
36		update the public copy of a window with changes to the
37		s no guarantee that process A in the following sequence will
38 39	see the value of X as updated by	y the local store by process B before the lock.
40	Process A:	Process B:
41	FIOCESS A.	window location X
42		WINDOW IOCALION X
43		store X /* update to private copy of B */
44		MPI_Win_lock(SHARED, B)
45	MPI_Barrier	MPI_Barrier
46	In 1_bullion	In I_Dullion
47	MPI_Win_lock(SHARED, B)	
48	MPI_Get(X) /* X may be the	X before the store */
	<i>v</i>	

Unofficial Draft for Comment Only

MPI_Win_unlock	(B)
----------------	-----

MP	I_Win_ur	nloc	ck	(B)					
/*	update	on	Х	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:	14
window location X	FICCESS D.	15
window location X window location Y		16
WINDOW LOCALION I		17
		18
store Y		19
MPI_Win_post(A, B) /* Y vis	-	20
MPI_Win_start(A)	MPI_Win_start(A)	21
		22
store X /* update to privat	te window */	23
		24
MPI_Win_complete	MPI_Win_complete	24
MPI_Win_wait		
<pre>/* update on X may not yet</pre>	visible in public window */	26
	·	27
MPI_Barrier	MPI_Barrier	28
		29
	MPI_Win_lock(EXCLUSIVE, A)	30
		31
	<pre>MPI_Get(X) /* may return an obsolete value */</pre>	32
	MPI_Get(Y)	33
	MPI_Win_unlock(A)	34

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.

```
1
 Process A:
 Process B:
\mathbf{2}
 window location X
3
4
 MPI_Win_lock(EXCLUSIVE, B)
5
 MPI_Put(X) /* update to public window */
6
 MPI_Win_unlock(B)
\overline{7}
8
 MPI_Barrier
 MPI_Barrier
9
10
 MPI_Win_post(B)
11
 MPI_Win_start(B)
12
13
 load X /* access to private window */
14
 /* may return an obsolete value */
15
16
 MPI_Win_complete
17
 MPI_Win_wait
18
 To ensure that the value put by process A is read, the local load must be replaced with a
19
```

local MPI_GET operation, or must be placed after the call to MPI_WIN_WAIT.

# ²² 12.7.1 Atomicity

21

36

37

23The outcome of concurrent accumulate operations to the same location with the same  24 predefined datatype is as if the accumulates were done at that location in some serial 25order. Additional restrictions on the operation apply; see the info key accumulate_ops in 26Section 12.2.1. Concurrent accumulate operations with different origin and target pairs are 27not ordered. Thus, there is no guarantee that the entire call to an accumulate operation is 28executed atomically. The effect of this lack of atomicity is limited: The previous correctness 29 conditions imply that a location updated by a call to an accumulate operation cannot be 30 accessed by a load or an RMA call other than accumulate until the accumulate operation has  31 completed (at the target). Different interleavings can lead to different results only to the 32 extent that computer arithmetics are not truly associative or commutative. The outcome 33 of accumulate operations with overlapping types of different sizes or target displacements 34is undefined. 35

# 12.7.2 Ordering

38Accumulate calls enable element-wise atomic read and write to remote memory locations. 39 MPI specifies ordering between accumulate operations from an origin process to the same 40(or overlapping) memory locations at a target process on a per-datatype granularity. The 41 default ordering is strict ordering, which guarantees that overlapping updates from the 42same origin to a remote location are committed in program order and that reads (e.g., with 43MPI_GET_ACCUMULATE) and writes (e.g., with MPI_ACCUMULATE) are executed and 44committed in program order. Ordering only applies to operations originating at the same 45origin that access overlapping target memory regions. MPI does not provide any guarantees 46for accesses or updates from different origin processes to overlapping target memory regions. 47The default strict ordering may incur a significant performance penalty. MPI specifies 48the info key "accumulate_ordering" to allow relaxation of the ordering semantics when specified

# Unofficial Draft for Comment Only

to any window creation function. The values for this key are as follows. If set to "none", then no ordering will be guaranteed for accumulate calls. This was the behavior for RMA in 3 MPI-2 but has not been the default since MPI-3. The key can be set to a comma-separated 4 list of required access orderings at the target. Allowed values in the comma-separated list are "rar", "war", "raw", and "waw" for read-after-read, write-after-read, read-after-write, and write-after-write ordering, respectively. These indicate whether operations of the specified type complete in the order they were issued. For example, "raw" means that any writes must complete at the target before subsequent reads. These ordering requirements apply only to operations issued by the same origin process and targeting the same target process. The default value for "accumulate_ordering" is rar, raw, war, waw, which implies that writes complete 10 11at the target in the order in which they were issued, reads complete at the target before any 12writes that are issued after the reads, and writes complete at the target before any reads 13 that are issued after the writes. Any subset of these four orderings can be specified. For 14example, if only read-after-read and write-after-write ordering is required, then the value of 15the "accumulate_ordering" key could be set to rar, waw. The order of values is not significant.

Note that the above ordering semantics apply only to accumulate operations, not put and get. Put and get within an epoch are unordered.

#### 12.7.3 Progress

One-sided communication has the same progress requirements as point-to-point communication: once a communication is enabled it is guaranteed to complete. RMA calls must have local semantics, except when required for synchronization with other RMA calls.

There is some fuzziness in the definition of the time when a RMA communication becomes enabled. This fuzziness provides to the implementor more flexibility than with point-to-point communication. Access to a target window becomes enabled once the corresponding synchronization (such as MPI_WIN_FENCE or MPI_WIN_POST) has executed. On the origin process, an RMA communication may become enabled as soon as the corresponding put, get or accumulate call has executed, or as late as when the ensuing synchronization call is issued. Once the communication is enabled both at the origin and at the target, the communication must complete.

Consider the code fragment in Example 12.4. Some of the calls may block if the target window is not posted. However, if the target window is posted, then the code fragment must complete. The data transfer may start as soon as the put call occurs, but may be delayed until the ensuing complete call occurs.

Consider the code fragment in Example 12.5. Some of the calls may block if another process holds a conflicting lock. However, if no conflicting lock is held, then the code fragment must complete.

Consider the code illustrated in Figure 12.6. Each process updates the window of the other process using a put operation, then accesses its own window. The post calls are nonblocking, and should complete. Once the post calls occur, RMA access to the windows is enabled, so that each process should complete the sequence of calls start-put-complete. Once 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 45each process. Then, the code may deadlock, as each process may block on the start call, 46waiting for the matching post to occur. Similarly, the program will deadlock if the order of 47the complete and wait calls is reversed at each process. 48

#### **Unofficial Draft for Comment Only**

1

 $\mathbf{2}$ 

 $\mathbf{5}$ 

6  $\overline{7}$ 

8 9

1617

18 19

20

21

22

23

 24 

25

26

27

28

29

30

31

32

33

34

35

36

37

38

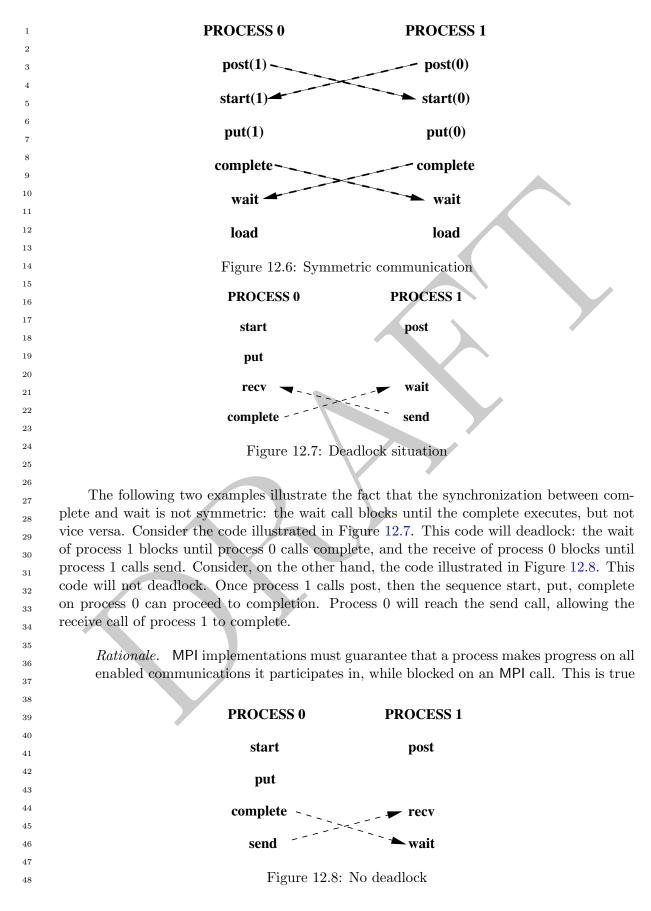
39

40

41

42

43



Unofficial Draft for Comment Only

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 by the following code:

Source of Process 1	Source of Process 2	Executed in Process 2	З
bbbb = 777	buff = 999	reg_A:=999	З
call MPI_WIN_FENCE	call MPI_WIN_FENCE		3
call MPI_PUT(bbbb		stop appl.thread	3
into buff of process 2)		buff:=777 in PUT handler	3
		continue appl.thread	3
call MPI_WIN_FENCE	call MPI_WIN_FENCE		3
	ccc = buff	ccc:=reg_A	4

In this example, variable buff is allocated in the register reg_A and therefore ccc will have the old value of buff and not the new value 777.

This problem, which also afflicts in some cases send/receive communication, is discussed more at length in Section 19.1.16.

Programs written in C avoid this problem, because of the semantics of C. Many Fortran 46 compilers will avoid this problem, without disabling compiler optimizations. However, in 47 order to avoid register coherence problems in a completely portable manner, users should 48

#### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

5

6

 $\overline{7}$ 

8

9

10

11

12

13

14

15

16

17

18

19 20

21

22 23

 24 

25

26

27

28

29

30

42

43

44

restrict their use of RMA windows to variables stored in modules or COMMON blocks. To
 prevent problems with the argument copying and register optimization done by Fortran
 compilers, please note the hints in Sections 19.1.10–19.1.20. Sections 19.1.17 to 19.1.17
 discuss several solutions for the problem in this example.

# 12.8 Examples

**Example 12.13** The following example shows a generic loosely synchronous, iterative code, using fence synchronization. The window at each process consists of array A, which contains the origin and target buffers of the put calls.

```
13
 . . .
14
 while (!converged(A)) {
15
 update(A);
16
 MPI_Win_fence(MPI_MODE_NOPRECEDE, win);
17
 for(i=0; i < toneighbors; i++)</pre>
18
 MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
19
 todisp[i], 1, totype[i], win);
20
 MPI_Win_fence((MPI_MODE_NOSTORE | MPI_MODE_NOSUCCEED), win);
 }
```

```
21
22
```

27

32

5 6

7 8 9

10

11

12

The same code could be written with get rather than put. Note that, during the communication phase, each window is concurrently read (as origin buffer of puts) and written (as target buffer of puts). This is OK, provided that there is no overlap between the target buffer of a put and another communication buffer.

Example 12.14 Same generic example, with more computation/communication overlap. We assume that the update phase is broken into two subphases: the first, where the "boundary," which is involved in communication, is updated, and the second, where the "core," which neither uses nor provides communicated data, is updated.

```
33
 while (!converged(A)) {
34
 update_boundary(A);
35
 MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win);
36
 for(i=0; i < fromneighbors; i++)</pre>
37
 MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
38
 fromdisp[i], 1, fromtype[i], win);
39
 update_core(A);
40
 MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
41
 }
42
```

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.

# Unofficial Draft for Comment Only

**Example 12.15** Same code as in Example 12.13, rewritten using post-start-complete-wait.

```
while (!converged(A)) {
 update(A);
 MPI_Win_post(fromgroup, 0, win);
 MPI_Win_start(togroup, 0, win);
 for(i=0; i < toneighbors; i++)
 MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
 todisp[i], 1, totype[i], win);
 MPI_Win_complete(win);
 MPI_Win_wait(win);
}</pre>
```

Example 12.16 Same example, with split phases, as in Example 12.14.

**Example 12.17** A checkerboard, or double buffer communication pattern, that allows more computation/communication overlap. Array A0 is updated using values of array A1, and vice versa. We assume that communication is symmetric: if process A gets data from process B, then process B gets data from process A. Window wini consists of array Ai.

```
35
if (!converged(A0,A1))
 36
 MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
 37
MPI_Barrier(comm0);
 38
/* the barrier is needed because the start call inside the
 39
loop uses the nocheck option */
 40
 41
while (!converged(A0, A1)) {
 42
 /* communication on AO and computation on A1 */
 update2(A1, A0); /* local update of A1 that depends on A0 (and A1) */
 43
 MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win0);
 44
 for(i=0; i < fromneighbors; i++)</pre>
 45
 MPI_Get(&tobuf0[i], 1, totype0[i], neighbor[i],
 46
 fromdisp0[i], 1, fromtype0[i], win0);
 47
 update1(A1); /* local update of A1 that is
 48
```

 24 

1	concurrent with communication	on that updates AO */
2	<pre>MPI_Win_post(neighbors, (MPI_MODE_NOCHECK  </pre>	<pre>MPI_MODE_NOPUT), win1);</pre>
3	<pre>MPI_Win_complete(win0);</pre>	
4	<pre>MPI_Win_wait(win0);</pre>	
5		
6	<pre>/* communication on A1 and computation on AC</pre>	) */
7	update2(AO, A1); /* local update of AO that	depends on A1 (and A0) $*/$
8	MPI_Win_start(neighbors, MPI_MODE_NOCHECK, w	vin1);
9	<pre>for(i=0; i &lt; fromneighbors; i++)</pre>	
10	<pre>MPI_Get(&amp;tobuf1[i], 1, totype1[i], neighbo</pre>	pr[i],
11	<pre>fromdisp1[i], 1, fromtype1[i],</pre>	win1);
12	update1(AO); /* local update of AO that depe	ends on AO only,
13	concurrent with communication	that updates A1 */
14	if (!converged(A0,A1))	
15	MPI_Win_post(neighbors, (MPI_MODE_NOCHECK	<pre>MPI_MODE_NOPUT), win0);</pre>
16	<pre>MPI_Win_complete(win1);</pre>	
17	<pre>MPI_Win_wait(win1);</pre>	
18	}	
19	A process posts the local window associated with wi	no before it completes RMA accesses
20	to the remote windows associated with win1. When the	-
21	neighbors of the calling process have posted the window	
22	when the wait(win0) call returns, then all neighbors of	
23	windows associated with win1. Therefore, the nocheck	
24	MPI_WIN_START.	option can be used with the cans to
25	Put calls can be used, instead of get calls, if the	area of array AO (resp. A1) used by
26	the update(A1, A0) (resp. update(A0, A1)) call is dis	
27	RMA communication. On some systems, a put call ma	
28	as it requires information exchange only in one direction	
29	In the next several examples, for conciseness, the	
30		Supression
31	<pre>z = MPI_Get_accumulate()</pre>	
32	means to perform an MPI_GET_ACCUMULATE with the	e result huffer (given by result addr
33 34	in the description of MPI_GET_ACCUMULATE) on the	(0
	case, z. This format is also used with MPI_COMPARE_	а ,
35	Process B refers to any process other than A.	
36		
37	<b>Example 12.18</b> The following example implements a	naive, non-scalable counting sema-
38 20	phore. The example demonstrates the use of MPI_WIN_	, .
39 40	of X, as well as MPI_WIN_FLUSH to complete operation	
40 41	opened with MPI_WIN_LOCK_ALL. To avoid the rule	
41 42	public and private copies of windows, MPI_ACCUMUL	
	are used to write to or read from the local public copy.	
43 44		
44	Process A: Proc	ess B:

45 MPI_Win_lock_all MPI_Win_lock_all 46 window location X 47 X=MPI_Comm_size() 48 MPI_Win_sync

MPI_Barrier	MPI_Barrier
MPI_Accumulate(X, MPI_SUM, -1)	MPI_Accumulate(X, MPI_SUM, -1)
stack variable z	stack variable z
do	do
<pre>z = MPI_Get_accumulate(X,</pre>	<pre>z = MPI_Get_accumulate(X,</pre>
MPI_NO_OP, 0)	MPI_NO_OP, 0)
MPI_Win_flush(A)	MPI_Win_flush(A)
while(z!=0)	while(z!=0)
MPI_Win_unlock_all	MPI_Win_unlock_all

**Example 12.19** Implementing a critical region between two processes (Peterson's algorithm). Despite their appearance in the following example, MPI_WIN_LOCK_ALL and MPI_WIN_UNLOCK_ALL are not collective calls, but it is frequently useful to start shared access epochs to all processes from all other processes in a window. Once the access epochs are established, accumulate communication operations and flush and sync synchronization operations can be used to read from or write to the public copy of the window.

Process A:	Process B:	21
window location X	window location Y	22
window location T		23
		24
MPI_Win_lock_all	MPI_Win_lock_all	25
X=1	Y=1	26
MPI_Win_sync	MPI_Win_sync	27
MPI_Barrier	MPI_Barrier	28
MPI_Accumulate(T, MPI_REPLACE, 1)	MPI_Accumulate(T, MPI_REPLACE, 0)	29
stack variables t,y	stack variable t,x	30
t=1	t=0	31
<pre>y=MPI_Get_accumulate(Y,</pre>	x=MPI_Get_accumulate(X,	32
MPI_NO_OP, 0)	MPI_NO_OP, 0)	33
while(y==1 && t==1) do	while(x==1 && t==0) do	34
y=MPI_Get_accumulate(Y,	x=MPI_Get_accumulate(X,	35
MPI_NO_OP, 0)	MPI_NO_OP, O)	36
t=MPI_Get_accumulate(T,	t=MPI_Get_accumulate(T,	37
MPI_NO_OP, O)	MPI_NO_OP, 0)	38
MPI_Win_flush_all	MPI_Win_flush(A)	39
done	done	40
<pre>// critical region</pre>	// critical region	41
MPI_Accumulate(X, MPI_REPLACE, 0)	0	42
MPI_Win_unlock_all	MPI_Win_unlock_all	43
	····	44

**Example 12.20** Implementing a critical region between multiple processes with compare and swap. The call to MPI_WIN_SYNC is necessary on Process A after local initialization of **A** to guarantee the public copy has been updated with the initialization value found in

# Unofficial Draft for Comment Only

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

³ A in the public copy of Process A had been updated before the barrier.

```
Process A:
 Process B...:
5
 MPI_Win_lock_all
 MPI_Win_lock_all
6
 atomic location A
7
 A=0
8
 MPI_Win_sync
9
 MPI_Barrier
 MPI_Barrier
10
 stack variable r=1
 stack variable r=1
11
 while(r != 0) do
 while(r != 0) do
12
 r = MPI_Compare_and_swap(A, 0, 1)
 r = MPI_Compare_and_swap(A, 0, 1)
13
 MPI_Win_flush(A)
 MPI_Win_flush(A)
14
 done
 done
15
 // critical region
 // critical region
16
 r = MPI_Compare_and_swap(A, 1, 0)
 r = MPI_Compare_and_swap(A, 1, 0)
17
 MPI_Win_unlock_all
 MPI_Win_unlock_all
18
19
```

20**Example 12.21** The following example demonstrates the proper synchronization in the 21unified memory model when a data transfer is implemented with load and store in the case 22of windows in shared memory (instead of MPI_PUT or MPI_GET) and the synchronization 23between processes is performed using point-to-point communication. The synchronization  24 between processes must be supplemented with a memory synchronization through calls to 25MPI_WIN_SYNC, which act locally as a processor-memory barrier. In Fortran, if 26MPI_ASYNC_PROTECTS_NONBLOCKING is .FALSE. or the variable X is not declared as 27ASYNCHRONOUS, reordering of the accesses to the variable X must be prevented with 28MPI_F_SYNC_REG operations. (No equivalent function is needed in C.)

The variable X is contained within a shared memory window and X corresponds to the same memory location at both processes. The MPI_WIN_SYNC operation performed by process A ensures completion of the load/store operations issued by process A. The MPI_WIN_SYNC operation performed by process B ensures that process A's updates to X are visible to process B.

```
35
 Process A:
 Process B:
36
 MPI_WIN_LOCK_ALL(
 MPI_WIN_LOCK_ALL(
37
 MPI_MODE_NOCHECK,win)
 MPI_MODE_NOCHECK,win)
38
39
 DO ...
 DO ...
40
 X=...
41
42
 MPI_F_SYNC_REG(X)
43
 MPI_WIN_SYNC(win)
44
 MPI_SEND
 MPI_RECV
45
 MPI_WIN_SYNC(win)
46
 MPI_F_SYNC_REG(X)
47
48
 print X
```

Unofficial Draft for Comment Only

MPI_RECV MPI_F_SYNC_REG(X)	MPI_F_SYNC_REG(X) MPI_SEND
END DO	END DO
MPI_WIN_UNLOCK_ALL(win)	MPI_WIN_UNLOCK_ALL(win)

**Example 12.22** The following example shows how request-based operations can be used to overlap communication with computation. Each process fetches, processes, and writes the result for NSTEPS chunks of data. Instead of a single buffer, M local buffers are used to allow up to M communication operations to overlap with computation.

```
int
 i, j;
 15
MPI_Win
 win;
 16
MPI_Request put_req[M] = { MPI_REQUEST_NULL };
 17
MPI_Request get_req;
 18
double
 *baseptr;
 19
double
 data[M][N];
 20
 21
MPI_Win_allocate(NSTEPS*N*sizeof(double), sizeof(double), MPI_INFO_NULL,
 22
 MPI_COMM_WORLD, &baseptr, &win);
 23
 ^{24}
MPI_Win_lock_all(0, win);
 25
 26
for (i = 0; i < NSTEPS; i++) {
 27
 if (i<M)
 28
 j=i;
 29
 else
 30
 MPI_Waitany(M, put_req, &j, MPI_STATUS_IGNORE);
 31
 32
 MPI_Rget(data[j], N, MPI_DOUBLE, target, i*N, N, MPI_DOUBLE, win,
 33
 &get_req);
 34
 MPI_Wait(&get_req,MPI_STATUS_IGNORE);
 35
 compute(i, data[j], ...);
 36
 MPI_Rput(data[j], N, MPI_DOUBLE, target, i*N, N, MPI_DOUBLE, win,
 37
 &put_req[j]);
 38
}
 39
 40
MPI_Waitall(M, put_req, MPI_STATUSES_IGNORE);
 41
MPI_Win_unlock_all(win);
 42
```

**Example 12.23** The following example constructs a distributed shared linked list using ⁴⁴ dynamic windows. Initially process 0 creates the head of the list, attaches it to the window, ⁴⁵ and broadcasts the pointer to all processes. All processes then concurrently append N new ⁴⁶ elements to the list. When a process attempts to attach its element to the tail of the ⁴⁷ list it may discover that its tail pointer is stale and it must chase ahead to the new tail ⁴⁸

# Unofficial Draft for Comment Only

1

2

5 6

9

10

11

12

13 14

```
before the element can be attached. This example requires some modification to work in
\mathbf{2}
 an environment where the layout of the structures is different on different processes.
3
4
 #define NUM_ELEMS 10
5
6
7
 #define LLIST_ELEM_NEXT_RANK (offsetof(llist_elem_t, next) + \
 offsetof(llist_ptr_t, rank))
8
9
 #define LLIST_ELEM_NEXT_DISP (offsetof(llist_elem_t, next) + \.
10
 offsetof(llist_ptr_t, disp))
11
 /* Linked list pointer */
12
 typedef struct {
13
14
 MPI_Aint disp;
15
 int
 rank;
16
 } llist_ptr_t;
17
 /* Linked list element */
18
19
 typedef struct {
 llist_ptr_t next;
20
 int value;
21
 } llist_elem_t;
22
23
 const llist_ptr_t nil = { (MPI_Aint) MPI_BOTTOM,
^{24}
25
26
 /* List of locally allocated list elements. */
 static llist_elem_t **my_elems = NULL;
27
 static int my_elems_size = 0;
28
 static int my_elems_count = 0;
29
30
 /* Allocate a new shared linked list element */
^{31}
 MPI_Aint alloc_elem(int value, MPI_Win win) {
32
 MPI_Aint disp;
33
34
 llist_elem_t *elem_ptr;
35
 /* Allocate the new element and register it with the window */
36
 MPI_Alloc_mem(sizeof(llist_elem_t), MPI_INFO_NULL, &elem_ptr);
37
 elem_ptr->value = value;
38
 elem_ptr->next = nil;
39
 MPI_Win_attach(win, elem_ptr, sizeof(llist_elem_t));
40
41
42
 /* Add the element to the list of local elements so we can free
 it later. */
43
 if (my_elems_size == my_elems_count) {
44
 my_elems_size += 100;
45
 my_elems = realloc(my_elems, my_elems_size*sizeof(void*));
46
47
 }
 my_elems[my_elems_count] = elem_ptr;
48
```

```
1
 my_elems_count++;
 \mathbf{2}
 3
 MPI_Get_address(elem_ptr, &disp);
 return disp;
 4
}
 5
 6
int main(int argc, char *argv[]) {
 7
 int
 procid, nproc, i;
 9
 MPI_Win
 llist_win;
 10
 llist_ptr_t
 head_ptr, tail_ptr;
 11
 MPI_Init(&argc, &argv);
 12
 13
 14
 MPI_Comm_rank(MPI_COMM_WORLD, &procid);
 15
 MPI_Comm_size(MPI_COMM_WORLD, &nproc);
 16
 MPI_Win_create_dynamic(MPI_INFO_NULL, MPI_COMM_WORLD, &llist_win);
 17
 18
 /* Process 0 creates the head node */
 19
 if (procid == 0)
 20
 21
 head_ptr.disp = alloc_elem(-1, llist_win);
 22
 /* Broadcast the head pointer to everyone */
 23
 24
 head_ptr.rank = 0;
 MPI_Bcast(&head_ptr.disp, 1, MPI_AINT, 0, MPI_COMM_WORLD);
 25
 26
 tail_ptr = head_ptr;
 27
 /* Lock the window for shared access to all targets */
 28
 29
 MPI_Win_lock_all(0, llist_win);
 30
 31
 /* All processes concurrently append NUM_ELEMS elements to the list */
 for (i = 0; i < NUM_ELEMS; i++) {</pre>
 32
 33
 llist_ptr_t new_elem_ptr;
 34
 int success;
 35
 /* Create a new list element and attach it to the window */
 36
 37
 new_elem_ptr.rank = procid;
 new_elem_ptr.disp = alloc_elem(procid, llist_win);
 38
 39
 /* Append the new node to the list. This might take multiple
 40
 41
 attempts if others have already appended and our tail pointer
 42
 is stale. */
 do {
 43
 44
 llist_ptr_t next_tail_ptr = nil;
 45
 46
 MPI_Compare_and_swap((void*) &new_elem_ptr.rank, (void*) &nil.rank,
 47
 (void*)&next_tail_ptr.rank, MPI_INT, tail_ptr.rank,
 48
 MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_RANK),
```

```
1
 llist_win);
2
3
 MPI_Win_flush(tail_ptr.rank, llist_win);
4
 success = (next_tail_ptr.rank == nil.rank);
5
6
 if (success) {
7
 MPI_Accumulate(&new_elem_ptr.disp, 1, MPI_AINT, tail_ptr.rank,
8
 MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_DISP), 1,
9
 MPI_AINT, MPI_REPLACE, llist_win);
10
11
 MPI_Win_flush(tail_ptr.rank, llist_win);
12
 tail_ptr = new_elem_ptr;
13
14
 } else {
15
 /* Tail pointer is stale, fetch the displacement.
 May take
16
 multiple tries if it is being updated. */
17
 do {
 MPI_Get_accumulate(NULL, 0, MPI_AINT, &next_tail_ptr.disp,
18
19
 1, MPI_AINT, tail_ptr.rank,
20
 MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_DISP),
21
 1, MPI_AINT, MPI_NO_OP, llist_win);
22
23
 MPI_Win_flush(tail_ptr.rank, llist_win);
24
 } while (next_tail_ptr.disp == nil.disp);
25
 tail_ptr = next_tail_ptr;
26
 }
27
 } while (!success);
 }
28
29
30
 MPI_Win_unlock_all(llist_win);
^{31}
 MPI_Barrier(MPI_COMM_WORLD);
32
33
 /* Free all the elements in the list */
34
 for (; my_elems_count > 0; my_elems_count--) {
35
 MPI_Win_detach(llist_win,my_elems[my_elems_count-1]);
36
 MPI_Free_mem(my_elems[my_elems_count-1]);
37
 }
38
 MPI_Win_free(&llist_win);
39
 . . .
40
41
42
43
44
45
46
47
48
```

# 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. Next, 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

 $45 \\ 46$ 

```
1
 MPI_GREQUEST_COMPLETE when the operation completes. MPI maintains the "comple-
\mathbf{2}
 tion" status of generalized requests. Any other request state has to be maintained by the
3
 user.
4
 A new generalized request is started with
5
6
 MPI_GREQUEST_START(query_fn, free_fn, cancel_fn, extra_state, request)
7
8
 IN
 callback function invoked when request status is
 query_fn
9
 queried (function)
10
 IN
 free_fn
 callback function invoked when request is freed
11
 (function)
12
 callback function invoked when request is cancelled
 IN
 cancel_fn
13
 (function)
14
15
 IN
 extra_state
 extra state
16
 OUT
 generalized request (handle)
 request
17
18
 C binding
19
 int MPI_Grequest_start(MPI_Grequest_query_function *query_fn,
20
 MPI_Grequest_free_function *free_fn,
21
 MPI_Grequest_cancel_function *cancel_fn, void *extra_state,
22
 MPI_Request *request)
23
^{24}
 Fortran 2008 binding
25
 MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,
26
 ierror)
27
 PROCEDURE(MPI_Grequest_query_function), INTENT(IN) :: query_fn
28
 PROCEDURE(MPI_Grequest_free_function), INTENT(IN) :: free_fn
29
 PROCEDURE(MPI_Grequest_cancel_function), INTENT(IN) :: cancel_fn
30
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
31
 TYPE(MPI_Request), INTENT(OUT) :: request
32
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
 Fortran binding
34
 MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST,
35
 IERROR)
36
 EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN
37
 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
38
 INTEGER REQUEST, IERROR
39
40
41
 Advice to users.
 Note that a generalized request is of the same type as regular
42
 requests, in C and Fortran. (End of advice to users.)
43
 The call starts a generalized request and returns a handle to it in request.
44
 The syntax and meaning of the callback functions are listed below. All callback func-
45
 tions are passed the extra_state argument that was associated with the request by the
46
47
 starting call MPI_GREQUEST_START; extra_state can be used to maintain user-defined
48
 state for the request.
```

```
1
 In C, the query function is
 2
typedef int MPI_Grequest_query_function(void *extra_state,
 MPI_Status *status);
in Fortran with the mpi_f08 module
 5
ABSTRACT INTERFACE
 SUBROUTINE MPI_Grequest_query_function(extra_state, status, ierror)
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
 TYPE(MPI_Status) :: status
 9
 INTEGER :: ierror
 10
 11
in Fortran with the mpi module and mpif.h
SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)
 12
 13
 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
 14
 INTEGER STATUS(MPI_STATUS_SIZE), IERROR
 15
 The query_fn function computes the status that should be returned for the generalized
 16
request. The status also includes information about successful/unsuccessful cancellation of
 17
the request (result to be returned by MPI_TEST_CANCELLED).
 18
 The query_fn callback is invoked by the MPI_{WAIT TEST}{ANY|SOME|ALL} call that
 19
completed the generalized request associated with this callback. The callback function is
 20
also invoked by calls to MPI_REQUEST_GET_STATUS, if the request is complete when
 21
the call occurs. In both cases, the callback is passed a reference to the corresponding
 22
status variable passed by the user to the MPI call; the status set by the callback function
 23
is returned by the MPI call. If the user provided MPI_STATUS_IGNORE or
 ^{24}
MPI_STATUSES_IGNORE to the MPI function that causes query_fn to be called, then MPI
 25
will pass a valid status object to query fn, and this status will be ignored upon return of the
 26
callback function. Note that query_fn is invoked only after MPI_GREQUEST_COMPLETE
 27
is called on the request; it may be invoked several times for the same generalized request,
 28
e.g., if the user calls MPI_REQUEST_GET_STATUS several times for this request. Note also
 29
that a call to MPI_{WAIT|TEST}{SOME|ALL} may cause multiple invocations of query_fn
 30
callback functions, one for each generalized request that is completed by the MPI call. The
 ^{31}
order of these invocations is not specified by MPI.
 32
 In C, the free function is
 33
typedef int MPI_Grequest_free_function(void *extra_state);
 34
 35
in Fortran with the mpi_f08 module
 36
ABSTRACT INTERFACE
 37
 SUBROUTINE MPI_Grequest_free_function(extra_state, ierror)
 38
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
 39
 INTEGER :: ierror
 40
in Fortran with the mpi module and mpif.h
 41
SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)
 42
 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
 43
 INTEGER IERROR
 44
 45
The free_fn function is invoked to clean up user-allocated resources when the generalized
 46
request is freed.
 47
```

1 The free_fn callback is invoked by the MPI_{WAIT|TEST}{ANY|SOME|ALL} call that  $\mathbf{2}$ completed the generalized request associated with this callback. free_fn is invoked after 3 the call to query_fn for the same request. However, if the MPI call completed multiple 4 generalized requests, the order in which free_fn callback functions are invoked is not specified  $\mathbf{5}$ by MPI.

6 The free_fn callback is also invoked for generalized requests that are freed by a call 7to MPI_REQUEST_FREE (no call to MPI_{WAIT|TEST}{ANY|SOME|ALL} will occur for 8 such a request). In this case, the callback function will be called either in the MPI call 9 MPI_REQUEST_FREE(request), or in the MPI call MPI_GREQUEST_COMPLETE(request), 10 whichever happens last, i.e., in this case the actual freeing code is executed as soon as both 11calls MPI_REQUEST_FREE and MPI_GREQUEST_COMPLETE have occurred. The request 12is not deallocated until after free_fn completes. Note that free_fn will be invoked only once 13per request by a correct program.

Advice to users. Calling MPI_REQUEST_FREE(request) will cause the request handle 15to be set to MPI_REQUEST_NULL. This handle to the generalized request is no longer 16valid. However, user copies of this handle are valid until after free_fn completes since 17 MPI does not deallocate the object until then. Since free_fn is not called until after 18 MPI_GREQUEST_COMPLETE, the user copy of the handle can be used to make this 19 call. Users should note that MPI will deallocate the object after free_fn executes. At 20this point, user copies of the request handle no longer point to a valid request. MPI will 21not set user copies to MPI_REQUEST_NULL in this case, so it is up to the user to avoid 22 accessing this stale handle. This is a special case in which MPI defers deallocating the 23object until a later time that is known by the user. (End of advice to users.)  24 

In C, the cancel function is

26typedef int MPI_Grequest_cancel_function(void *extra_state, int complete);

```
in Fortran with the mpi_f08 module
28
```

29ABSTRACT INTERFACE

```
30
 SUBROUTINE MPI_Grequest_cancel_function(extra_state, complete, ierror)
31
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
32
 LOGICAL :: complete
33
 INTEGER :: ierror
```

```
34
 in Fortran with the mpi module and mpif.h
```

```
35
 SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)
36
 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
37
 LOGICAL COMPLETE
38
 INTEGER IERROR
```

40 The cancel_fn function is invoked to start the cancelation of a generalized request. It 41 is called by MPI_CANCEL(request). MPI passes complete = true to the callback function 42if MPI_GREQUEST_COMPLETE was already called on the request, and complete = false 43otherwise.

44All callback functions return an error code. The code is passed back and dealt with as 45appropriate for the error code by the MPI function that invoked the callback function. For 46example, if error codes are returned then the error code returned by the callback function 47will be returned by the MPI function that invoked the callback function. In the case of 48an MPI_{WAIT|TEST}{ANY} call that invokes both query_fn and free_fn, the MPI call will

# Unofficial Draft for Comment Only

14

25

27

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.*)

# Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $\frac{44}{45}$ 

```
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 non-root node receives two messages, sums them, and sends them up. We assume that no status is returned and that the operation cannot be cancelled.

```
7
 typedef struct {
8
 MPI_Comm comm;
9
 int tag;
10
 int root;
11
 int valin;
12
 int *valout;
13
 MPI_Request request;
14
 } ARGS;
15
16
17
 int myreduce(MPI_Comm comm, int tag, int root,
18
 int valin, int *valout, MPI_Request *request)
19
 {
20
 ARGS *args;
21
 pthread_t thread;
22
23
 /* start request */
24
 MPI_Grequest_start(query_fn, free_fn, cancel_fn, NULL, request);
25
26
 args = (ARGS*)malloc(sizeof(ARGS));
27
 args->comm = comm;
28
 args \rightarrow tag = tag;
29
 args->root = root;
30
 args->valin = valin;
31
 args->valout = valout;
32
 args->request = *request;
33
34
 /* spawn thread to handle request */
35
 /* The availability of the pthread_create call is system dependent */
36
 pthread_create(&thread, NULL, reduce_thread, args);
37
 return MPI_SUCCESS;
38
39
 }
40
41
 /* thread code */
42
 void* reduce_thread(void *ptr)
43
 ſ
44
 int lchild, rchild, parent, lval, rval, val;
45
 MPI_Request req[2];
46
 ARGS *args;
47
48
 args = (ARGS*)ptr;
```

1

2 3

4

5

```
2
 /* compute left and right child and parent in tree; set
 to MPI_PROC_NULL if does not exist */
 /* code not shown */
 . . .
 MPI_Irecv(&lval, 1, MPI_INT, lchild, args->tag, args->comm, &req[0]);
 MPI_Irecv(&rval, 1, MPI_INT, rchild, args->tag, args->comm, &req[1]);
 MPI_Waitall(2, req, MPI_STATUSES_IGNORE);
 a
 10
 val = lval + args->valin + rval;
 11
 MPI_Send(&val, 1, MPI_INT, parent, args->tag, args->comm);
 if (parent == MPI_PROC_NULL) *(args->valout) = val;
 12
 MPI_Grequest_complete((args->request));
 13
 14
 free(ptr);
 15
 return(NULL);
 16
}
 17
 18
int query_fn(void *extra_state, MPI_Status *status)
 19
Ł
 /* always send just one int */
 20
 21
 MPI_Status_set_elements(status, MPI_INT, 1);
 /* can never cancel so always true */
 22
 23
 MPI_Status_set_cancelled(status, 0);
 24
 /* choose not to return a value for this */
 25
 status->MPI_SOURCE = MPI_UNDEFINED;
 26
 /* tag has no meaning for this generalized request */
 status->MPI_TAG = MPI_UNDEFINED;
 27
 /* this generalized request never fails */
 28
 29
 return MPI_SUCCESS;
}
 30
 31
 32
 33
int free_fn(void *extra_state)
 34
{
 /* this generalized request does not need to do any freeing */
 35
 /* as a result it never fails here */
 36
 37
 return MPI_SUCCESS;
 38
}
 39
 40
 41
int cancel_fn(void *extra_state, int complete)
 42
{
 /* This generalized request does not support cancelling.
 43
 44
 Abort if not already done. If done then treat as if cancel failed.*/
 45
 if (!complete) {
 46
 fprintf(stderr,
 47
 "Cannot cancel generalized request - aborting program\n");
 48
 MPI_Abort(MPI_COMM_WORLD, 99);
```

```
}
return MPI_SUCCESS;
}
```

# 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:

```
25
26
```

MPI_STATUS_SET_ELEMENTS(status, datatype, count)

			, , , , , , , , , , , , , , , , , , , ,
27 28	INOUT	status	status with which to associate count (status)
29	IN	datatype	datatype associated with count (handle)
30	IN	count	number of elements to associate with status (integer)
31			
32	C binding	g	
33	int MPI_S	- Status_set_element:	s(MPI_Status *status, MPI_Datatype datatype,
34 35		int count)	*
35 36	Fortran 2	2008 binding	
37		Ŭ	atus, datatype, count, ierror)
38			NT(INOUT) :: status
39	TYPE(	(MPI_Datatype), IN	TENT(IN) :: datatype
40	INTEG	SER, INTENT(IN) ::	count
41	INTEG	ER, OPTIONAL, INT	ENT(OUT) :: ierror
42	Fortran k	oinding	
43		0	ATUS, DATATYPE, COUNT, IERROR)
44	INTEG	ER STATUS(MPI_STAT	TUS_SIZE), DATATYPE, COUNT, IERROR
45 46			
40 47			
48			

1

 $\mathbf{2}$ 

3

4 5

> 6 7

8

9

10

11

MPI_STAT	US_SET_ELEMENTS_X(stat	us, datatype, count)	1
INOUT	status	status with which to associate count (status)	2
IN	datatype	datatype associated with count (handle)	3 4
IN	count	number of elements to associate with status (integer)	5
	count	number of clements to associate with status (meeger)	6
C bindin	g		7
	-	_Status *status, MPI_Datatype datatype,	8
	MPI_Count count)		9
Fortran 2	2008 binding		10 11
	0	datatype, count, ierror)	12
	(MPI_Status), INTENT(INOU		13
	(MPI_Datatype), INTENT(IN		14
	GER(KIND=MPI_COUNT_KIND),		15
INTEC	GER, OPTIONAL, INTENT(OUT	) :: lerror	16
Fortran h			17 18
		DATATYPE, COUNT, IERROR)	19
	GER STATUS(MPI_STATUS_SIZ) GER(KIND=MPI_COUNT_KIND)		20
			21
	· · · ·	he part of status so that a call to	22
		_EMENTS_X will return count. MPI_GET_COUNT	23
will return	a compatible value.		24 25
Rati	onale. The number of eleme	ents is set instead of the count because the former	26
can		er of datatypes. (End of rationale.)	27
			28
	_	OUNT(status, datatype, count),	29
	_ELEMENTS(status, datatype, ELEMENTS_X(status_dataty	rpe, count) must use a datatype argument that has	30
		be argument that was used in the call to	31 32
		_STATUS_SET_ELEMENTS_X.	33
			34
	-	matching type signatures for these calls is similar	35
		n count is set by a receive operation: in that case, PI_GET_ELEMENTS, and MPI_GET_ELEMENTS_X	36
		e signature as the datatype used in the receive call.	37
	l of rationale.)		38
× ×			39 40
			41
MPI_STAT	US_SET_CANCELLED(status	s, flag)	42
- INOUT	status	status with which to associate cancel flag (status)	43
			44
IN	flag	if true, indicates request was cancelled (logical)	45
C bindin	σ		46 47
	•	Status *status, int flag)	48

Unofficial Draft for Comment Only

1 2 3 4 5	<pre>Fortran 2008 binding MPI_Status_set_cancelled(status, flag, ierror)     TYPE(MPI_Status), INTENT(INOUT) :: status     LOGICAL, INTENT(IN) :: flag     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>
6 7 8 9 10	Fortran binding MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), IERROR LOGICAL FLAG
11 12 13	If flag is set to true then a subsequent call to MPI_TEST_CANCELLED(status, flag) will also return flag = true, otherwise it will return false.
14 15 16 17 18 19 20	Advice to users. Users are advised not to reuse the status fields for values other than those for which they were intended. Doing so may lead to unexpected results when using the status object. For example, calling MPI_GET_ELEMENTS may cause an error if the value is out of range or it may be impossible to detect such an error. The extra_state argument provided with a generalized request can be used to return information that does not logically belong in status. Furthermore, modifying the values in a status set internally by MPI, e.g., MPI_RECV, may lead to unpredictable provide and is strength discours and (End of advice to come)
21 22	results and is strongly discouraged. (End of advice to users.)
23	
24	
25	
26	
27	
28 29	
29 30	
31	
32	
33	
34	
35	
36	
37	
38	
39	
40	
41	
42	
43	
44	
45	
46	
47	
48	

# Chapter 14

# I/O

# 14.1 Introduction

POSIX provides a model of a widely portable file system, but the portability and optimization needed for parallel I/O cannot be achieved with the POSIX interface.

The significant optimizations required for efficiency (e.g., grouping [54], collective buffering [8, 16, 55, 59, 66], and disk-directed I/O [48]) can only be implemented if the parallel I/O system provides a high-level interface supporting partitioning of file data among processes and a collective interface supporting complete transfers of global data structures between process memories and files. In addition, further efficiencies can be gained via support for asynchronous I/O, strided accesses, and control over physical file layout on storage devices (disks). The I/O environment described in this chapter provides these facilities.

Instead of defining I/O access modes to express the common patterns for accessing a shared file (broadcast, reduction, scatter, gather), we chose another approach in which data partitioning is expressed using derived datatypes. Compared to a limited set of predefined access patterns, this approach has the advantage of added flexibility and expressiveness.

# 14.1.1 Definitions

- file An MPI file is an ordered collection of typed data items. MPI supports random or sequential access to any integral set of these items. A file is opened collectively by a group of processes. All collective I/O calls on a file are collective over this group.
- **displacement** A file *displacement* is an absolute byte position relative to the beginning of a file. The displacement defines the location where a *view* begins. Note that a "file displacement" is distinct from a "typemap displacement."
- etype An *etype* (*elementary* datatype) is the unit of data access and positioning. It can be any MPI predefined or derived datatype. Derived etypes can be constructed using any of the MPI datatype constructor routines, provided all resulting typemap displacements are non-negative and monotonically nondecreasing. Data access is performed in etype units, reading or writing whole data items of type etype. Offsets are expressed as a count of etypes; file pointers point to the beginning of etypes. Depending on context, the term "etype" is used to describe one of three aspects of an elementary datatype: a particular MPI type, a data item of that type, or the extent of that type.

 24 

**filetype** A *filetype* is the basis for partitioning a file among processes and defines a template 2 for accessing the file. A filetype is either a single etype or a derived MPI datatype 3 constructed from multiple instances of the same etype. In addition, the extent of any 4 hole in the filetype must be a multiple of the etype's extent. The displacements in the 5typemap of the filetype are not required to be distinct, but they must be non-negative 6 and monotonically nondecreasing.

**view** A *view* defines the current set of data visible and accessible from an open file as an ordered set of etypes. Each process has its own view of the file, defined by three quantities: a displacement, an etype, and a filetype. The pattern described by a filetype is repeated, beginning at the displacement, to define the view. The pattern of repetition is defined to be the same pattern that MPI_TYPE_CONTIGUOUS would produce if it were passed the filetype and an arbitrarily large count. Figure 14.1 shows how the tiling works; note that the filetype in this example must have explicit lower and upper bounds set in order for the initial and final holes to be repeated in the view. Views can be changed by the user during program execution. The default view is a linear byte stream (displacement is zero, etype and filetype equal to MPI_BYTE).

18	etype
19	
20	filetype
21	L holes
22	
23	tiling a file with the filetype:
24	
25	displacement accessible data
26	
27	Figure 14.1: Etypes and filetypes
28	
29	A group of processes can use complementary views to achieve a global data distribution
30	such as a scatter/gather pattern (see Figure $14.2$ ).
31	etype
32	process 0 filetype
33	
34	process 1 filetype
35	process 2 filetype
36	
37	tiling a file with the filetypes:
38	
39	displacement
40	
41	Figure 14.2: Partitioning a file among parallel processes
42	
43	offset An affect is a position in the file relative to the current view, expressed as a count of

offset An offset is a position in the file relative to the current view, expressed as a count of 44etypes. Holes in the view's filetype are skipped when calculating this position. Offset 0 45is the location of the first etype visible in the view (after skipping the displacement and 46any initial holes in the view). For example, an offset of 2 for process 1 in Figure 14.2 is the position of the eighth etype in the file after the displacement. An "explicit offset" is an offset that is used as an argument in explicit data access routines.

# **Unofficial Draft for Comment Only**

1

7

8

9

10

11

12

13

14

15

16

17

47

- file size and end of file The *size* of an MPI file is measured in bytes from the beginning of the file. A newly created file has a size of zero bytes. Using the size as an absolute displacement gives the position of the byte immediately following the last byte in the file. For any given view, the *end of file* is the offset of the first etype accessible in the current view starting after the last byte in the file.
- file pointer A *file pointer* is an implicit offset maintained by MPI. "Individual file pointers" are file pointers that are local to each process that opened the file. A "shared file pointer" is a file pointer that is shared by the group of processes that opened the file.
- file handle A *file handle* is an opaque object created by MPI_FILE_OPEN and freed by MPI_FILE_CLOSE. All operations on an open file reference the file through the file handle.

# 14.2 File Manipulation

14.2.1 Opening a File

MPI_FILE_OPEN(comm, filename, amode, info, fh)					
IN	comm	communicator (handle)			
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

# Fortran 2008 binding

MPI_File_open(comm, filename, amode, info, fh, ierror)
 TYPE(MPI_Comm), INTENT(IN) :: comm
 CHARACTER(LEN=*), INTENT(IN) :: filename
 INTEGER, INTENT(IN) :: amode
 TYPE(MPI_Info), INTENT(IN) :: info
 TYPE(MPI_File), INTENT(OUT) :: fh
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

# Fortran binding

MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR)
INTEGER COMM, AMODE, INFO, FH, IERROR
CHARACTER*(*) FILENAME

MPI_FILE_OPEN opens the file identified by the file name filename on all processes in the comm communicator group. MPI_FILE_OPEN is a collective routine: all processes must provide the same value for amode, and all processes must provide filenames that reference

1 the same file. (Values for info may vary.) comm must be an intra-communicator; it is  $\mathbf{2}$ erroneous to pass an inter-communicator to MPI_FILE_OPEN. Errors in MPI_FILE_OPEN 3 are raised using the default file error handler (see Section 14.7). When using the World 4 Model (Section 11.1), a process can open a file independently of other processes by using 5the MPI_COMM_SELF communicator. Applications using the Sessions Model (Section 11.3) 6 can achieve the same result using communicators created from the "mpi://SELF" process  $\overline{7}$ set. The file handle returned, fh, can be subsequently used to access the file until the file is 8 closed using MPI_FILE_CLOSE. Before calling MPI_FINALIZE, the user is required to close 9 (via MPI_FILE_CLOSE) all files that were opened with MPI_FILE_OPEN. Note that the 10 communicator comm is unaffected by MPI_FILE_OPEN and continues to be usable in all 11MPI routines (e.g., MPI_SEND). Furthermore, the use of comm will not interfere with I/O 12behavior.

The format for specifying the file name in the filename argument is implementation dependent and must be documented by the implementation.

Advice to implementors. An implementation may require that filename include a string or strings specifying additional information about the file. Examples include the type of filesystem (e.g., a prefix of ufs:), a remote hostname (e.g., a prefix of machine.univ.edu:), or a file password (e.g., a suffix of /PASSWORD=SECRET). (End of advice to implementors.)

Advice to users. On some implementations of MPI, the file namespace may not be identical from all processes of all applications. For example, "/tmp/foo" may denote different files on different processes, or a single file may have many names, dependent on process location. The user is responsible for ensuring that a single file is referenced by the filename argument, as it may be impossible for an implementation to detect this type of namespace error. (*End of advice to users.*)

27 28

29

30

 31 

32

33

34

35 36

37 38

39

40 41

42 43

44

 $45 \\ 46$ 

47 48

15

16

17

18

19

20 21

22

23

24

25

26

Initially, all processes view the file as a linear byte stream, and each process views data in its own native representation (no data representation conversion is performed). (POSIX files are linear byte streams in the native representation.) The file view can be changed via the MPI_FILE_SET_VIEW routine.

The following access modes are supported (specified in amode, a bit vector OR of the following integer constants):

- MPI_MODE_RDONLY—read only,
- MPI_MODE_RDWR—reading and writing,
- MPI_MODE_WRONLY—write only,
- MPI_MODE_CREATE—create the file if it does not exist,
- MPI_MODE_EXCL—error if creating file that already exists,
- MPI_MODE_DELETE_ON_CLOSE—delete file on close,
- MPI_MODE_UNIQUE_OPEN—file will not be concurrently opened elsewhere,
- MPI_MODE_SEQUENTIAL—file will only be accessed sequentially,
  - MPI_MODE_APPEND—set initial position of all file pointers to end of file.

Advice to users. C users can use bit vector OR (|) to combine these constants; Fortran 90 users can use the bit vector IOR intrinsic. Fortran 77 users can use (nonportably) bit vector IOR on systems that support it. Alternatively, Fortran users can portably use integer addition to OR the constants (each constant should appear at most once in the addition.). (End of advice to users.)

Advice to implementors. The values of these constants must be defined such that the bitwise OR and the sum of any distinct set of these constants is equivalent. (End of advice to implementors.)

The modes MPI_MODE_RDONLY, MPI_MODE_RDWR, MPI_MODE_WRONLY, MPI_MODE_CREATE, and MPI_MODE_EXCL have identical semantics to their POSIX counterparts [44]. Exactly one of MPI_MODE_RDONLY, MPI_MODE_RDWR, or MPI_MODE_WRONLY, must be specified. It is erroneous to specify MPI_MODE_CREATE or MPI_MODE_EXCL in conjunction with MPI_MODE_RDONLY; it is erroneous to specify MPI_MODE_SEQUENTIAL together with MPI_MODE_RDWR.

The MPI_MODE_DELETE_ON_CLOSE mode causes the file to be deleted (equivalent to performing an MPI_FILE_DELETE) when the file is closed.

The MPI_MODE_UNIQUE_OPEN mode allows an implementation to optimize access by eliminating the overhead of file locking. It is erroneous to open a file in this mode unless the file will not be concurrently opened elsewhere.

Advice to users. For MPI_MODE_UNIQUE_OPEN, not opened elsewhere includes both inside and outside the MPI environment. In particular, one needs to be aware of potential external events which may open files (e.g., automated backup facilities). When MPI_MODE_UNIQUE_OPEN is specified, the user is responsible for ensuring that no such external events take place. (End of advice to users.)

The MPI_MODE_SEQUENTIAL mode allows an implementation to optimize access to some sequential devices (tapes and network streams). It is erroneous to attempt nonsequential access to a file that has been opened in this mode.

Specifying MPI_MODE_APPEND only guarantees that all shared and individual file pointers are positioned at the initial end of file when MPI_FILE_OPEN returns. Subsequent positioning of file pointers is application dependent. In particular, the implementation does not ensure that all writes are appended.

Errors related to the access mode are raised in the class MPI_ERR_AMODE.

The info argument is used to provide information regarding file access patterns and file system specifics (see Section 14.2.8). The constant MPI_INFO_NULL can be used when no info needs to be specified.

Advice to users. Some file attributes are inherently implementation dependent (e.g., file permissions). These attributes must be set using either the info argument or facilities outside the scope of MPI. (End of advice to users.)

Files are opened by default using nonatomic mode file consistency semantics (see Section 14.6.1). The more stringent atomic mode consistency semantics, required for atomicity of conflicting accesses, can be set using MPI_FILE_SET_ATOMICITY.

Unofficial Draft for Comment Only

 $11 \\ 12$ 

 24 

 $\overline{7}$ 

```
1
 14.2.2 Closing a File
\mathbf{2}
3
4
 MPI_FILE_CLOSE(fh)
5
 INOUT
 fh
 file handle (handle)
6
7
 C binding
8
 int MPI_File_close(MPI_File *fh)
9
10
 Fortran 2008 binding
11
 MPI_File_close(fh, ierror)
12
 TYPE(MPI_File), INTENT(INOUT) :: fh
13
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
 Fortran binding
15
16
 MPI_FILE_CLOSE(FH, IERROR)
17
 INTEGER FH, IERROR
18
 MPI_FILE_CLOSE first synchronizes file state (equivalent to performing an
19
 MPI_FILE_SYNC), then closes the file associated with fh. The file is deleted if it was
20
 opened with access mode MPI_MODE_DELETE_ON_CLOSE (equivalent to performing an
21
 MPI_FILE_DELETE). MPI_FILE_CLOSE is a collective routine.
22
23
 Advice to users. If the file is deleted on close, and there are other processes currently
^{24}
 accessing the file, the status of the file and the behavior of future accesses by these
25
 processes are implementation dependent. (End of advice to users.)
26
27
 The user is responsible for ensuring that all outstanding nonblocking requests and
28
 split collective operations associated with fh made by a process have completed before that
29
 process calls MPI_FILE_CLOSE.
30
 The MPI_FILE_CLOSE routine deallocates the file handle object and sets fh to
^{31}
 MPI_FILE_NULL.
32
33
 Deleting a File
 14.2.3
34
35
36
 MPI_FILE_DELETE(filename, info)
37
 IN
 filename
 name of file to delete (string)
38
39
 IN
 info
 info object (handle)
40
41
 C binding
42
 int MPI_File_delete(const char *filename, MPI_Info info)
43
 Fortran 2008 binding
44
 MPI_File_delete(filename, info, ierror)
45
46
 CHARACTER(LEN=*), INTENT(IN) :: filename
47
 TYPE(MPI_Info), INTENT(IN) :: info
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

#### Fortran binding

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

MPI_FILE_DELETE deletes the file identified by the file name filename. If the file does not exist, MPI_FILE_DELETE raises an error in the class MPI_ERR_NO_SUCH_FILE.

The info argument can be used to provide information regarding file system specifics (see Section 14.2.8). The constant MPI_INFO_NULL refers to the null info, and can be used when no info needs to be specified.

If a process currently has the file open, the behavior of any access to the file (as well as the behavior of any outstanding accesses) is implementation dependent. In addition, whether an open file is deleted or not is also implementation dependent. If the file is not deleted, an error in the class MPI_ERR_FILE_IN_USE or MPI_ERR_ACCESS will be raised. Errors are raised using the default file error handler (see Section 14.7).

14.2.4 Resizing a File

MPI_FILE_S	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

## Unofficial Draft for Comment Only

 $\frac{4}{5}$ 

1 pointer. If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is  $\mathbf{2}$ erroneous to call this routine. 3 4 Advice to users. It is possible for the file pointers to point beyond the end of file after a MPI_FILE_SET_SIZE operation truncates a file. This is valid, and equivalent 5to seeking beyond the current end of file. (End of advice to users.) 6 7 All nonblocking requests and split collective operations on fh must be completed before 8 calling MPI_FILE_SET_SIZE. Otherwise, calling MPI_FILE_SET_SIZE is erroneous. As far 9 as consistency semantics are concerned, MPI_FILE_SET_SIZE is a write operation that 10 conflicts with operations that access bytes at displacements between the old and new file 11 sizes (see Section 14.6.1). 121314.2.5 Preallocating Space for a File 14151617MPI_FILE_PREALLOCATE(fh, size) 18 INOUT fh file handle (handle) 19 IN size to preallocate file (integer) size 202122 C binding 23int MPI_File_preallocate(MPI_File fh, MPI_Offset size)  24 Fortran 2008 binding 25MPI_File_preallocate(fh, size, ierror) 26TYPE(MPI_File), INTENT(IN) :: fh 27INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size 28INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2930 Fortran binding  31 MPI_FILE_PREALLOCATE(FH, SIZE, IERROR) 32 INTEGER FH, IERROR 33 INTEGER(KIND=MPI_OFFSET_KIND) SIZE 34 MPI_FILE_PREALLOCATE ensures that storage space is allocated for the first size bytes 35 of the file associated with fh. MPI_FILE_PREALLOCATE is collective; all processes in the 36 group must pass identical values for size. Regions of the file that have previously been 37 written are unaffected. For newly allocated regions of the file, MPI_FILE_PREALLOCATE 38 has the same effect as writing undefined data. If size is larger than the current file size, the 39 file size increases to size. If size is less than or equal to the current file size, the file size is 40 unchanged.

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.  $(End \ of \ advice \ to \ users.)$ 

47 48

45

14.2.6 Querying the Size of a File		1		
		2		
		3		
MPI_FILE_GET_SIZE(fh, size)		4		
IN fh	file handle (handle)	5 6		
OUT size	size of the file in bytes (integer)	7		
	she of the fits in 59 tes (moger)	8		
C binding		9		
int MPI_File_get_size(MPI_File fh,	MPI_Offset *size)	10		
Fortran 2008 binding		11 12		
MPI_File_get_size(fh, size, ierror)				
TYPE(MPI_File), INTENT(IN) ::		13 14		
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: size				
INTEGER, OPTIONAL, INTENT(OUT)	:: ierror	16		
Fortran binding		17		
MPI_FILE_GET_SIZE(FH, SIZE, IERROF		18		
INTEGER FH, IERROR		19		
INTEGER(KIND=MPI_OFFSET_KIND)	SIZE	20 21		
MPI_FILE_GET_SIZE returns, in size	e, the current size in bytes of the file associated with	21		
the file handle fh. As far as consistency s	semantics are concerned, MPI_FILE_GET_SIZE is a	23		
data access operation (see Section 14.6.1).				
		25		
14.2.7 Querying File Parameters		26		
		27		
		28 29		
MPI_FILE_GET_GROUP(fh, group)		30		
IN fh	file handle (handle)	31		
OUT group	group which opened the file (handle)	32		
		33		
C binding		34		
<pre>int MPI_File_get_group(MPI_File fh</pre>	n, MPI_Group *group)	$\frac{35}{36}$		
Fortran 2008 binding		30 37		
<pre>MPI_File_get_group(fh, group, ierr</pre>		38		
TYPE(MPI_File), INTENT(IN) ::		39		
TYPE(MPI_Group), INTENT(OUT) :		40		
INTEGER, OPTIONAL, INTENT(OUT)		41		
Fortran binding				
MPI_FILE_GET_GROUP(FH, GROUP, IERF	ROR)	43 44		
INTEGER FH, GROUP, IERROR		44 45		
MPI_FILE_GET_GROUP returns a duplicate of the group of the communicator used to open the file associated with <b>fb</b> . The group is returned in group. The user is responsible for				

MPI_FILE_GET_GROUP returns a duplicate of the group of the communicator used to open the file associated with fh. The group is returned in group. The user is responsible for freeing group.

47

```
1
 MPI_FILE_GET_AMODE(fh, amode)
2
 IN
 fh
 file handle (handle)
3
 OUT
 amode
 file access mode used to open the file (integer)
4
5
6
 C binding
7
 int MPI_File_get_amode(MPI_File fh, int *amode)
8
 Fortran 2008 binding
9
 MPI_File_get_amode(fh, amode, ierror)
10
 TYPE(MPI_File), INTENT(IN) :: fh
11
 INTEGER, INTENT(OUT) :: amode
12
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
 Fortran binding
 MPI_FILE_GET_AMODE(FH, AMODE, IERROR)
15
16
 INTEGER FH, AMODE, IERROR
17
 MPI_FILE_GET_AMODE returns, in amode, the access mode of the file associated with
18
 fh.
19
20
 Example 14.1 In Fortran 77, decoding an amode bit vector will require a routine such as
21
 the following:
22
23
 SUBROUTINE BIT_QUERY(TEST_BIT, MAX_BIT, AMODE, BIT_FOUND)
^{24}
 !
25
 !
 TEST IF THE INPUT TEST_BIT IS SET IN THE INPUT AMODE
26
 IF SET, RETURN 1 IN BIT_FOUND, O OTHERWISE
 !
27
 !
28
 INTEGER TEST_BIT, AMODE, BIT_FOUND, CP_AMODE, HIFOUND
29
 BIT_FOUND = 0
30
 CP\_AMODE = AMODE
^{31}
 100 CONTINUE
32
 LBIT = 0
33
 HIFOUND = 0
34
 DO L = MAX_BIT, 0, -1
35
 MATCHER = 2**L
36
 IF (CP_AMODE .GE. MATCHER .AND. HIFOUND .EQ. 0) THEN
37
 HIFOUND = 1
38
 LBIT = MATCHER
39
 CP_AMODE = CP_AMODE - MATCHER
40
 END IF
41
 END DO
42
 IF (HIFOUND .EQ. 1 .AND. LBIT .EQ. TEST_BIT) BIT_FOUND = 1
43
 IF (BIT_FOUND .EQ. O .AND. HIFOUND .EQ. 1 .AND. &
44
 CP_AMODE .GT. 0) GO TO 100
45
 END
46
47
 This routine could be called successively to decode amode, one bit at a time. For
```

⁴⁸ example, the following code fragment would check for MPI_MODE_RDONLY.

```
CALL BIT_QUERY(MPI_MODE_RDONLY, 30, AMODE, BIT_FOUND)
IF (BIT_FOUND .EQ. 1) THEN
PRINT *, ' FOUND READ-ONLY BIT IN AMODE=', AMODE
ELSE
PRINT *, ' READ-ONLY BIT NOT FOUND IN AMODE=', AMODE
END IF
```

# 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. 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.*)

	29			
MPI_FILE_SET_INFO(fh, info)	30			
INOUT fh file handle (handle)	31			
	32			
IN info info object (handle)	33			
	34			
C binding	35			
<pre>int MPI_File_set_info(MPI_File fh, MPI_Info info)</pre>	36			
	37			
Fortran 2008 binding				
MPI_File_set_info(fh, info, ierror)				
TYPE(MPI_File), INTENT(IN) :: fh	40			
TYPE(MPI_Info), INTENT(IN) :: info	41			
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	42			
Fortran binding	43			
MPI_FILE_SET_INFO(FH, INFO, IERROR)				
INTEGER FH, INFO, IERROR	45			
	46			
MPL FILE SET INFO undates the hints of the file associated with the using	the hints			

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

 $\mathbf{2}$ 

 24 

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.*)

file handle (handle)

new info object (handle)

¹⁶ 17 MPI_FILE_GET_INFO(fh, info_used)

7

8

9

10

11

12

13 14 15

18 19

20 21

25 26

27

28

IN

OUT

C binding

```
int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)
```

²⁴ Fortran 2008 binding

fh

info_used

```
MPI_File_get_info(fh, info_used, ierror)
 TYPE(MPI_File), INTENT(IN) :: fh
 TYPE(NPI_L_f(r)) = f(r)
```

TYPE(MPI_Info), INTENT(OUT) :: info_used INTEGER, OPTIONAL, INTENT(OUT) :: ierror

²⁹ Fortran binding

```
MPI_FILE_GET_INFO(FH, INFO_USED, IERROR)
INTEGER FH, INFO_USED, IERROR
```

³³ MPI_FILE_GET_INFO returns a new info object containing the hints of the file associ-³⁴ ated with fh. The current setting of all hints related to this file is returned in info_used. An ³⁵ MPI implementation is required to return all hints that are supported by the implementa-³⁶ tion and have default values specified; any user-supplied hints that were not ignored by the ³⁷ implementation; and any additional hints that were set by the implementation. If no such ³⁸ hints exist, a handle to a newly created info object is returned that contains no (key,value) ³⁹ pairs. The user is responsible for freeing info_used via MPI_INFO_FREE.

- 40
- ⁴¹ Reserved File Hints

⁴² ⁴³ Some potentially useful hints (info key values) are outlined below. The following key values ⁴⁴ are reserved. An implementation is not required to interpret these key values, but if it does ⁴⁵ interpret the key value, it must provide the functionality described. (For more details on ⁴⁶ "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

# Unofficial Draft for Comment Only

value. The "[**SAME**]" annotation specifies that the hint values provided by all participating processes must be identical; otherwise the program is erroneous. In addition, some hints are context dependent, and are only used by an implementation at specific times (e.g., "file_perm" is only useful during file creation).

- "access_style" (comma separated list of strings): This hint specifies the manner in which the file will be accessed until the file is closed or until the "access_style" key value is altered. The hint value is a comma separated list of the following: "read_once", "write_once", "read_mostly", "write_mostly", "sequential", "reverse_sequential", and "random".
- "collective_buffering" (boolean) [SAME]: This hint specifies whether the application may benefit from collective buffering. Collective buffering is an optimization performed on collective accesses. Accesses to the file are performed on behalf of all processes in the group by a number of target nodes. These target nodes coalesce small requests into large disk accesses. Valid values for this key are "true" and "false". Collective buffering parameters are further directed via additional hints: "cb_block_size", "cb_buffer_size", and "cb_nodes".
- "cb_block_size" (integer) [SAME]: This hint specifies the block size to be used for collective buffering file access. *Target nodes* access data in chunks of this size. The chunks are distributed among target nodes in a round-robin (cyclic) pattern.
- "cb_buffer_size" (integer) [SAME]: This hint specifies the total buffer space that can be used for collective buffering on each target node, usually a multiple of "cb_block_size".
- "cb_nodes" (integer) [SAME]: This hint specifies the number of target nodes to be used for collective buffering.
- "chunked" (comma separated list of integers) [SAME]: This hint specifies that the file consists of a multidimentional array that is often accessed by subarrays. The value for this hint is a comma separated list of array dimensions, starting from the most significant one (for an array stored in row-major order, as in C, the most significant dimension is the first one; for an array stored in column-major order, as in Fortran, the most significant dimension is the last one, and array dimensions should be reversed).
- "chunked_item" (comma separated list of integers) [SAME]: This hint specifies the size of each array entry, in bytes.
- "chunked_size" (comma separated list of integers) [SAME]: This hint specifies the dimensions of the subarrays. This is a comma separated list of array dimensions, starting from the most significant one.
- "filename" (string): This hint specifies the file name used when the file was opened. If the implementation is capable of returning the file name of an open file, it will be returned using this key by MPI_FILE_GET_INFO. This key is ignored when passed to MPI_FILE_OPEN, MPI_FILE_SET_VIEW, MPI_FILE_SET_INFO, and MPI_FILE_DELETE.
- "file_perm" (string) [SAME]: This hint specifies the file permissions to use for file creation. Setting this hint is only useful when passed to MPI_FILE_OPEN with an amode

#### Unofficial Draft for Comment Only

 $\overline{7}$ 

 31 

1 2		includes MPI_MOI dependent.	DE_CREATE. The set of valid values for this key is implementa-	
$3 \\ 4 \\ 5 \\ 6$	of I/	"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.		
7 8 9 10	will	"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.		
11 12			<b>SAME</b> ]: This hint specifies the number of I/O devices in the ost relevant when the file is created.	
13 14 15			<b>SAME</b> ]: This hint specifies the number of I/O devices that ed across, and is relevant only when the file is created.	
16 17 18 19 20	used I/O	for this file. The device before prog	<b>AME</b> ]: This hint specifies the suggested striping unit to be striping unit is the amount of consecutive data assigned to one gressing to the next device, when striping across a number of l in bytes. This hint is relevant only when the file is created.	
21 22	14.3 F	ile Views		
23				
24 25		SET VIEW(fb di	sp, etype, filetype, datarep, info)	
26	INOUT	fh	file handle (handle)	
27	IN	disp	displacement (integer)	
28 29	IN	etype	elementary datatype (handle)	
30	IN	filetype	filetype (handle)	
31	IN	datarep	data representation (string)	
32	IN	info	info object (handle)	
33 34	IN	inio	nno object (nandie)	
35	C bindin	g		
36		-	PI_File fh, MPI_Offset disp, MPI_Datatype etype,	
37		MPI_Dataty	pe filetype, const char *datarep, MPI_Info info)	
38	Fortran 2	2008 binding		
39 40			sp, etype, filetype, datarep, info, ierror)	
41		(MPI_File), INTE		
42			SET_KIND), INTENT(IN) :: disp	
43		• -	INTENT(IN) :: etype, filetype	
44		(MPI_Info), INTE	NTENT(IN) :: datarep	
45			INTENT(OUT) :: ierror	
46				
47 48	Fortran l			
	ULT_LTP	_061_016W(FR, D)	ISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)	

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

The MPI_FILE_SET_VIEW routine changes the process's view of the data in the file. The start of the view is set to disp; the type of data is set to etype; the distribution of data to processes is set to filetype; and the representation of data in the file is set to datarep. In addition, MPI_FILE_SET_VIEW resets the individual file pointers and the shared file pointer to zero. MPI_FILE_SET_VIEW is collective; the values for datarep and the extents of etype in the file data representation must be identical on all processes in the group; values for disp, filetype, and info may vary. The datatypes passed in etype and filetype must be committed.

The etype always specifies the data layout in the file. If etype is a portable datatype (see Section 2.4), the extent of etype is computed by scaling any displacements in the datatype to match the file data representation. If etype is not a portable datatype, no scaling is done when computing the extent of etype. The user must be careful when using nonportable etypes in heterogeneous environments; see Section 14.5.1 for further details.

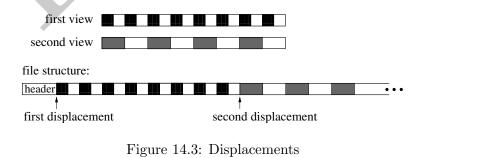
If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, the special displacement MPI_DISPLACEMENT_CURRENT must be passed in disp. This sets the displacement to the current position of the shared file pointer. MPI_DISPLACEMENT_CURRENT is invalid unless the amode for the file has MPI_MODE_SEQUENTIAL set.

Rationale. For some sequential files, such as those corresponding to magnetic tapes or streaming network connections, the *displacement* may not be meaningful. MPI_DISPLACEMENT_CURRENT allows the view to be changed for these types of files. (*End of rationale.*)

Advice to implementors. It is expected that a call to MPI_FILE_SET_VIEW will immediately follow MPI_FILE_OPEN in numerous instances. A high-quality implementation will ensure that this behavior is efficient. (*End of advice to implementors.*)

The disp displacement argument specifies the position (absolute offset in bytes from the beginning of the file) where the view begins.

Advice to users. disp can be used to skip headers or when the file includes a sequence of data segments that are to be accessed in different patterns (see Figure 14.3). Separate views, each using a different displacement and filetype, can be used to access each segment.



(End of advice to users.)

#### Unofficial Draft for Comment Only

 24 

1 An etype (elementary datatype) is the unit of data access and positioning. It can be  $\mathbf{2}$ any MPI predefined or derived datatype. Derived etypes can be constructed by using any 3 of the MPI datatype constructor routines, provided all resulting typemap displacements are 4 non-negative and monotonically nondecreasing. Data access is performed in etype units,  $\mathbf{5}$ reading or writing whole data items of type etype. Offsets are expressed as a count of 6 etypes; file pointers point to the beginning of etypes.

> In order to ensure interoperability in a heterogeneous environ-Advice to users. ment, additional restrictions must be observed when constructing the etype (see Section 14.5). (End of advice to users.)

A filetype is either a single etype or a derived MPI datatype constructed from multiple instances of the same etype. In addition, the extent of any hole in the filetype must be a multiple of the etype's extent. These displacements are not required to be distinct, but they cannot be negative, and they must be monotonically nondecreasing.

If the file is opened for writing, neither the etype nor the filetype is permitted to 16contain overlapping regions. This restriction is equivalent to the "datatype used in a receive 17cannot specify overlapping regions" restriction for communication. Note that filetypes from 18 different processes may still overlap each other. 19

If a filetype has holes in it, then the data in the holes is inaccessible to the calling 20process. However, the disp, etype, and filetype arguments can be changed via future calls to 21MPI_FILE_SET_VIEW to access a different part of the file. 22

It is erroneous to use absolute addresses in the construction of the etype and filetype. 23The info argument is used to provide information regarding file access patterns and file 24system specifics to direct optimization (see Section 14.2.8). The constant MPI_INFO_NULL 25refers to the null info and can be used when no info needs to be specified. 26

The datarep argument is a string that specifies the representation of data in the file. 27See the file interoperability section (Section 14.5) for details and a discussion of valid values. 28

The user is responsible for ensuring that all nonblocking requests and split collective operations on fh have been completed before calling MPI_FILE_SET_VIEW—otherwise, the call to MPI_FILE_SET_VIEW is erroneous.

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

35	IN	fh	file handle (handle)
36	OUT	disp	displacement (integer)
37 38	OUT	etype	elementary datatype (handle)
39	OUT	filetype	filetype (handle)
40	OUT	datarep	data representation (string)
41			

```
C binding
```

```
43
 int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype,
44
 MPI_Datatype *filetype, char *datarep)
45
```

```
Fortran 2008 binding
46
```

```
47
 MPI_File_get_view(fh, disp, etype, filetype, datarep, ierror)
 TYPE(MPI_File), INTENT(IN) :: fh
48
```

7

8

9

10 11

12

13

14

15

29

30

 31 32 33

34

```
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp
TYPE(MPI_Datatype), INTENT(OUT) :: etype, filetype
CHARACTER(LEN=*), INTENT(OUT) :: datarep
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

## Fortran binding

```
MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)
INTEGER FH, ETYPE, FILETYPE, IERROR
INTEGER(KIND=MPI_OFFSET_KIND) DISP
CHARACTER*(*) DATAREP
```

MPI_FILE_GET_VIEW returns the process's view of the data in the file. The current value of the displacement is returned in disp. The etype and filetype are new datatypes with typemaps equal to the typemaps of the current etype and filetype, respectively.

The data representation is returned in datarep. The user is responsible for ensuring that datarep is large enough to hold the returned data representation string. The length of a data representation string is limited to the value of MPI_MAX_DATAREP_STRING.

In addition, if a portable datatype was used to set the current view, then the corresponding datatype returned by MPI_FILE_GET_VIEW is also a portable datatype. If etype or filetype are derived datatypes, the user is responsible for freeing them. The etype and filetype returned are both in a committed state.

# 14.4 Data Access

## 14.4.1 Data Access Routines

Data is moved between files and processes by issuing read and write calls. There are three orthogonal aspects to data access: positioning (explicit offset *vs.* implicit file pointer), synchronism (blocking *vs.* nonblocking and split collective), and coordination (noncollective *vs.* collective). The following combinations of these data access routines, including two types of file pointers (individual and shared) are provided in Table 14.1.

positioning	synchronism		ordination
		noncollective	collective
explicit	blocking	MPI_FILE_READ_AT	MPI_FILE_READ_AT_ALL
offsets		MPI_FILE_WRITE_AT	MPI_FILE_WRITE_AT_ALL
	nonblocking	MPI_FILE_IREAD_AT	MPI_FILE_IREAD_AT_ALL
		MPI_FILE_IWRITE_AT	MPI_FILE_IWRITE_AT_ALL
	split collective	N/A	MPI_FILE_READ_AT_ALL_BEGIN
			MPI_FILE_READ_AT_ALL_END
			MPI_FILE_WRITE_AT_ALL_BEGIN
			MPI_FILE_WRITE_AT_ALL_END
individual	blocking	MPI_FILE_READ	MPI_FILE_READ_ALL
file pointers		MPI_FILE_WRITE	MPI_FILE_WRITE_ALL
	nonblocking	MPI_FILE_IREAD	MPI_FILE_IREAD_ALL
		MPI_FILE_IWRITE	MPI_FILE_IWRITE_ALL
	split collective	N/A	MPI_FILE_READ_ALL_BEGIN
*			MPI_FILE_READ_ALL_END
			MPI_FILE_WRITE_ALL_BEGIN
			MPI_FILE_WRITE_ALL_END
shared	blocking	MPI_FILE_READ_SHARED	MPI_FILE_READ_ORDERED
file pointer		MPI_FILE_WRITE_SHARED	MPI_FILE_WRITE_ORDERED
	nonblocking	MPI_FILE_IREAD_SHARED	N/A
		MPI_FILE_IWRITE_SHARED	
	split collective	N/A	MPI_FILE_READ_ORDERED_BEGIN
			MPI_FILE_READ_ORDERED_END
			MPI_FILE_WRITE_ORDERED_BEG
			MPI_FILE_WRITE_ORDERED_END

Table 14.1: Data access routines

## Unofficial Draft for Comment Only

 $\mathbf{2}$ 

POSIX read()/fread() and write()/fwrite() are blocking, noncollective operations  $\mathbf{2}$ and use individual file pointers. The MPI equivalents are MPI_FILE_READ and 3 MPI_FILE_WRITE. 4 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 6 completes. For writes, however, the MPI_FILE_SYNC routine provides the only guarantee that data has been transferred to the storage device. 9 Positioning 10

MPI provides three types of positioning for data access routines: explicit offsets, indi-11 vidual file pointers, and shared file pointers. The different positioning methods may 12be mixed within the same program and do not affect each other. 13

The data access routines that accept explicit offsets contain _AT in their name (e.g., 14MPI_FILE_WRITE_AT). Explicit offset operations perform data access at the file position 15given directly as an argument—no file pointer is used nor updated. Note that this is not 16equivalent to an atomic seek-and-read or seek-and-write operation, as no "seek" is issued. 17Operations with explicit offsets are described in Section 14.4.2. 18

The names of the individual file pointer routines contain no positional qualifier (e.g., 19MPI_FILE_WRITE). Operations with individual file pointers are described in Section 14.4.3. 20The data access routines that use shared file pointers contain _SHARED or _ORDERED 21in their name (e.g., MPI_FILE_WRITE_SHARED). Operations with shared file pointers are 22 described in Section 14.4.4. 23

The main semantic issues with MPI-maintained file pointers are how and when they are  24 updated by I/O operations. In general, each I/O operation leaves the file pointer pointing to 25the next data item after the last one that is accessed by the operation. In a nonblocking or 26split collective operation, the pointer is updated by the call that initiates the I/O, possibly 27before the access completes. 28

More formally, 29

30  31 

32

37

 $new_file_offset = old_file_offset + \frac{elements(datatype)}{elements(etype)} \times count$ 

where *count* is the number of *datatype* items to be accessed, elements(X) is the number of 33 predefined datatypes in the typemap of X, and  $old_file_offset$  is the value of the implicit 34offset before the call. The file position, *new_file_offset*, is in terms of a count of etypes 35 relative to the current view. 36

Synchronism 38

39 MPI supports blocking and nonblocking I/O routines. 40

A blocking I/O call will not return until the I/O request is completed.

41 A nonblocking I/O call initiates an I/O operation, but does not wait for it to complete. 42Given suitable hardware, this allows the transfer of data out of and into the user's buffer 43 to proceed concurrently with computation. A separate request complete call (MPI_WAIT, 44MPI_TEST, or any of their variants) is needed to complete the I/O request, i.e., to confirm 45that the data has been read or written and that it is safe for the user to reuse the buffer. 46The nonblocking versions of the routines are named MPI_FILE_IXXX, where the I stands 47for immediate. 48

1

5

 $\overline{7}$ 

 $\mathbf{2}$ 

3

4

5

6 7

8

9

10

11

12

13

14

15

16

17

18 19

20 21

22

23

 24 

25

26

27

28

29

30

 31 

32

33

34

35

36

37

38

39

40

41

42

43 44

45

46

It is erroneous to access the local buffer of a nonblocking data access operation, or to use that buffer as the source or target of other communications, between the initiation and completion of the operation.

The split collective routines support a restricted form of "nonblocking" operations for collective data access (see Section 14.4.5).

#### Coordination

Every noncollective data access routine MPI_FILE_XXX has a collective counterpart. For most routines, this counterpart is MPI_FILE_XXX_ALL or a pair of MPI_FILE_XXX_BEGIN and MPI_FILE_XXX_END. The counterparts to the MPI_FILE_XXX_SHARED routines are MPI_FILE_XXX_ORDERED.

The completion of a noncollective call only depends on the activity of the calling process. However, the completion of a collective call (which must be called by all members of the process group) may depend on the activity of the other processes participating in the collective call. See Section 14.6.4 for rules on semantics of collective calls.

Collective operations may perform much better than their noncollective counterparts, as global data accesses have significant potential for automatic optimization.

#### Data Access Conventions

Data is moved between files and processes by calling read and write routines. Read routines move data from a file into memory. Write routines move data from memory into a file. The file is designated by a file handle, fh. The location of the file data is specified by an offset into the current view. The data in memory is specified by a triple: buf, count, and datatype. Upon completion, the amount of data accessed by the calling process is returned in a status.

An offset designates the starting position in the file for an access. The offset is always in etype units relative to the current view. Explicit offset routines pass offset as an argument (negative values are erroneous). The file pointer routines use implicit offsets maintained by MPI.

A data access routine attempts to transfer (read or write) count data items of type datatype between the user's buffer buf and the file. The datatype passed to the routine must be a committed datatype. The layout of data in memory corresponding to buf, count, datatype is interpreted the same way as in MPI communication functions; see Section 3.2.2 and Section 5.1.11. The data is accessed from those parts of the file specified by the current view (Section 14.3). The type signature of datatype must match the type signature of some number of contiguous copies of the etype of the current view. As in a receive, it is erroneous to specify a datatype for reading that contains overlapping regions (areas of memory which would be stored into more than once).

The nonblocking data access routines indicate that MPI can start a data access and associate a request handle, request, with the I/O operation. Nonblocking operations are completed via MPI_TEST, MPI_WAIT, or any of their variants.

Data access operations, when completed, return the amount of data accessed in status.

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

# $47 \\ 48$

1 For blocking routines, status is returned directly. For nonblocking routines and split  $\mathbf{2}$ collective routines, status is returned when the operation is completed. The number of 3 datatype entries and predefined elements accessed by the calling process can be extracted 4 from status by using MPI_GET_COUNT and MPI_GET_ELEMENTS (or

 $\mathbf{5}$ MPI_GET_ELEMENTS_X), respectively. The interpretation of the MPI_ERROR field is the 6 same as for other operations—normally undefined, but meaningful if an MPI routine returns 7MPI_ERR_IN_STATUS. The user can pass (in C and Fortran) MPI_STATUS_IGNORE in the 8 status argument if the return value of this argument is not needed. The status can be 9 passed to MPI_TEST_CANCELLED to determine if the operation was cancelled. All other 10 fields of status are undefined.

11When reading, a program can detect the end of file by noting that the amount of data 12read is less than the amount requested. Writing past the end of file increases the file size. 13The amount of data accessed will be the amount requested, unless an error is raised (or a 14read reaches the end of file).

1614.4.2 Data Access with Explicit Offsets

17If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous to 18 call the routines in this section. 19

2021

22

15

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

23	IN	fh	file handle (handle)
24	IN	offset	file offset (integer)
25	OUT	buf	initial address of buffer (choice)
26 27	IN	count	number of elements in buffer (integer)
28	IN	datatype	datatype of each buffer element (handle)
29 30	OUT	status	status object (status)
31 32	C bindir int MPI_	9	File fh, MPI_Offset offset, void *buf, int count,
33 34		MPI_Dataty	pe datatype, MPI_Status *status)
35 36 37			PI_File fh, MPI_Offset offset, void *buf, count, MPI_Datatype datatype, MPI_Status *status)
38 39 40	MPI_File		set, buf, count, datatype, status, ierror) NT(IN) :: fh
41		CGER(KIND=MPI_OFF C(*), DIMENSION(.	'SET_KIND), INTENT(IN) :: offset .) :: buf
42 43	INTE	GER, INTENT(IN)	:: count
44 45	TYPE	C(MPI_Status) ::	
46 47			NTENT(OUT) :: ierror
48		e_read_at(In, OII	set, buf, count, datatype, status, ierror)

```
TYPE(MPI_File), INTENT(IN) :: fh
```

		FSET_KIND), INTENT(IN) :: offset	1
	PE(*), DIMENSION(.		2
		JNT_KIND), INTENT(IN) :: count	$\frac{3}{4}$
	PE(MPI_Datatype), PE(MPI_Status) ::	INTENT(IN) :: datatype	5
		INTENT(OUT) :: ierror	6
			7
	n binding		8
		SET, BUF, COUNT, DATATYPE, STATUS, IERROR) DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	9
	TEGER(KIND=MPI_OFF		10
	ype> BUF(*)		11 12
	-	ads a file beginning at the position specified by offset.	12
		add a me beginning at the position specified by suber	14
			15
MPI_FI	LE_READ_AI_ALL(†	h, offset, buf, count, datatype, status)	16
IN	fh	file handle (handle)	17
IN	offset	file offset (integer)	18 19
OUT	buf	initial address of buffer (choice)	20
IN	count	number of elements in buffer (integer)	21
IN	datatype	datatype of each buffer element (handle)	22 23
OUT	status	status object (status)	23 24
			25
C binding			
int MP	I_File_read_at_all	L(MPI_File fh, MPI_Offset offset, void *buf,	27
int count, MPI_Datatype datatype, MPI_Status *status) 28			
int MP	I_File_read_at_all	L_c(MPI_File fh, MPI_Offset offset, void *buf,	29 30
MPI Count count MPI Datatype datatype MPI Status *status)			31
			32
		, offset, buf, count, datatype, status, ierror)	33
	PE(MPI_File), INTE		34
		FSET_KIND), INTENT(IN) :: offset	35
	PE(*), DIMENSION(.		36
	TEGER, INTENT(IN)		37 38
	PE(MPI_Datatype), PE(MPI_Status) ::	INTENT(IN) :: datatype	39
		INTENT(OUT) :: ierror	40
			41
	PE(MPI_File), INTE	, offset, buf, count, datatype, status, ierror)	42
	-	FSET_KIND), INTENT(IN) :: offset	43
	PE(*), DIMENSION(.	-	44 45
		<pre>JNT_KIND), INTENT(IN) :: count</pre>	46
		INTENT(IN) :: datatype	47
TYPE(MPI_Status) :: status       48			

```
1
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
 Fortran binding
3
 MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
4
 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
5
 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
6
 <type> BUF(*)
7
8
 MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT
9
 interface.
10
11
 MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status)
12
13
 INOUT
 fh
 file handle (handle)
14
 IN
 offset
 file offset (integer)
15
 IN
 buf
 initial address of buffer (choice)
16
17
 number of elements in buffer (integer)
 IN
 count
18
 IN
 datatype of each buffer element (handle)
 datatype
19
 OUT
 status
 status object (status)
20
21
 C binding
22
 int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf,
23
24
 int count, MPI_Datatype datatype, MPI_Status *status)
25
 int MPI_File_write_at_c(MPI_File fh, MPI_Offset offset, const void *buf,
26
 MPI_Count count, MPI_Datatype datatype, MPI_Status *status)
27
28
 Fortran 2008 binding
 MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror)
29
30
 TYPE(MPI_File), INTENT(IN) :: fh
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
31
 TYPE(*), DIMENSION(..), INTENT(IN) :: buf
32
33
 INTEGER, INTENT(IN) :: count
34
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
35
 TYPE(MPI_Status) :: status
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
 MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror)
38
 TYPE(MPI_File), INTENT(IN) :: fh
39
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
40
 TYPE(*), DIMENSION(..), INTENT(IN) :: buf
41
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
42
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
43
 TYPE(MPI_Status) :: status
44
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
 Fortran binding
47
 MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
48
 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

INTEGER(KIND-MFI_OFFSEI_KIND) OFFSEI			$\frac{1}{2}$
MPI_	$FILE_WRITE_AT$ writes a file	beginning at the position specified by offset.	3 4 5
MPI_FILE	_WRITE_AT_ALL(fh, offset, b	uf, count, datatype, status)	6
INOUT	fh	file handle (handle)	7
IN	offset	file offset (integer)	8 9
IN	buf		10
		initial address of buffer (choice)	11
IN	count	number of elements in buffer (integer)	12
IN	datatype	datatype of each buffer element (handle)	13 14
OUT	status	status object (status)	14
Chindin			16
C bindin	0	e fh, MPI_Offset offset, const void *buf,	17
1110 111 1_		type datatype, MPI_Status *status)	18
int MDT			19 20
INC MPI_		<pre>ile fh, MPI_Offset offset, _Count count, MPI_Datatype datatype,</pre>	20 21
	MPI_Status *status)		22
Fortron	2008 hinding		23
	2008 binding write at all(fh. offset.	buf, count, datatype, status, ierror)	24
	(MPI_File), INTENT(IN) ::		25
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset			26 27
	(*), DIMENSION(), INTEN	T(IN) :: buf	28
	GER, INTENT(IN) :: count		29
	(MPI_Datatype), INTENT(IN	) :: datatype	30
TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror			31
			32
	_write_at_all(in, offset, (MPI_File), INTENT(IN) ::	buf, count, datatype, status, ierror)	33 34
	GER(KIND=MPI_OFFSET_KIND)		35
	(*), DIMENSION(), INTEN	-	36
INTE	GER(KIND=MPI_COUNT_KIND),	INTENT(IN) :: count	37
	(MPI_Datatype), INTENT(IN	) :: datatype	38
	(MPI_Status) :: status		39 40
TNIF	GER, OPTIONAL, INTENT(OUT	) :: lerror	40
Fortran	0		42
		BUF, COUNT, DATATYPE, STATUS, IERROR)	43
	GER FH, COUNT, DATATYPE, GER(KIND=MPI_OFFSET_KIND)	STATUS(MPI_STATUS_SIZE), IERROR	44
	e> BUF(*)	011011	45
• -			46 47
	FILE_WRITE_AT_ALL is a co _WRITE_AT interface.	ollective version of the blocking	47
WF1_HLL_WKHLL_AT Interface.			

```
1
 MPI_FILE_IREAD_AT(fh, offset, buf, count, datatype, request)
2
 IN
 fh
 file handle (handle)
3
 offset
 IN
 file offset (integer)
4
5
 OUT
 buf
 initial address of buffer (choice)
6
 IN
 number of elements in buffer (integer)
 count
7
 IN
 datatype
 datatype of each buffer element (handle)
8
9
 OUT
 request
 request object (handle)
10
11
 C binding
12
 int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
13
 MPI_Datatype datatype, MPI_Request *request)
14
 int MPI_File_iread_at_c(MPI_File fh, MPI_Offset offset, void *buf,
15
 MPI_Count count, MPI_Datatype datatype, MPI_Request *request)
16
17
 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
 MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror)
27
 TYPE(MPI_File), INTENT(IN) :: fh
28
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
29
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
30
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
31
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
32
 TYPE(MPI_Request), INTENT(OUT) :: request
33
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
 Fortran binding
36
 MPI_FILE_IREAD_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_IREAD_AT is a nonblocking version of the MPI_FILE_READ_AT interface.
41
42
43
44
45
46
47
48
```

MPI_FILE_IREAD_AT_ALL(fh, offset, buf, count, datatype, request) ¹				
IN	fh	file handle (handle)	2 3	
IN	offset	file offset (integer)	4	
OUT	buf	initial address of buffer (choice)	5	
IN	count	number of elements in buffer (integer)	6 7	
IN	datatype	datatype of each buffer element (handle)	8	
OUT	request	request object (handle)	9	
			10 11	
C binding			11	
int MPI_F		e fh, MPI_Offset offset, void *buf, ype datatype, MPI_Request *request)	13	
int MDT I		le fh, MPI_Offset offset, void *buf,	14	
IIIC MFI_f		_Datatype datatype, MPI_Request *request)	15 16	
Fortran 2	2008 binding		17	
	•	buf, count, datatype, request, ierror)	18	
	<pre>MPI_File), INTENT(IN) ::</pre>		19 20	
	<pre>SER(KIND=MPI_OFFSET_KIND), (*), DIMENSION(), ASYNCH</pre>		21	
	ER, INTENT(IN) :: count		22	
TYPE	MPI_Datatype), INTENT(IN)		23 24	
	TYPE(MPI_Request), INTENT(OUT) :: request			
INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
	_iread_at_all(fh, offset, [MPI_File), INTENT(IN) ::	<pre>buf, count, datatype, request, ierror) fb</pre>	27	
	ER(KIND=MPI_OFFSET_KIND),		28 29	
TYPE	(*), DIMENSION(), ASYNCH	IRONOUS :: buf	30	
	ER(KIND=MPI_COUNT_KIND),		31	
	<pre>MPI_Datatype), INTENT(IN) MPI_Request), INTENT(OUT)</pre>		32 33	
	ER, OPTIONAL, INTENT(OUT)	-	34	
Fortran k	oinding		35	
	J	BUF, COUNT, DATATYPE, REQUEST, IERROR)	36	
	ER FH, COUNT, DATATYPE, F		37 38	
	<pre>SER(KIND=MPI_OFFSET_KIND) &gt;&gt; BUF(*)</pre>	OFFSET	39	
01			40	
	FILE_IREAD_AT_ALL is a non .6.5 for semantics of nonblocki	blocking version of MPI_FILE_READ_AT_ALL. See	41 42	
Section 14	.0.5 for semantics of holiblock	ing conective me operations.	43	
			44	
			45	
			$46 \\ 47$	
			-	

```
1
 MPI_FILE_IWRITE_AT(fh, offset, buf, count, datatype, request)
2
 INOUT
 fh
 file handle (handle)
3
 offset
 IN
 file offset (integer)
4
5
 buf
 initial address of buffer (choice)
 IN
6
 IN
 count
 number of elements in buffer (integer)
7
 IN
 datatype
 datatype of each buffer element (handle)
8
9
 OUT
 request
 request object (handle)
10
11
 C binding
12
 int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, const void *buf,
13
 int count, MPI_Datatype datatype, MPI_Request *request)
14
 int MPI_File_iwrite_at_c(MPI_File fh, MPI_Offset offset, const void *buf,
15
 MPI_Count count, MPI_Datatype datatype, MPI_Request *request)
16
17
 Fortran 2008 binding
18
 MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror)
19
 TYPE(MPI_File), INTENT(IN) :: fh
20
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
21
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
22
 INTEGER, INTENT(IN) :: count
23
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
24
 TYPE(MPI_Request), INTENT(OUT) :: request
25
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
 MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror)
27
 TYPE(MPI_File), INTENT(IN) :: fh
28
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
29
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
30
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
31
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
32
 TYPE(MPI_Request), INTENT(OUT) :: request
33
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
 Fortran binding
36
 MPI_FILE_IWRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
37
 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
38
 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
39
 <type> BUF(*)
40
 MPI_FILE_IWRITE_AT is a nonblocking version of the MPI_FILE_WRITE_AT interface.
41
42
43
44
45
46
47
48
```

MPI_FILE_IWRITE_AT_ALL(fh, offset, buf, count, datatype, request) ¹			1
INOUT	fh	file handle (handle)	2 3
IN	offset	file offset (integer)	4
IN	buf	initial address of buffer (choice)	5
IN	count	number of elements in buffer (integer)	6
IN	datatype	datatype of each buffer element (handle)	7 8
OUT	request	request object (handle)	9
			10
C binding	רי ס		11
int MPI_F		le fh, MPI_Offset offset, const void *buf,	12 13
	int count, MPI_Datat	ype datatype, MPI_Request *request)	14
int MPI_F		File fh, MPI_Offset offset,	15
	const void *buf, MP1 MPI_Request *request	_Count count, MPI_Datatype datatype,	16
		,	17 18
	2008 binding	, buf, count, datatype, request, ierror)	19
	MPI_File), INTENT(IN) ::		20
	ER(KIND=MPI_OFFSET_KIND)		21
		T(IN), ASYNCHRONOUS :: buf	22 23
	ER, INTENT(IN) :: count		24
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) 24			
	<pre>MPI_File), INTENT(IN) ::</pre>		29
	ER(KIND=MPI_OFFSET_KIND)		30
		(IN), ASYNCHRONOUS :: buf	31
	ER(KIND=MPI_COUNT_KIND), MPI_Datatype), INTENT(IN)		32
	MPI_Datatype), INTENT(IN)	• -	33 34
	ER, OPTIONAL, INTENT(OUT)	•	35
Fortran b			36
		, BUF, COUNT, DATATYPE, REQUEST, IERROR)	37
	ER FH, COUNT, DATATYPE, H		38
INTEG	ER(KIND=MPI_OFFSET_KIND)	OFFSET	39 40
<type< td=""><td>&gt; BUF(*)</td><td></td><td>40</td></type<>	> BUF(*)		40
MPI_F	FILE_IWRITE_AT_ALL is a no	nblocking version of MPI_FILE_WRITE_AT_ALL.	42
			43
14.4.3 D	ata Access with Individual Fi	le Pointers	44
MPI maint	ains one individual file point	er per process per file handle. The current value	45 46
	-	ffset in the data access routines described in this	40 47
			48

```
1
 section. These routines only use and update the individual file pointers maintained by MPI.
\mathbf{2}
 The shared file pointer is not used nor updated.
3
 The individual file pointer routines have the same semantics as the data access with
4
 explicit offset routines described in Section 14.4.2, with the following modification:
5
 • the offset is defined to be the current value of the MPI-maintained individual file
6
 pointer.
7
8
 After an individual file pointer operation is initiated, the individual file pointer is updated
9
 to point to the next etype after the last one that will be accessed. The file pointer is updated
10
 relative to the current view of the file.
11
 If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous
12
 to call the routines in this section, with the exception of MPI_FILE_GET_BYTE_OFFSET.
13
14
15
 MPI_FILE_READ(fh, buf, count, datatype, status)
16
 INOUT
 fh
 file handle (handle)
17
 OUT
 buf
 initial address of buffer (choice)
18
19
 number of elements in buffer (integer)
 IN
 count
20
 datatype of each buffer element (handle)
 IN
 datatype
21
 OUT
 status
 status object (status)
22
23
^{24}
 C binding
 int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
25
26
 MPI_Status *status)
27
 int MPI_File_read_c(MPI_File fh, void *buf, MPI_Count count,
28
 MPI_Datatype datatype, MPI_Status *status)
29
30
 Fortran 2008 binding
 MPI_File_read(fh, buf, count, datatype, status, ierror)
31
 TYPE(MPI_File), INTENT(IN) :: fh
32
33
 TYPE(*), DIMENSION(..) :: buf
34
 INTEGER, INTENT(IN) :: count
35
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
36
 TYPE(MPI_Status) :: status
37
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
 MPI_File_read(fh, buf, count, datatype, status, ierror)
39
 TYPE(MPI_File), INTENT(IN) :: fh
40
 TYPE(*), DIMENSION(..) :: buf
41
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
42
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
43
 TYPE(MPI_Status) :: status
44
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
 Fortran binding
47
 MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
48
 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

```
1
 <type> BUF(*)
 \mathbf{2}
 MPI_FILE_READ reads a file using the individual file pointer.
 3
 4
Example 14.2 The following Fortran code fragment is an example of reading a file until
 5
the end of file is reached:
 6
 7
!
 Read a preexisting input file until all data has been read.
 8
 Call routine "process_input" if all requested data is read.
!
 9
Т
 The Fortran 90 "exit" statement exits the loop.
 10
 11
 bufsize, numread, totprocessed, status(MPI_STATUS_SIZE)
integer
 12
parameter (bufsize=100)
 13
 localbuffer(bufsize)
real
 14
integer (kind=MPI_OFFSET_KIND) zero
 15
 16
zero = 0
 17
 18
call MPI_FILE_OPEN(MPI_COMM_WORLD, 'myoldfile', &
 19
 MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr)
 20
call MPI_FILE_SET_VIEW(myfh, zero, MPI_REAL, MPI_REAL, 'native', &
 21
 MPI_INFO_NULL, ierr)
 22
totprocessed = 0
 23
do
 ^{24}
 call MPI_FILE_READ(myfh, localbuffer, bufsize, MPI_REAL, &
 25
 status, ierr)
 26
 call MPI_GET_COUNT(status, MPI_REAL, numread, ierr)
 27
 call process_input(localbuffer, numread)
 28
 totprocessed = totprocessed + numread
 29
 if (numread < bufsize) exit
 30
end do
 31
 32
write(6, 1001) numread, bufsize, totprocessed
 33
1001 format("No more data: read", I3, "and expected", I3, &
 34
 "Processed total of", I6, "before terminating job.")
 35
 36
call MPI_FILE_CLOSE(myfh, ierr)
 37
 38
 39
 40
 41
 42
 43
 44
 45
 46
 47
```

```
1
 MPI_FILE_READ_ALL(fh, buf, count, datatype, status)
2
 INOUT
 fh
 file handle (handle)
3
 OUT
 buf
 initial address of buffer (choice)
4
5
 IN
 count
 number of elements in buffer (integer)
6
 IN
 datatype
 datatype of each buffer element (handle)
7
 OUT
 status
 status object (status)
8
9
10
 C binding
 int MPI_File_read_all(MPI_File fh, void *buf, int count,
11
 MPI_Datatype datatype, MPI_Status *status)
12
13
 int MPI_File_read_all_c(MPI_File fh, void *buf, MPI_Count count,
14
 MPI_Datatype datatype, MPI_Status *status)
15
16
 Fortran 2008 binding
17
 MPI_File_read_all(fh, buf, count, datatype, status, ierror)
18
 TYPE(MPI_File), INTENT(IN) :: fh
19
 TYPE(*), DIMENSION(..) :: buf
 INTEGER, INTENT(IN) :: count
20
21
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
22
 TYPE(MPI_Status) :: status
23
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{24}
 MPI_File_read_all(fh, buf, count, datatype, status, ierror)
25
 TYPE(MPI_File), INTENT(IN) :: fh
26
 TYPE(*), DIMENSION(..) :: buf
27
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
28
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
 TYPE(MPI_Status) :: status
30
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
32
 Fortran binding
33
 MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
34
 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
35
 <type> BUF(*)
36
 MPI_FILE_READ_ALL is a collective version of the blocking MPI_FILE_READ interface.
37
38
39
40
41
42
43
44
45
46
47
48
```

```
MPI_FILE_WRITE(fh, buf, count, datatype, status)
 1
 \mathbf{2}
 INOUT
 fh
 file handle (handle)
 IN
 buf
 initial address of buffer (choice)
 IN
 count
 number of elements in buffer (integer)
 5
 6
 IN
 datatype of each buffer element (handle)
 datatype
 OUT
 status
 status object (status)
C binding
 10
int MPI_File_write(MPI_File fh, const void *buf, int count,
 11
 MPI_Datatype datatype, MPI_Status *status)
 12
 13
int MPI_File_write_c(MPI_File fh, const void *buf, MPI_Count count,
 14
 MPI_Datatype datatype, MPI_Status *status)
 15
 16
Fortran 2008 binding
MPI_File_write(fh, buf, count, datatype, status, ierror)
 17
 TYPE(MPI_File), INTENT(IN) :: fh
 18
 TYPE(*), DIMENSION(..), INTENT(IN) :: buf
 19
 INTEGER, INTENT(IN) :: count
 20
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 21
 TYPE(MPI_Status) :: status
 22
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 23
 ^{24}
MPI_File_write(fh, buf, count, datatype, status, ierror)
 25
 TYPE(MPI_File), INTENT(IN) :: fh
 26
 TYPE(*), DIMENSION(..), INTENT(IN) :: buf
 27
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
 28
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 29
 TYPE(MPI_Status) :: status
 30
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 31
Fortran binding
 32
 33
MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
 34
 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
 <type> BUF(*)
 35
 36
 MPI_FILE_WRITE writes a file using the individual file pointer.
 37
 38
 39
 40
 41
 42
 43
 44
 45
 46
 47
```

```
1
 MPI_FILE_WRITE_ALL(fh, buf, count, datatype, status)
2
 INOUT
 fh
 file handle (handle)
3
 IN
 buf
 initial address of buffer (choice)
4
5
 number of elements in buffer (integer)
 IN
 count
6
 IN
 datatype
 datatype of each buffer element (handle)
7
 OUT
 status
 status object (status)
8
9
10
 C binding
11
 int MPI_File_write_all(MPI_File fh, const void *buf, int count,
 MPI_Datatype datatype, MPI_Status *status)
12
13
 int MPI_File_write_all_c(MPI_File fh, const void *buf, MPI_Count count,
14
 MPI_Datatype datatype, MPI_Status *status)
15
16
 Fortran 2008 binding
17
 MPI_File_write_all(fh, buf, count, datatype, status, ierror)
18
 TYPE(MPI_File), INTENT(IN) :: fh
19
 TYPE(*), DIMENSION(..), INTENT(IN) :: buf
 INTEGER, INTENT(IN) :: count
20
21
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
22
 TYPE(MPI_Status) :: status
23
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{24}
 MPI_File_write_all(fh, buf, count, datatype, status, ierror)
25
 TYPE(MPI_File), INTENT(IN) :: fh
26
 TYPE(*), DIMENSION(..), INTENT(IN) :: buf
27
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
28
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
 TYPE(MPI_Status) :: status
30
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
32
 Fortran binding
33
 MPI_FILE_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
34
 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
35
 <type> BUF(*)
36
 MPI_FILE_WRITE_ALL is a collective version of the blocking MPI_FILE_WRITE inter-
37
 face.
38
39
40
41
42
43
44
45
46
47
```

1  $\mathbf{2}$ 

3

4 5

6

9

10

11

1213

14

1516

17

18

19

20

22

 24 

25

26

27

28

30

31

32

38

39

```
MPI_FILE_IREAD(fh, buf, count, datatype, request)
 INOUT
 fh
 file handle (handle)
 OUT
 buf
 initial address of buffer (choice)
 IN
 count
 number of elements in buffer (integer)
 IN
 datatype of each buffer element (handle)
 datatype
 OUT
 request
 request object (handle)
C binding
int MPI_File_iread(MPI_File fh, void *buf, int count,
 MPI_Datatype datatype, MPI_Request *request)
int MPI_File_iread_c(MPI_File fh, void *buf, MPI_Count count,
 MPI_Datatype datatype, MPI_Request *request)
Fortran 2008 binding
MPI_File_iread(fh, buf, count, datatype, request, ierror)
 TYPE(MPI_File), INTENT(IN) :: fh
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
 INTEGER, INTENT(IN) :: count
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 21
 TYPE(MPI_Request), INTENT(OUT) :: request
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 23
MPI_File_iread(fh, buf, count, datatype, request, ierror)
 TYPE(MPI_File), INTENT(IN) :: fh
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 29
 TYPE(MPI_Request), INTENT(OUT) :: request
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
 33
MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
 34
 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
 <type> BUF(*)
 35
 36
 MPI_FILE_IREAD is a nonblocking version of the MPI_FILE_READ interface.
 37
Example 14.3 The following Fortran code fragment illustrates file pointer update seman-
tics:
```

```
1
 !
 Read the first twenty real words in a file into two local
\mathbf{2}
 buffers. Note that when the first MPI_FILE_IREAD returns,
 !
3
 the file pointer has been updated to point to the
 !
4
 eleventh real word in the file.
 1
5
6
 bufsize, req1, req2
 integer
\overline{7}
 integer, dimension(MPI_STATUS_SIZE) :: status1, status2
8
 parameter (bufsize=10)
9
 real
 buf1(bufsize), buf2(bufsize)
10
 integer (kind=MPI_OFFSET_KIND) zero
11
12
 zero = 0
13
 call MPI_FILE_OPEN(MPI_COMM_WORLD, 'myoldfile', &
14
 MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr)
15
 call MPI_FILE_SET_VIEW(myfh, zero, MPI_REAL, MPI_REAL, 'native',
 &
16
 MPI_INFO_NULL, ierr)
17
 call MPI_FILE_IREAD(myfh, buf1, bufsize, MPI_REAL, &
18
 req1, ierr)
19
 call MPI_FILE_IREAD(myfh, buf2, bufsize, MPI_REAL, &
20
 req2, ierr)
21
22
 call MPI_WAIT(req1, status1, ierr)
23
 call MPI_WAIT(req2, status2, ierr)
^{24}
25
 call MPI_FILE_CLOSE(myfh, ierr)
26
27
28
 MPI_FILE_IREAD_ALL(fh, buf, count, datatype, request)
29
30
 INOUT
 fh
 file handle (handle)
31
 OUT
 buf
 initial address of buffer (choice)
32
 IN
 number of elements in buffer (integer)
 count
33
34
 IN
 datatype
 datatype of each buffer element (handle)
35
 OUT
 request
 request object (handle)
36
37
 C binding
38
 int MPI_File_iread_all(MPI_File fh, void *buf, int count,
39
 MPI_Datatype datatype, MPI_Request *request)
40
41
 int MPI_File_iread_all_c(MPI_File fh, void *buf, MPI_Count count,
42
 MPI_Datatype datatype, MPI_Request *request)
43
 Fortran 2008 binding
44
 MPI_File_iread_all(fh, buf, count, datatype, request, ierror)
45
 TYPE(MPI_File), INTENT(IN) :: fh
46
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
47
 INTEGER, INTENT(IN) :: count
48
```

	<pre>MPI_Datatype), INTENT(IN)</pre>		1	
	MPI_Request), INTENT(OUT)	-	2	
INTEG	ER, OPTIONAL, INTENT(OUT)	:: lerror	3 4	
MPI_File_	iread_all(fh, buf, count,	, datatype, request, ierror)	5	
TYPE(	<pre>MPI_File), INTENT(IN) ::</pre>	fh	6	
	TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf			
	ER(KIND=MPI_COUNT_KIND),		8	
	<pre>MPI_Datatype), INTENT(IN)</pre>		9	
	MPI_Request), INTENT(OUT)	-	10	
INTEG	ER, OPTIONAL, INTENT(OUT)	) :: lerror	11	
Fortran b	oinding		12	
MPI_FILE_	IREAD_ALL(FH, BUF, COUNT	, DATATYPE, REQUEST, IERROR)	13	
	ER FH, COUNT, DATATYPE, H	REQUEST, IERROR	14	
<type< td=""><td>&gt; BUF(*)</td><td></td><td>15 16</td></type<>	> BUF(*)		15 16	
MPI_F	ILE_IREAD_ALL is a nonbloc	king version of MPI_FILE_READ_ALL.	10	
			18	
			19	
	IWRITE(fh, buf, count, dataty	,	20	
INOUT	fh	file handle (handle)	21	
IN	buf	initial address of buffer (choice)	22	
IN	count	number of elements in buffer (integer)	23	
IN	datatype	datatype of each buffer element (handle)	24	
			25 26	
OUT	request	request object (handle)	20	
Chindin			28	
C binding		const void *buf, int count,	29	
IIIC FILL_F		e, MPI_Request *request)	30	
			31	
int MPI_F		, const void *buf, MPI_Count count,	32	
	MPI_Datatype datatyp	e, MPI_Request *request)	33	
Fortran 2	008 binding		34	
MPI_File_	iwrite(fh, buf, count, da	atatype, request, ierror)	35	
TYPE(	<pre>MPI_File), INTENT(IN) ::</pre>	fh	36 37	
		I(IN), ASYNCHRONOUS :: buf	38	
	ER, INTENT(IN) :: count		39	
	MPI_Datatype), INTENT(IN)	• •	40	
	<pre>MPI_Request), INTENT(OUT) ER, OPTIONAL, INTENT(OUT)</pre>	-	41	
	ER, OFIIONAL, INIENI(UUI)		42	
MPI_File_	iwrite(fh, buf, count, da	atatype, request, ierror)	43	
	<pre>MPI_File), INTENT(IN) ::</pre>		44	
		(IN), ASYNCHRONOUS :: buf	45	
	ER(KIND=MPI_COUNT_KIND),		46	
	<pre>MPI_Datatype), INTENT(IN) MPI_Request), INTENT(OUT)</pre>	• •	47	
IIPE(	mri_nequest), INIENI(UUI)	request	48	

```
1
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
 Fortran binding
3
 MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
4
 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
5
 <type> BUF(*)
6
7
 MPI_FILE_IWRITE is a nonblocking version of the MPI_FILE_WRITE interface.
8
9
 MPI_FILE_IWRITE_ALL(fh, buf, count, datatype, request)
10
11
 INOUT
 file handle (handle)
 fh
12
 IN
 initial address of buffer (choice)
 buf
13
 IN
 count
 number of elements in buffer (integer)
14
15
 datatype of each buffer element (handle)
 IN
 datatype
16
 OUT
 request
 request object (handle)
17
18
 C binding
19
 int MPI_File_iwrite_all(MPI_File fh, const void *buf, int count,
20
 MPI_Datatype datatype, MPI_Request *request)
21
22
 int MPI_File_iwrite_all_c(MPI_File fh, const void *buf, MPI_Count count,
23
 MPI_Datatype datatype, MPI_Request *request)
24
 Fortran 2008 binding
25
 MPI_File_iwrite_all(fh, buf, count, datatype, request, ierror)
26
 TYPE(MPI_File), INTENT(IN) :: fh
27
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
28
 INTEGER, INTENT(IN) :: count
29
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
30
 TYPE(MPI_Request), INTENT(OUT) :: request
31
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33
 MPI_File_iwrite_all(fh, buf, count, datatype, request, ierror)
34
 TYPE(MPI_File), INTENT(IN) :: fh
35
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
36
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
37
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
38
 TYPE(MPI_Request), INTENT(OUT) :: request
39
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
 Fortran binding
41
 MPI_FILE_IWRITE_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
42
 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
43
 <type> BUF(*)
44
45
 MPI_FILE_IWRITE_ALL is a nonblocking version of MPI_FILE_WRITE_ALL.
46
47
48
```

MPI_FILE	_SEEK(fh, offset, whence)		1
INOUT	fh	file handle (handle)	2
IN	offset	file offset (integer)	3 4
IN	whence	update mode (state)	5
	Whenee		6
C bindin	g		7
int MPI_	File_seek(MPI_File fh, M	PI_Offset offset, int whence)	8 9
Fortran 2	2008 binding		9 10
MPI_File	_seek(fh, offset, whence		11
	(MPI_File), INTENT(IN) :		12
	GER(KIND=MPI_OFFSET_KIND		13
	GER, INTENT(IN) :: whenc GER, OPTIONAL, INTENT(OU		14 15
			16
Fortran MPT FILF	binding _SEEK(FH, OFFSET, WHENCE	TERROR)	17
	GER FH, WHENCE, IERROR	, illition)	18
	GER(KIND=MPI_OFFSET_KIND	) OFFSET	19
MPI	FILE SEEK updates the indiv	vidual file pointer according to whence, which has the	20 21
	possible values:		22
• MPI	_SEEK_SET: the pointer is set	t to offset	23 24
• MPI	_SEEK_CUR: the pointer is se	t to the current pointer position plus offset	25
• MPI	_SEEK_END: the pointer is se	t to the end of file plus offset	26 27
The <b>c</b>	offset can be negative, which	allows seeking backwards. It is erroneous to seek to	28
	e position in the view.		29 30
			31
MPI_FILE	_GET_POSITION(fh, offset)		32
IN	fh	file handle (handle)	33
OUT	offset	offset of individual pointer (integer)	34 35
001	Unset	onset of individual pointer (integer)	36
C bindin	g		37
	-	le fh, MPI_Offset *offset)	38
Fortran 2	2008 binding		39
	_get_position(fh, offset	, ierror)	40 41
	(MPI_File), INTENT(IN) :		42
		), INTENT(OUT) :: offset	43
TN1E(	GER, OPTIONAL, INTENT(OU	1) :: lerror	44
Fortran			45
	_GET_POSITION(FH, OFFSET	, IERROR)	46 47
	GER FH, IERROR GER(KIND=MPI_OFFSET_KIND	) OFFSET	48
	QPIC(UIND-III I_OLIOEI_UIND	/ 011001	

1 2	MPI_FILE_GET_POSITION returns, in offset, the current position of the individual file pointer in etype units relative to the current view.		
3 4 5 6 7 8	<i>Advice to users.</i> 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. ( <i>End of advice to users.</i> )		
9 10			
11 12	MPI_FILE_GET_BYTE_OF	FSET(fh, offset, disp)	
13	IN fh	file handle (handle)	
14	IN offset	offset (integer)	
15 16	OUT disp	absolute byte position of offset (integer)	
17 18 19 20	C binding int MPI_File_get_byte_c MPI_Offse	offset(MPI_File fh, MPI_Offset offset, t *disp)	
21 22 23 24 25 26 27	<pre>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</pre>		
27 28 29 30 31	Fortran binding MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP		
32 33 34 35	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.		
36 37	14.4.4 Data Access with	Shared File Pointers	
38 39 40 41 42 43 44	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:		
45		b be the current value of the MPI-maintained shared file pointer,	
46 47 48		calls to shared file pointer routines is defined to behave as if the	

ste serranzeu, and

• the use of shared file pointer routines is erroneous unless all processes use the same file view.

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

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

Noncollective Operations

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

```
MPI_Datatype datatype, MPI_Status *status)
```

## Fortran 2008 binding

<b>u</b>
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)
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_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
<type> BUF(*)

 $\mathbf{2}$ 

 $45 \\ 46$ 

```
1
 MPI_FILE_READ_SHARED reads a file using the shared file pointer.
\mathbf{2}
3
 MPI_FILE_WRITE_SHARED(fh, buf, count, datatype, status)
4
5
 INOUT
 fh
 file handle (handle)
6
 IN
 buf
 initial address of buffer (choice)
7
 IN
 count
 number of elements in buffer (integer)
8
9
 IN
 datatype of each buffer element (handle)
 datatype
10
 OUT
 status
 status object (status)
11
12
 C binding
13
 int MPI_File_write_shared(MPI_File fh, const void *buf, int count,
14
 MPI_Datatype datatype, MPI_Status *status)
15
16
 int MPI_File_write_shared_c(MPI_File fh, const void *buf, MPI_Count count,
17
 MPI_Datatype datatype, MPI_Status *status)
18
 Fortran 2008 binding
19
 MPI_File_write_shared(fh, buf, count, datatype, status, ierror)
20
 TYPE(MPI_File), INTENT(IN) :: fh
21
 TYPE(*), DIMENSION(...), INTENT(IN) :: buf
22
 INTEGER, INTENT(IN) :: count
23
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
24
 TYPE(MPI_Status) :: status
25
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27
 MPI_File_write_shared(fh, buf, count, datatype, status, ierror)
28
 TYPE(MPI_File), INTENT(IN) :: fh
29
 TYPE(*), DIMENSION(...), INTENT(IN) :: buf
30
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
31
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
32
 TYPE(MPI_Status) :: status
33
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
 Fortran binding
35
 MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
36
 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
37
 <type> BUF(*)
38
39
 MPI_FILE_WRITE_SHARED writes a file using the shared file pointer.
40
41
42
43
44
45
46
47
48
```

MPI_FILE_IREAD_SHARED(fh, buf, count, datatype, request) ¹			
INOUT	fh	file handle (handle)	2
OUT	buf	initial address of buffer (choice)	3 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
001	request	request object (nandic)	8 9
C bindin	g		10
	•	e fh, void *buf, int count,	11
	MPI_Datatype datatyp	pe, MPI_Request *request)	12
int MPI_	File_iread_shared_c(MPI_F	lile fh, void *buf, MPI_Count count,	13
	MPI_Datatype datatyp	pe, MPI_Request *request)	14 15
Fortran 2	2008 binding		16
	0	unt, datatype, request, ierror)	17
	(MPI_File), INTENT(IN) ::		18
	(*), DIMENSION(), ASYNC	HRONOUS :: buf	19
	GER, INTENT(IN) :: count (MPI_Datatype), INTENT(IN		20 21
	(MPI_Request), INTENT(OUT		22
	GER, OPTIONAL, INTENT(OUT		23
MPT Filo	iread shared (fh buf co	unt, datatype, request, ierror)	24
	(MPI_File), INTENT(IN) ::		25
	(*), DIMENSION(), ASYNC		26 27
	GER(KIND=MPI_COUNT_KIND),		28
	(MPI_Datatype), INTENT(IN		29
	(MPI_Request), INTENT(OUT GER, OPTIONAL, INTENT(OUT		30
		/ 161101	31
Fortran	J		32 33
	GER FH, COUNT, DATATYPE,	UNT, DATATYPE, REQUEST, IERROR) REQUEST IERROR	34
	<pre>BUF(*)</pre>		35
01		nblocking version of the MPI_FILE_READ_SHARED	36
interface.	TILL_INLAD_STANLD IS a 110	holocking version of the MFT_TEL_NEAD_STARED	37
11100110000			38 39
			40
			41
			42
			43
			44 45
			46
			47
			48

```
1
 MPI_FILE_IWRITE_SHARED(fh, buf, count, datatype, request)
2
 INOUT
 fh
 file handle (handle)
3
 IN
 buf
 initial address of buffer (choice)
4
5
 IN
 number of elements in buffer (integer)
 count
6
 IN
 datatype of each buffer element (handle)
 datatype
7
 OUT
 request
 request object (handle)
8
9
10
 C binding
 int MPI_File_iwrite_shared(MPI_File fh, const void *buf, int count,
11
 MPI_Datatype datatype, MPI_Request *request)
12
13
 int MPI_File_iwrite_shared_c(MPI_File fh, const void *buf, MPI_Count count,
14
 MPI_Datatype datatype, MPI_Request *request)
15
16
 Fortran 2008 binding
17
 MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror)
18
 TYPE(MPI_File), INTENT(IN) :: fh
19
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
 INTEGER, INTENT(IN) :: count
20
21
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 TYPE(MPI_Request), INTENT(OUT) :: request
22
23
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{24}
 MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror)
25
 TYPE(MPI_File), INTENT(IN) :: fh
26
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
27
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
28
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
 TYPE(MPI_Request), INTENT(OUT) :: request
30
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
32
 Fortran binding
 MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
33
34
 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
35
 <type> BUF(*)
36
 MPI_FILE_IWRITE_SHARED is a nonblocking version of the
37
 MPI_FILE_WRITE_SHARED interface.
38
39
 Collective Operations
40
41
 The semantics of a collective access using a shared file pointer is that the accesses to the
42
 file will be in the order determined by the ranks of the processes within the group. For each
43
 process, the location in the file at which data is accessed is the position at which the shared
44
 file pointer would be after all processes whose ranks within the group less than that of this
45
 process had accessed their data. In addition, in order to prevent subsequent shared offset
46
 accesses by the same processes from interfering with this collective access, the call might
47
 return only after all the processes within the group have initiated their accesses. When the
48
```

call returns, the shared file pointer points to the next etype accessible, according to the file view used by all processes, after the last etype requested.

Advice to users. There may be some programs in which all processes in the group need to access the file using the shared file pointer, but the program may not *require* that data be accessed in order of process rank. In such programs, using the shared ordered routines (e.g., MPI_FILE_WRITE_ORDERED rather than MPI_FILE_WRITE_SHARED) may enable an implementation to optimize access, improving performance. (*End of advice to users.*)

Advice to implementors. Accesses to the data requested by all processes do not have to be serialized. Once all processes have issued their requests, locations within the file for all accesses can be computed, and accesses can proceed independently from each other, possibly in parallel. (*End of advice to implementors.*)

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

MPI FILE	.READ_ORDERED(fh, buf, cou	unt. datatype. status)	
INOUT	× ×		18
	fh	file handle (handle)	19
OUT	buf	initial address of buffer (choice)	20
IN	count	number of elements in buffer (integer)	21
IN	datatype	datatype of each buffer element (handle)	22
	51		23 24
OUT	status	status object (status)	24 25
~			26
C binding			27
int MPI_F		e fh, void *buf, int count,	28
	MPI_Datatype datatype	e, MPI_Status *status)	29
int MPI_F	ile_read_ordered_c(MPI_Fi	ile fh, void *buf, MPI_Count count,	30
	MPI_Datatype datatype	e, MPI_Status *status)	31
Fontnon 2	008 binding		32
	<b>y</b>	mt, datatype, status, ierror)	33
	MPI_File), INTENT(IN) ::		34
	*), DIMENSION() :: buf	111	35
	ER, INTENT(IN) :: count		36
	MPI_Datatype), INTENT(IN)	:: datatype	37
	MPI_Status) :: status		38
	ER, OPTIONAL, INTENT(OUT)	:: ierror	39
			40
		nt, datatype, status, ierror)	41
	<pre>MPI_File), INTENT(IN) ::</pre>	Íh	42
	*), DIMENSION() :: buf		43
	ER(KIND=MPI_COUNT_KIND),		44
	MPI_Datatype), INTENT(IN) MPI_Status) :: status	:: datatype	45
	ER, OPTIONAL, INTENT(OUT)	···ierror	46 47
TNIEG	ER, OF ITOWAL, INTENT(UOT)	ICIIOI	47 48
			40

1 2

3

4

5

6

7

8

9 10

11

12

13

```
1
 Fortran binding
\mathbf{2}
 MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
3
 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
4
 <type> BUF(*)
5
 MPI_FILE_READ_ORDERED is a collective version of the MPI_FILE_READ_SHARED
6
 interface.
7
8
9
 MPI_FILE_WRITE_ORDERED(fh, buf, count, datatype, status)
10
 INOUT
 fh
 file handle (handle)
11
 IN
 buf
 initial address of buffer (choice).
12
13
 IN
 count
 number of elements in buffer (integer)
14
 datatype of each buffer element (handle)
 IN
 datatype
15
16
 OUT
 status
 status object (status)
17
18
 C binding
19
 int MPI_File_write_ordered(MPI_File fh, const void *buf, int count,
20
 MPI_Datatype datatype, MPI_Status *status)
21
 int MPI_File_write_ordered_c(MPI_File fh, const void *buf, MPI_Count count,
22
 MPI_Datatype datatype, MPI_Status *status)
23
^{24}
 Fortran 2008 binding
25
 MPI_File_write_ordered(fh, buf, count, datatype, status, ierror)
26
 TYPE(MPI_File), INTENT(IN) :: fh
27
 TYPE(*), DIMENSION(..), INTENT(IN) :: buf
28
 INTEGER, INTENT(IN) :: count
29
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
30
 TYPE(MPI_Status) :: status
31
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
 MPI_File_write_ordered(fh, buf, count, datatype, status, ierror)
33
 TYPE(MPI_File), INTENT(IN) :: fh
34
 TYPE(*), DIMENSION(..), INTENT(IN) :: buf
35
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
36
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
37
 TYPE(MPI_Status) :: status
38
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
 Fortran binding
41
 MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
42
 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
43
 <type> BUF(*)
44
 MPI_FILE_WRITE_ORDERED is a collective version of the MPI_FILE_WRITE_SHARED
45
 interface.
46
47
48
```

Seek			1
	If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous		
	e following two routines (Mf	/	3
	_GET_POSITION_SHARED).		4 5
			6
	_SEEK_SHARED(fh, offset, wh	pence)	7
	× ×	,	8
INOUT	fh	file handle (handle)	9
IN	offset	file offset (integer)	10
IN	whence	update mode (state)	11 12
			12
C bindin	•		14
int MPI_H	File_seek_shared(MPI_File	fh, MPI_Offset offset, int whence)	15
Fortran 2	2008 binding		16
	seek_shared(fh, offset,		17
	(MPI_File), INTENT(IN) ::		18
	SER(KIND=MPI_OFFSET_KIND)	, INTENT(IN) :: offset	19 20
	GER, INTENT(IN) :: whence GER, OPTIONAL, INTENT(OUT	) · · · jerror	20 21
		, 161101	22
Fortran h	0		23
	SEEK_SHARED(FH, OFFSET,	WHENCE, IERROR)	24
	INTEGER FH, WHENCE, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET		
			26
	-	the shared file pointer according to whence, which	27 28
has the following possible values:		28 29	
• MPI_SEEK_SET: the pointer is set to offset		30	
• MDI	SEEK CUD: the pointer is get	to the current pointer position plus offset	31
• WP1_	SEEK_COR: the pointer is set	to the current pointer position plus onset	32
• MPI_SEEK_END: the pointer is set to the end of file plus offset		33	
MPI_FILE_SEEK_SHARED is collective; all the processes in the communicator group		34	
associated with the file handle fh must call MPI_FILE_SEEK_SHARED with the same values		35 36	
for offset and whence.		37	
The c	ffset can be negative, which a	llows seeking backwards. It is erroneous to seek to	38
a negative	position in the view.		39
			40
MPI FILE	_GET_POSITION_SHARED(fh	. offset)	41
IN	fh	file handle (handle)	42
			$\frac{43}{44}$
OUT	offset	offset of shared pointer (integer)	44 45
	~		46
C binding int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)		47	
INC IN I_I	TTO-200-hoproron-pugred()	millio in, millibet wollbet/	48

1	
1	Fortran 2008 binding
2	MPI_File_get_position_shared(fh, offset, ierror)
3	TYPE(MPI_File), INTENT(IN) :: fh
4	INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset
5	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6	Fortran binding
7	
8	MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERROR)
9	INTEGER FH, IERROR
10	INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
11	MPI_FILE_GET_POSITION_SHARED returns, in offset, the current position of the
12	shared file pointer in etype units relative to the current view.
13	shared me pointer in coppe and relative to the current view.
14	Advice to users. The offset can be used in a future call to MPI_FILE_SEEK_SHARED
15	using whence = MPI_SEEK_SET to return to the current position. To set the displace-
16	ment to the current file pointer position, first convert offset into an absolute byte
17	position using MPI_FILE_GET_BYTE_OFFSET, then call MPI_FILE_SET_VIEW with
18	the resulting displacement. (End of advice to users.)
19	the resulting displacement. (End of dubice to users.)
20	
	14.4.5 Split Collective Data Access Routines
21	MPI provides a restricted form of "nonblocking collective" I/O operations for all data ac-
22	cesses using split collective data access routines. These routines are referred to as "split"
23	collective routines because a single collective operation is split in two: a begin routine and
24	an end routine. The begin routine begins the operation, much like a nonblocking data access
25	(e.g., MPI_FILE_IREAD). The end routine completes the operation, much like the matching
26	
27	test or wait (e.g., MPI_WAIT). As with nonblocking data access operations, the user must
28	not use the buffer passed to a begin routine while the routine is outstanding; the operation
29	must be completed with an end routine before it is safe to free buffers, etc.
30	Split collective data access operations on a file handle fh are subject to the semantic
31	rules given below.
32	
33	• On any MPI process, each file handle may have at most one active split collective
34	operation at any time.
35	• Begin calls are collective over the group of processes that participated in the collective
36	• Begin cans are conective over the group of processes that participated in the conective open and follow the ordering rules for collective calls.
37	open and follow the ordering fules for conective cans.
38	• End calls are collective over the group of processes that participated in the collective
39	open and follow the ordering rules for collective calls. Each end call matches the
40	preceding begin call for the same collective operation. When an "end" call is made,
40	exactly one unmatched "begin" call for the same operation must precede it.
	chargery one unmatched begin can for the same operation must precede it.
42	• An implementation is free to implement any split collective data access routine using
43	the corresponding blocking collective routine when either the begin call (e.g.,
44	MPI_FILE_READ_ALL_BEGIN) or the end call (e.g., MPI_FILE_READ_ALL_END) is
45	issued. The begin and end calls are provided to allow the user and MPI implementation
46	to optimize the collective operation.
47	
48	

According to the definitions in Section 2.4.2, the begin procedures are incomplete. They are also non-local procedures because they may or may not return before they are called in all MPI processes of the process group.

Advice to users. This is one of the exceptions in which incomplete procedures are non-local and therefore blocking. (End of advice to users.)

- Split collective operations do not match the corresponding regular collective operation. For example, in a single collective read operation, an MPI_FILE_READ_ALL on one process does not match an MPI_FILE_READ_ALL_BEGIN/ MPI_FILE_READ_ALL_END pair on another process.
- Split collective routines must specify a buffer in both the begin and end routines. By specifying the buffer that receives data in the end routine, we can avoid the problems described in "A Problem with Code Movements and Register Optimization," Section 19.1.17, but not all of the problems, such as those described in Sections 19.1.12, 19.1.13, and 19.1.16.
- No collective I/O operations are permitted on a file handle concurrently with a split collective access on that file handle (i.e., between the begin and end of the access). That is

<pre>MPI_File_read_all_begin(fh,);</pre>
<pre> MPI_File_read_all(fh,);</pre>
<pre> MPI_File_read_all_end(fh,);</pre>

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.

```
1
 MPI_FILE_READ_AT_ALL_BEGIN(fh, offset, buf, count, datatype)
\mathbf{2}
 IN
 fh
 file handle (handle)
3
 offset
 IN
 file offset (integer)
4
5
 OUT
 buf
 initial address of buffer (choice)
6
 IN
 number of elements in buffer (integer)
 count
7
 IN
 datatype
 datatype of each buffer element (handle)
8
9
 C binding
10
 int MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,
11
 int count, MPI_Datatype datatype)
12
13
 int MPI_File_read_at_all_begin_c(MPI_File fh, MPI_Offset offset, void *buf,
14
 MPI_Count count, MPI_Datatype datatype)
15
16
 Fortran 2008 binding
17
 MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror)
18
 TYPE(MPI_File), INTENT(IN) :: fh
19
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
20
21
 INTEGER, INTENT(IN) :: count
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
22
23
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
 MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror)
25
 TYPE(MPI_File), INTENT(IN) :: fh
26
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
27
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
28
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
29
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
30
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
32
 Fortran binding
33
 MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
34
 INTEGER FH, COUNT, DATATYPE, IERROR
35
 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
36
 <type> BUF(*)
37
38
39
 MPI_FILE_READ_AT_ALL_END(fh, buf, status)
40
 fh
 IN
 file handle (handle)
41
42
 OUT
 buf
 initial address of buffer (choice)
43
 OUT
 status
 status object (status)
44
45
 C binding
46
 int MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)
47
48
```

<pre>Fortran 2008 binding MPI_File_read_at_all_end(fh, buf, status, ierror) TYPE(MPI_File), INTENT(IN) :: fh TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>					
INTEC	Dinding READ_AT_ALL_END(FH, BUF, ER FH, STATUS(MPI_STATUS) >> BUF(*)		7 8 9 10 11 12		
MPI_FILE	_WRITE_AT_ALL_BEGIN(fh, o	offset, buf, count, datatype)	13 14		
INOUT	fh	file handle (handle)	15		
IN	offset	file offset (integer)	16		
			17		
IN	buf	initial address of buffer (choice)	18 19		
IN	count	number of elements in buffer (integer)	20		
IN	datatype	datatype of each buffer element (handle)	21		
$\alpha$ $1$ $1$ $1$			22		
C binding	•	PI_File fh, MPI_Offset offset,	23		
IIIC MFI_I	0	count, MPI_Datatype datatype)	24		
			25 26		
int MPI_F	J	(MPI_File fh, MPI_Offset offset, _Count count, MPI_Datatype datatype)	27		
		_count count, MF1_Datatype datatype)	28		
	2008 binding		29		
	5	ffset, buf, count, datatype, ierror)	30		
	<pre>(MPI_File), INTENT(IN) :: ER(KIND=MPI_OFFSET_KIND)</pre>		31		
		T(IN), ASYNCHRONOUS :: buf	32		
	GER, INTENT(IN) :: count		33 34		
	(MPI_Datatype), INTENT(IN	) :: datatype	35		
INTEC	ER, OPTIONAL, INTENT(OUT)	) :: ierror	36		
MPT File	write at all begin(fh. o	ffset, buf, count, datatype, ierror)	37		
	(MPI_File), INTENT(IN) ::		38		
	ER(KIND=MPI_OFFSET_KIND)		39		
TYPE (	(*), DIMENSION(), INTEN	T(IN), ASYNCHRONOUS :: buf	40		
	ER(KIND=MPI_COUNT_KIND),		41 42		
	(MPI_Datatype), INTENT(IN	• =	43		
INTEC	ER, OPTIONAL, INTENT(OUT	) :: lerror	44		
Fortran b	0		45		
		FFSET, BUF, COUNT, DATATYPE, IERROR)	46		
	ER FH, COUNT, DATATYPE,		47		
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET 48					

```
1
 <type> BUF(*)
\mathbf{2}
3
4
 MPI_FILE_WRITE_AT_ALL_END(fh, buf, status)
5
 INOUT
 fh
 file handle (handle)
6
 IN
7
 buf
 initial address of buffer (choice)
8
 OUT
 status object (status)
 status
9
10
 C binding
11
 int MPI_File_write_at_all_end(MPI_File fh, const void *buf,
12
 MPI_Status *status)
13
14
 Fortran 2008 binding
 MPI_File_write_at_all_end(fh, buf, status, ierror)
15
16
 TYPE(MPI_File), INTENT(IN) :: fh
17
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
18
 TYPE(MPI_Status) :: status
19
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
 Fortran binding
21
 MPI_FILE_WRITE_AT_ALL_END(FH, BUF, STATUS, IERROR)
22
 INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
23
 <type> BUF(*)
24
25
26
 MPI_FILE_READ_ALL_BEGIN(fh, buf, count, datatype)
27
28
 INOUT
 fh
 file handle (handle)
29
 OUT
 initial address of buffer (choice)
 buf
30
 number of elements in buffer (integer)
 IN
 count
^{31}
32
 IN
 datatype
 datatype of each buffer element (handle)
33
34
 C binding
35
 int MPI_File_read_all_begin(MPI_File fh, void *buf, int count,
36
 MPI_Datatype datatype)
37
 int MPI_File_read_all_begin_c(MPI_File fh, void *buf, MPI_Count count,
38
 MPI_Datatype datatype)
39
40
 Fortran 2008 binding
41
 MPI_File_read_all_begin(fh, buf, count, datatype, ierror)
42
 TYPE(MPI_File), INTENT(IN) :: fh
43
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
44
 INTEGER, INTENT(IN) :: count
45
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
46
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
 MPI_File_read_all_begin(fh, buf, count, datatype, ierror)
48
```

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			1 2 3 4 5	
INTE	binding _READ_ALL_BEGIN(FH, BUF, GER FH, COUNT, DATATYPE, e> BUF(*)		6 7 8 9 10 11 12	
MPI_FILE	_READ_ALL_END(fh, buf, stat	tus)	13	
INOUT	fh	file handle (handle)	14	
OUT	buf	initial address of buffer (choice)	15 16	
OUT	status	status object (status)	17	
			18	
C bindin	0		19	
int MPI_	File_read_all_end(MPI_Fil	e fh, void *buf, MPI_Status *status)	20 21	
Fortran	2008 binding		21	
	_read_all_end(fh, buf, st		23	
TYPE(MPI_File), INTENT(IN) :: fh				
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf TYPE(MPI_Status) :: status			25	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror			26 27	
Fortran binding				
	_READ_ALL_END(FH, BUF, ST	ATUS, TERBOR)	29	
	GER FH, STATUS(MPI_STATUS		30	
<typ< td=""><td>e&gt; BUF(*)</td><td></td><td>31</td></typ<>	e> BUF(*)		31	
			32	
			33 34	
MPI_FILE	_WRITE_ALL_BEGIN(fh, buf,	count, datatype)	35	
INOUT	fh	file handle (handle)	36	
IN	buf	initial address of buffer (choice)	37	
IN	count	number of elements in buffer (integer)	38	
IN	datatype	datatype of each buffer element (handle)	39 40	
	51		41	
C bindin	g		42	
int MPI_	•	File fh, const void *buf, int count,	43	
	MPI_Datatype datatyp	e)	44	
int MPI_	File_write_all_begin_c(MP	I_File fh, const void *buf,	45 46	
	MPI_Count count, MPI_Datatype datatype)			
			48	

```
1
 Fortran 2008 binding
\mathbf{2}
 MPI_File_write_all_begin(fh, buf, count, datatype, ierror)
3
 TYPE(MPI_File), INTENT(IN) :: fh
4
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
5
 INTEGER, INTENT(IN) :: count
6
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
7
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
 MPI_File_write_all_begin(fh, buf, count, datatype, ierror)
9
 TYPE(MPI_File), INTENT(IN) :: fh
10
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
11
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
12
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
13
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
 Fortran binding
16
 MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
17
 INTEGER FH, COUNT, DATATYPE, IERROR
18
 <type> BUF(*)
19
20
21
 MPI_FILE_WRITE_ALL_END(fh, buf, status)
22
 file handle (handle)
 INOUT
 fh
23
^{24}
 IN
 buf
 initial address of buffer (choice)
25
 status object (status)
 OUT
 status
26
27
 C binding
28
 int MPI_File_write_all_end(MPI_File fh, const void *buf,
29
 MPI_Status *status)
30
^{31}
 Fortran 2008 binding
32
 MPI_File_write_all_end(fh, buf, status, ierror)
33
 TYPE(MPI_File), INTENT(IN) :: fh
34
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
35
 TYPE(MPI_Status) :: status
36
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
 Fortran binding
38
 MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)
39
 INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
40
 <type> BUF(*)
41
42
43
44
45
46
47
48
```

MPI_FILE_	_READ_ORDERED_BEGIN(fh,	buf, count, datatype)	1	
INOUT	fh	file handle (handle)	2 3	
OUT	buf	initial address of buffer (choice)	4	
IN	count	number of elements in buffer (integer)	5	
IN	datatype	datatype of each buffer element (handle)	6	
			7	
C binding	<u>,</u>		8 9	
int MPI_F	'ile_read_ordered_begin(MA	PI_File fh, void *buf, int count,	10	
	MPI_Datatype datatyp	e)	11	
int MPI_F	'ile_read_ordered_begin_c	(MPI_File fh, void *buf, MPI_Count count,	12	
	MPI_Datatype datatyp	e)	13	
Fortran 2	2008 binding		14 15	
MPI_File_	read_ordered_begin(fh, bu	if, count, datatype, ierror)	16	
	<pre>MPI_File), INTENT(IN) ::</pre>		17	
	(*), DIMENSION(), ASYNCH	HRONOUS :: buf	18	
	ER, INTENT(IN) :: count [MPI_Datatype), INTENT(IN]	) : datatype	19	
	ER, OPTIONAL, INTENT(OUT)		20 21	
	·		21	
	<pre>[Pead_ordered_begin(In, b) [MPI_File), INTENT(IN) ::</pre>	if, count, datatype, ierror) fh	23	
	(*), DIMENSION(), ASYNCH		24	
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count				
TYPE(MPI_Datatype), INTENT(IN) :: datatype				
INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
Fortran b	oinding		28 29	
		JF, COUNT, DATATYPE, IERROR)	30	
	ER FH, COUNT, DATATYPE, I	IERROR	31	
<type< td=""><td>&gt; BUF(*)</td><td></td><td>32</td></type<>	> BUF(*)		32	
			33 34	
MPL FILE	_READ_ORDERED_END(fh, b	uf. status)	35	
INOUT	fh	file handle (handle)	36	
			37	
OUT	buf	initial address of buffer (choice)	38	
OUT	status	status object (status)	$39 \\ 40$	
			40 41	
C binding		_File fh, void *buf, MPI_Status *status)	42	
		_File III, Vold *bul, MIT_Status *status/	43	
	2008 binding		44	
	read_ordered_end(fh, buf MPI_File), INTENT(IN) ::		45	
	(*), DIMENSION(), ASYNCH		$46 \\ 47$	
	[MPI_Status) :: status		47	

```
688
```

```
1
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
 Fortran binding
3
 MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR)
4
 INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
5
 <type> BUF(*)
6
7
8
 MPI_FILE_WRITE_ORDERED_BEGIN(fh, buf, count, datatype)
9
10
 INOUT
 fh
 file handle (handle)
11
 IN
 buf
 initial address of buffer (choice)
12
 number of elements in buffer (integer)
 IN
 count
13
14
 IN
 datatype of each buffer element (handle)
 datatype
15
16
 C binding
17
 int MPI_File_write_ordered_begin(MPI_File fh, const void *buf, int count,
18
 MPI_Datatype datatype)
19
 int MPI_File_write_ordered_begin_c(MPI_File fh, const void *buf,
20
 MPI_Count count, MPI_Datatype datatype)
21
22
 Fortran 2008 binding
23
 MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror)
24
 TYPE(MPI_File), INTENT(IN) :: fh
25
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
26
 INTEGER, INTENT(IN) :: count
27
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
28
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
 MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror)
30
 TYPE(MPI_File), INTENT(IN) :: fh
31
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
32
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
33
34
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
 Fortran binding
37
 MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
38
 INTEGER FH, COUNT, DATATYPE, IERROR
39
 <type> BUF(*)
40
41
42
43
44
45
46
47
48
```

#### C binding

#### Fortran 2008 binding

```
MPI_File_write_ordered_end(fh, buf, status, ierror)
 TYPE(MPI_File), INTENT(IN) :: fh
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
 TYPE(MPI_Status) :: status
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

#### Fortran binding

```
MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)
 INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
 <type> BUF(*)
```

## 14.5 File Interoperability

At the most basic level, file interoperability is the ability to read the information previously written to a file—not just the bits of data, but the actual information the bits represent. MPI guarantees full interoperability within a single MPI environment, and supports increased interoperability outside that environment through the external data representation (Section 14.5.2) as well as the data conversion functions (Section 14.5.3).

Interoperability within a single MPI environment (which could be considered "operability") ensures that file data written by one MPI process can be read by any other MPI process, subject to the consistency constraints (see Section 14.6.1), provided that it would have been possible to start the two processes simultaneously and have them reside in a single MPI_COMM_WORLD. Furthermore, both processes must see the same data values at every absolute byte offset in the file for which data was written.

This single environment file interoperability implies that file data is accessible regardless of the number of processes.

There are three aspects to file interoperability:

- transferring the bits,
- converting between different file structures, and
- converting between different machine representations.

The first two aspects of file interoperability are beyond the scope of this standard, as both are highly machine dependent. However, transferring the bits of a file into and out of the MPI environment (e.g., by writing a file to tape) is required to be supported by all MPI implementations. In particular, an implementation must specify how familiar

#### Unofficial Draft for Comment Only

2

10

11

12

13

14

15

16 17

18

19

20

21 22 23

24 25

26

27

28

29

30

 31 

32

33

34

35

36

37

38

39

40 41

42 43

1 operations similar to POSIX cp, rm, and mv can be performed on the file. Furthermore, it  $\mathbf{2}$ is expected that the facility provided maintains the correspondence between absolute byte 3 offsets (e.g., after possible file structure conversion, the data bits at byte offset 102 in the 4 MPI environment are at byte offset 102 outside the MPI environment). As an example,  $\mathbf{5}$ a simple off-line conversion utility that transfers and converts files between the native file 6 system and the MPI environment would suffice, provided it maintained the offset coherence 7mentioned above. In a high-quality implementation of MPI, users will be able to manipulate 8 MPI files using the same or similar tools that the native file system offers for manipulating 9 its files. 10 The remaining aspect of file interoperability, converting between different machine 11representations, is supported by the typing information specified in the etype and filetype. 12This facility allows the information in files to be shared between any two applications, 13regardless of whether they use MPI, and regardless of the machine architectures on which

¹⁴ they run.

22

23

 24 

25 26

27

28

29

30

31

32

33 34

35

36

37

38 39

40

41

42

43

44

45

¹⁵ MPI supports multiple data representations: "native", "internal", and "external32". An ¹⁶ implementation may support additional data representations. MPI also supports user-¹⁷ defined data representations (see Section 14.5.3). The "native" and "internal" data repre-¹⁸ sentations are implementation dependent, while the "external32" representation is common ¹⁹ to all MPI implementations and facilitates file interoperability. The data representation is ²⁰ specified in the datarep argument to MPI_FILE_SET_VIEW.

- Advice to users. MPI is not guaranteed to retain knowledge of what data representation was used when a file is written. Therefore, to correctly retrieve file data, an MPI application is responsible for specifying the same data representation as was used to create the file. (*End of advice to users.*)
- "native" Data in this representation is stored in a file exactly as it is in memory. The advantage of this data representation is that data precision and I/O performance are not lost in type conversions with a purely homogeneous environment. The disadvantage is the loss of transparent interoperability within a heterogeneous MPI environment.

Advice to users. This data representation should only be used in a homogeneous MPI environment, or when the MPI application is capable of performing the datatype conversions itself. (*End of advice to users.*)

Advice to implementors. When implementing read and write operations on top of MPI message-passing, the message data should be typed as MPI_BYTE to ensure that the message routines do not perform any type conversions on the data. (*End of advice to implementors.*)

"internal" This data representation can be used for I/O operations in a homogeneous or heterogeneous environment; the implementation will perform type conversions if necessary. The implementation is free to store data in any format of its choice, with the restriction that it will maintain constant extents for all predefined datatypes in any one file. The environment in which the resulting file can be reused is implementationdefined 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.

One can logically think of the file as if it were stored in the 37 Advice to users. memory of a file server. The etype and filetype are interpreted as if they were defined 3839 at this file server, by the same sequence of calls used to define them at the calling process. If the data representation is "native", then this logical file server runs on 40 41 the same architecture as the calling process, so that these types define the same data 42layout on the file as they would define in the memory of the calling process. If the etype and filetype are portable datatypes, then the data layout defined in the file is 43the same as would be defined in the calling process memory, up to a scaling factor. 44The routine MPI_FILE_GET_TYPE_EXTENT can be used to calculate this scaling 4546factor. Thus, two equivalent, portable datatypes will define the same data layout in the file, even in a heterogeneous environment with "internal", "external32", or user 4748 defined data representations. Otherwise, the etype and filetype must be constructed

#### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6 7

8

9 10

11

12

13

14

15

16

17 18

19

20

21

22

23 24

25 26

27

28

29

30

31

32

33

34

35

1 so that their typemap and extent are the same on any architecture. This can be 2 achieved if they have an explicit upper bound and lower bound (defined using 3 MPI_TYPE_CREATE_RESIZED). This condition must also be fulfilled by any datatype 4 that is used in the construction of the etype and filetype, if this datatype is replicated 5contiguously, either explicitly, by a call to MPI_TYPE_CONTIGUOUS, or implicitly, 6 by a blocklength argument that is greater than one. If an etype or filetype is not 7 portable, and has a typemap or extent that is architecture dependent, then the data 8 layout specified by it on a file is implementation dependent. 9 File data representations other than "native" may be different from corresponding 10 data representations in memory. Therefore, for these file data representations, it is 11 important not to use hardwired byte offsets for file positioning, including the initial 12displacement that specifies the view. When a portable datatype (see Section 2.4) is 13 used in a data access operation, any holes in the datatype are scaled to match the data 14representation. However, note that this technique only works when all the processes 15that created the file view build their etypes from the same predefined datatypes. For 16example, if one process uses an etype built from MPI_INT and another uses an etype 17 built from MPI_FLOAT, the resulting views may be nonportable because the relative 18 sizes of these types may differ from one data representation to another. (End of advice 19 to users.) 202122 MPI_FILE_GET_TYPE_EXTENT(fh, datatype, extent) 23 24 file handle (handle) IN fh 25IN datatype datatype (handle) 26OUT extent datatype extent (integer) 2728C binding 29 int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype, 30  31 MPI_Aint *extent) 32 int MPI_File_get_type_extent_c(MPI_File fh, MPI_Datatype datatype, 33 MPI_Count *extent) 3435 Fortran 2008 binding 36 MPI_File_get_type_extent(fh, datatype, extent, ierror) 37 TYPE(MPI_File), INTENT(IN) :: fh 38 TYPE(MPI_Datatype), INTENT(IN) :: datatype 39 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: extent 40 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 41 MPI_File_get_type_extent(fh, datatype, extent, ierror) 42TYPE(MPI_File), INTENT(IN) :: fh 43 TYPE(MPI_Datatype), INTENT(IN) :: datatype 44 INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: extent 45INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4647Fortran binding 48 MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERROR)

#### INTEGER FH, DATATYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT

Returns the extent of datatype in the file fh. This extent will be the same for all processes accessing the file fh. If the current view uses a user-defined data representation (see Section 14.5.3), MPI uses the dtype_file_extent_fn callback to calculate the extent.

If the datatype extent cannot be represented in extent, it is set to MPI_UNDEFINED.

Advice to implementors. In the case of user-defined data representations, the extent of a derived datatype can be calculated by first determining the extents of the predefined datatypes in this derived datatype using dtype_file_extent_fn (see Section 14.5.3). (End of advice to implementors.)

#### 14.5.2 External Data Representation: "external32"

All MPI implementations are required to support the data representation defined in this section. Support of optional datatypes (e.g., MPI_INTEGER2) is not required.

All floating point values are in big-endian IEEE format [42] of the appropriate size. Floating point values are represented by one of three IEEE formats. These are the IEEE "Single (binary32)," "Double (binary64)," and "Double Extended (binary128)" formats, requiring 4, 8, and 16 bytes of storage, respectively. For the IEEE "Double Extended (binary128)" formats, MPI specifies a Format Width of 16 bytes, with 15 exponent bits, bias = +16383, 112 fraction bits, and an encoding analogous to the "Double (binary64)" format. All integral values are in two's complement big-endian format. Big-endian means most significant byte at lowest address byte. For C _Bool, Fortran LOGICAL, and C++ bool, 0 implies false and nonzero implies true. C float _Complex, double _Complex, and long double _Complex, Fortran COMPLEX and DOUBLE COMPLEX, and other complex types are represented by a pair of floating point format values for the real and imaginary components. Characters are in ISO 8859-1 format [43]. Wide characters (of type MPI_WCHAR) are in Unicode format [68].

All signed numerals (e.g., MPI_INT, MPI_REAL) have the sign bit at the most significant bit. MPI_COMPLEX and MPI_DOUBLE_COMPLEX have the sign bit of the real and imaginary parts at the most significant bit of each part.

According to IEEE specifications [42], the "NaN" (not a number) is system dependent. It should not be interpreted within MPI as anything other than "NaN."

Advice to implementors. The MPI treatment of "NaN" is similar to the approach used in XDR [65]. (End of advice to implementors.)

All data is byte aligned, regardless of type. All data items are stored contiguously in the file (if the file view is contiguous).

Advice to implementors. All bytes of LOGICAL and bool must be checked to determine the value. (End of advice to implementors.)

Advice to users. The type MPI_PACKED is treated as bytes and is not converted. The user should be aware that MPI_PACK has the option of placing a header in the beginning of the pack buffer. (*End of advice to users.*)

1

 $\mathbf{2}$ 

3

4

5

6

7 8

9

10

11

12 13

14 15

16

17

18

19

20

21

22

23

 24 

25

26

27

28

29

30

31

32

33

34

35

36

37 38

39

40 41

42

43 44

45

		T .1	1
1	Predefined Type	Length	
2	MPI_PACKED	1	
3	MPI_BYTE	1	
4	MPI_CHAR	1	
5	MPI_UNSIGNED_CHAR	1	
6	MPI_SIGNED_CHAR	1	
7	MPI_WCHAR	2	
8	MPI_SHORT	2	
9	MPI_UNSIGNED_SHORT	2	
10	MPI_INT	4	
11	MPI_LONG	4	K
12	MPI_UNSIGNED	4	
13	MPI_UNSIGNED_LONG	4	
14	MPI_LONG_LONG_INT	8	
15	MPI_UNSIGNED_LONG_LONG	8	
16	MPI_FLOAT	4	
17	MPI_DOUBLE	8	
18	MPI_LONG_DOUBLE	16	
19	MPI_C_BOOL	1	1
20	MPI_INT8_T	1	
21	MPI_INT16_T	2	
22	MPI_INT32_T	4	
23	MPI_INT64_T	8	
24	MPI_UINT8_T	1	
25	MPI_UINT16_T	2	
26	MPI_UINT32_T	4	
27	MPI_UINT64_T	8	
28	MPI_AINT	8	
29	MPI_COUNT	8	
30	MPI_OFFSET	8	
31	MPI_C_COMPLEX	2*4	
32	MPI_C_FLOAT_COMPLEX	2*4	
33	MPI_C_DOUBLE_COMPLEX	2*8	
34	MPI_C_LONG_DOUBLE_COMPLEX	2*16	
35	MPI_CHARACTER	1	
36	MPI_LOGICAL	4	
37	MPI_INTEGER	4	
38	MPI_REAL	4	
39	MPI_DOUBLE_PRECISION	8	
40	MPI_COMPLEX	2*4	
41	MPI_DOUBLE_COMPLEX	2*8	
42	MPI_CXX_BOOL	1	-
43	MPI_CXX_BOOL MPI_CXX_FLOAT_COMPLEX	$2^{*}4$	
44	MPI_CXX_PLOAT_COMPLEX MPI_CXX_DOUBLE_COMPLEX	$2^{*}4$ $2^{*}8$	
45	MPI_CXX_DOUBLE_COMPLEX MPI_CXX_LONG_DOUBLE_COMPLEX	$2^{*}8$ $2^{*}16$	
46		2 10	
47			

Table 14.2: "external32" sizes of predefined datatypes

Unofficial Draft for Comment Only

47

Predefined Type	Length
MPI_INTEGER1	1
MPI_INTEGER2	2
MPI_INTEGER4	4
MPI_INTEGER8	8
MPI_INTEGER16	16
MPI_REAL2	2
MPI_REAL4	4
MPI_REAL8	8
MPI_REAL16	16
MPI_COMPLEX4	2*2
MPI_COMPLEX8	2*4
MPI_COMPLEX16	2*8
MPI_COMPLEX32	2*16

Table 14.3: "external32"	sizes	of	optional	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

Table 14.4:	"external32"	sizes of	C++	datatypes

1 2	MPI_TYPE_CREATE_F9	0_COMPLEX	types returned from MPI_TYPE_CREATE_F90_REAL, X, and MPI_TYPE_CREATE_F90_INTEGER are defined
3 4	in Section 19.1.9, page 8	05.	
5 6 7 8	the sign bit value.	east significa This allows	hen converting a larger size integer to a smaller size ant bytes are moved. Care must be taken to preserve s no conversion errors if the data range is within the r. ( <i>End of advice to implementors.</i> )
9 10 11	Table 14.2, 14.3, and in "external32" format, re		y the sizes of predefined, optional, and C++ datatypes
12 13	14.5.3 User-Defined D	ata Represe	ntations
14			be handled by the required representations:
15 16			a representation unknown to the implementation, and
17 18	2. a user wants to read	d a file writte	en in a representation unknown to the implementation.
19 20 21	User-defined data returns the I/O stream to do the		as allow the user to insert a third party converter into sentation conversion.
22			
23 24		EP(datarep, e_extent_fn,	read_conversion_fn, write_conversion_fn, extra_state)
25 26	IN datarep		data representation identifier (string)
27 28	IN read_convers	ion_fn	function invoked to convert from file representation to native representation (function)
29 30	IN write_conver	sion_fn	function invoked to convert from native representation to file representation (function)
31 32 33	IN dtype_file_ex	tent_fn	function invoked to get the extent of a datatype as represented in the file (function)
34	IN extra_state		extra state
35			
36	C binding		
37	int MPI_Register_data	-	-
38		-	rsion_function *read_conversion_fn,
39 40		-	rsion_function *write_conversion_fn,
40		-	t_function *dtype_file_extent_fn,
41 42	void *e	xtra_state	)
42 43	int MPI_Register_data	arep_c(cons	st char *datarep,
43	-	-	rsion_function_c *read_conversion_fn,
45	MPI_Dat	arep_conve	rsion_function_c *write_conversion_fn,
46	MPI_Dat	arep_exten	t_function *dtype_file_extent_fn,
47	void *e	xtra_state	)
48			

```
Fortran 2008 binding
 1
 \mathbf{2}
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), INTENT(IN) ::
 read_conversion_fn, write_conversion_fn
 6
 PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 10
MPI_Register_datarep_c(datarep, read_conversion_fn, write_conversion_fn,
 11
 dtype_file_extent_fn, extra_state, ierror)
 12
 CHARACTER(LEN=*), INTENT(IN) :: datarep
 13
 PROCEDURE(MPI_Datarep_conversion_function_c), INTENT(IN) ::
 14
 read_conversion_fn, write_conversion_fn
 15
 PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn
 16
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
 17
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 18
 19
Fortran binding
MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN,
 20
 21
 DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR)
 22
 CHARACTER*(*) DATAREP
 23
 EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN
 24
 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
 25
 INTEGER IERROR
 26
 The call associates read_conversion_fn, write_conversion_fn, and dtype_file_extent_fn
 27
with the data representation identifier datarep. datarep can then be used as an argument
 28
to MPI_FILE_SET_VIEW, causing subsequent data access operations to call the conver-
 29
sion functions to convert all data items accessed between file data representation and na-
 30
tive representation. MPI_REGISTER_DATAREP is a local operation and only registers the
 ^{31}
data representation for the calling MPI process. If datarep is already defined, an error
 32
in the error class MPI_ERR_DUP_DATAREP is raised using the default file error handler
 33
(see Section 14.7). The length of a data representation string is limited to the value of
 34
MPI_MAX_DATAREP_STRING. MPI_MAX_DATAREP_STRING must have a value of at least 64.
 35
No routines are provided to delete data representations and free the associated resources;
 36
it is not expected that an application will generate them in significant numbers.
 37
 38
Extent Callback
 39
 40
typedef int MPI_Datarep_extent_function(MPI_Datatype datatype,
 41
 MPI_Aint *extent, void *extra_state);
 42
ABSTRACT INTERFACE
 43
 SUBROUTINE MPI_Datarep_extent_function(datatype, extent, extra_state,
 44
 ierror)
 45
 TYPE(MPI_Datatype) :: datatype
 46
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extent, extra_state
 47
 INTEGER :: ierror
 48
```

1 2	SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR) INTEGER DATATYPE, IERROR
3	INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE
4 5	The function dtype_file_extent_fn must return, in file_extent, the number of bytes re-
6	quired to store datatype in the file representation. The function is passed, in extra_state,
7	the argument that was passed to the MPI_REGISTER_DATAREP call. MPI will only call
8	this routine with predefined datatypes employed by the user.
9	Rationale. This callback does not have a large count variant because it is anticipated
10 11	that large counts will not be required to represent the extent output value. (End of
12	rationale.)
13	MPI_DATAREP_CONVERSION_FUNCTION also supports large count types in separate
14	additional MPI procedures in C (suffixed with the "_c") and interface polymorphism in
15 16	Fortran when using USE mpi_f08.
17	If the extent cannot be represented in extent, the callback function shall set extent to
18	MPI_UNDEFINED. The MPI implementation will then raise an error of class MPI_ERR_VALUE_TOO_LARGE.
19	
20 21	Datarep Conversion Functions
22	typedef int MPI_Datarep_conversion_function(void *userbuf,
23	MPI_Datatype datatype, int count, void *filebuf,
24	<pre>MPI_Offset position, void *extra_state);</pre>
25 26	<pre>typedef int MPI_Datarep_conversion_function_c(void *userbuf,</pre>
27	MPI_Datatype datatype, MPI_Count count, void *filebuf,
28	<pre>MPI_Offset position, void *extra_state);</pre>
29	ABSTRACT INTERFACE
30 31	SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count,
32	filebuf, position, extra_state, ierror) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
33	TYPE(C_PTR), VALUE :: userbuf, filebuf
34	TYPE(MPI_Datatype) :: datatype
35	INTEGER :: count, ierror
36 37	INTEGER(KIND=MPI_OFFSET_KIND) :: position
38	INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
39	ABSTRACT INTERFACE
40	SUBROUTINE MPI_Datarep_conversion_function_c(userbuf, datatype, count, filebuf, position, extra_state, ierror)
41 42	USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
42	TYPE(C_PTR), VALUE :: userbuf, filebuf
44	TYPE(MPI_Datatype) :: datatype
45	INTEGER(KIND=MPI_COUNT_KIND) :: count
46	INTEGER(KIND=MPI_OFFSET_KIND) :: position INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
47 48	INTEGER :: ierror
40	

The function read_conversion_fn must convert from file data representation to native representation. Before calling this routine, MPI allocates and fills filebuf with count contiguous data items. The type of each data item matches the corresponding entry for the predefined datatype in the type signature of datatype. The function is passed, in extra_state, the argument that was passed to the MPI_REGISTER_DATAREP call. The function must copy all count data items from filebuf to userbuf in the distribution described by datatype, converting each data item from file representation to native representation. datatype will be equivalent to the datatype that the user passed to the read function. If the size of datatype is less than the size of the count data items, the conversion function must treat datatype as being contiguously tiled over the userbuf. The conversion function must begin storing converted data at the location in userbuf specified by position into the (tiled) datatype.

Advice to users. Although the conversion functions have similarities to MPI_PACK and MPI_UNPACK, one should note the differences in the use of the arguments count and position. In the conversion functions, count is a count of data items (i.e., count of typemap entries of datatype), and position is an index into this typemap. In MPI_PACK, incount refers to the number of whole datatypes, and position is a number of bytes. (*End of advice to users.*)

Advice to implementors. A converted read operation could be implemented as follows:

- 1. Get file extent of all data items
- 2. Allocate a filebuf large enough to hold all count data items
- 3. Read data from file into filebuf
- 4. Call read_conversion_fn to convert data and place it into userbuf
- 5. Deallocate filebuf

#### (End of advice to implementors.)

If MPI cannot allocate a buffer large enough to hold all the data to be converted from a read operation, it may call the conversion function repeatedly using the same datatype and userbuf, and reading successive chunks of data to be converted in filebuf. For the first call (and in the case when all the data to be converted fits into filebuf), MPI will call the function with position set to zero. Data converted during this call will be stored in the userbuf according to the first count data items in datatype. Then in subsequent calls to the conversion function, MPI will increment the value in position by the count of items converted in the previous call, and the userbuf pointer will be unchanged.

Rationale. Passing the conversion function a position and one datatype for the transfer allows the conversion function to decode the datatype only once and cache an internal representation of it on the datatype. Then on subsequent calls, the conversion 48

#### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

5 6

7

8

9

10

11

12

13

14

15

16

17

18 19

20

21

22

23

 24 

25 26

27 28

29

30

31 32

33

34 35

36 37

38

39

40

41

42

43

44

function can use the **position** to quickly find its place in the datatype and continue storing converted data where it left off at the end of the previous call. (*End of rationale.*)

Advice to users. Although the conversion function may usefully cache an internal representation on the datatype, it should not cache any state information specific to an ongoing conversion operation, since it is possible for the same datatype to be used concurrently in multiple conversion operations. (*End of advice to users.*)

9 The function write_conversion_fn must convert from native representation to file data 10 representation. Before calling this routine, MPI allocates filebuf of a size large enough to 11 hold **count** contiguous data items. The type of each data item matches the corresponding 12entry for the predefined datatype in the type signature of datatype. The function must copy 13 count data items from userbuf in the distribution described by datatype, to a contiguous 14distribution in filebuf, converting each data item from native representation to file repre-15sentation. If the size of datatype is less than the size of count data items, the conversion 16function must treat datatype as being contiguously tiled over the userbuf.

The function must deat datatype as being contiguously inclusive the discribut. The function must begin copying at the location in userbuf specified by position into the (tiled) datatype. datatype will be equivalent to the datatype that the user passed to the write function. The function is passed, in extra_state, the argument that was passed to the MPI_REGISTER_DATAREP call.

The predefined constant MPI_CONVERSION_FN_NULL may be used as either write_conversion_fn or read_conversion_fn. In that case, MPI will not attempt to invoke write_conversion_fn or read_conversion_fn, respectively, but will perform the requested data access using the native data representation.

An MPI implementation must ensure that all data accessed is converted, either by using a filebuf large enough to hold all the requested data items or else by making repeated calls to the conversion function with the same datatype argument and appropriate values for position.

An implementation will only invoke the callback routines in this section

(read_conversion_fn, write_conversion_fn, and dtype_file_extent_fn) when one of the read or write routines in Section 14.4, or MPI_FILE_GET_TYPE_EXTENT is called by the user. dtype_file_extent_fn will only be passed predefined datatypes employed by the user. The conversion functions will only be passed datatypes equivalent to those that the user has passed to one of the routines noted above.

The conversion functions must be reentrant. User defined data representations are restricted to use byte alignment for all types. Furthermore, it is erroneous for the conversion functions to call any collective routines or to free datatype.

The conversion functions should return an error code. If the returned error code has a value other than MPI_SUCCESS, the implementation will raise an error in the class MPI_ERR_CONVERSION.

40 41 42

43

39

## 14.5.4 Matching Data Representations

⁴⁴ It is the user's responsibility to ensure that the data representation used to read data from ⁴⁵ 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 repre sentation names on two different implementations may yield compatible representations.

## Unofficial Draft for Comment Only

1

2

3

4

5

6

7

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 from a single collective open (e.g.,  $fh_{1a} \in FH_1$  and  $fh_{1b} \in FH_1$ ), and two file handles from different collective opens (e.g.,  $fh_1 \in FH_1$  and  $fh_2 \in FH_2$ ). 

For the purpose of consistency semantics, a matched pair (Section 14.4.5) of split collective data access operations (e.g., MPI_FILE_READ_ALL_BEGIN and 45 MPI_FILE_READ_ALL_END) compose a single data access operation. Similarly, a nonblocking data access routine (e.g., MPI_FILE_IREAD) and the routine which completes the 47 request (e.g., MPI_WAIT) also compose a single data access operation. For all cases below, 48

#### Unofficial Draft for Comment Only

2 3 4

1

5 6

7

12

Advice to users. For an MPI_FILE_IREAD and MPI_WAIT pair, the operation begins when MPI_FILE_IREAD is called and ends when MPI_WAIT returns. (*End of advice to users.*)

Assume that  $A_1$  and  $A_2$  are two data access operations. Let  $D_1$  ( $D_2$ ) be the set of absolute byte displacements of every byte accessed in  $A_1$  ( $A_2$ ). The two data accesses *overlap* if  $D_1 \cap D_2 \neq \emptyset$ . The two data accesses *conflict* if they overlap and at least one is a write access.

Let  $SEQ_{fh}$  be a sequence of file operations on a single file handle, bracketed by

¹³ MPI_FILE_SYNCs on that file handle. (Both opening and closing a file implicitly perform ¹⁴ an MPI_FILE_SYNC.)  $SEQ_{fh}$  is a "write sequence" if any of the data access operations in ¹⁵ the sequence are writes or if any of the file manipulation operations in the sequence change ¹⁶ the state of the file (e.g., MPI_FILE_SET_SIZE or MPI_FILE_PREALLOCATE). Given two ¹⁷ sequences,  $SEQ_1$  and  $SEQ_2$ , we say they are not *concurrent* if one sequence is guaranteed ¹⁸ to completely precede the other (temporally).

The requirements for guaranteeing sequential consistency among all accesses to a particular file are divided into the three cases given below. If any of these requirements are not met, then the value of all data in that file is implementation dependent.

22

²³ Case 1:  $fh_1 \in FH_1$  All operations on  $fh_1$  are sequentially consistent if atomic mode is ²⁴ set. If nonatomic mode is set, then all operations on  $fh_1$  are sequentially consistent if they ²⁵ are either nonconcurrent, nonconflicting, or both.

26 27

28

29

Case 2:  $fh_{1a} \in FH_1$  and  $fh_{1b} \in FH_1$  Assume  $A_1$  is a data access operation using  $fh_{1a}$ , and  $A_2$  is a data access operation using  $fh_{1b}$ . If for any access  $A_1$ , there is no access  $A_2$ that conflicts with  $A_1$ , then MPI guarantees sequential consistency.

³⁰ However, unlike POSIX semantics, the default MPI semantics for conflicting accesses ³¹ do not guarantee sequential consistency. If  $A_1$  and  $A_2$  conflict, sequential consistency can ³² be guaranteed by either enabling atomic mode via the MPI_FILE_SET_ATOMICITY routine, ³³ or meeting the condition described in Case 3 below.

³⁵ Case 3:  $fh_1 \in FH_1$  and  $fh_2 \in FH_2$  Consider access to a single file using file handles from ³⁶ distinct collective opens. In order to guarantee sequential consistency, MPI_FILE_SYNC ³⁷ must be used (both opening and closing a file implicitly perform an MPI_FILE_SYNC).

³⁸ Sequential consistency is guaranteed among accesses to a single file if for any write ³⁹ sequence  $SEQ_1$  to the file, there is no sequence  $SEQ_2$  to the file which is *concurrent* with ⁴⁰  $SEQ_1$ . To guarantee sequential consistency when there are write sequences,

⁴¹ MPI_FILE_SYNC must be used together with a mechanism that guarantees nonconcurrency
 ⁴² of the sequences.

43 See the examples in Section 14.6.11 for further clarification of some of these consistency
 44 semantics.

- 45
- 46 47
- 48

MPI FILE	_SET_ATOMICITY(fh, flag)		1
INOUT	fh	file handle (handle)	2
IN			3
IIN	flag	true to set atomic mode, false to set nonatomic mode (logical)	4
		(logical)	5 6
C bindin	Ø		7
	File_set_atomicity(MPI_Fi	le fh, int flag)	8
Fortron	2008 binding	-	9
	2008 binding _set_atomicity(fh, flag,	ierror)	10
	(MPI_File), INTENT(IN) ::		11
	CAL, INTENT(IN) :: flag		12
INTE	GER, OPTIONAL, INTENT(OUT	) :: ierror	13 14
Fortran	hinding		15
	_SET_ATOMICITY(FH, FLAG,	IERROR)	16
	GER FH, IERROR		17
LOGI	CAL FLAG		18
Let <i>I</i>	FH be the set of file handles	created by one collective open. The consistency	19
		sing $FH$ is set by collectively calling	20
	_	MPI_FILE_SET_ATOMICITY is collective; all pro-	21 22
cesses in t	the group must pass identical	values for fh and flag. If flag is true, atomic mode is	22
, .	is false, nonatomic mode is se		24
		cs for an open file only affects new data accesses.	25
_		seed to abide by the consistency semantics in effect	26
-		ta accesses and split collective operations that have are only guaranteed to abide by nonatomic mode	27
-	cy semantics.	the only guaranteed to able by nonatonne mode	28
00110100010	y semanolos.		29
Adv	ice to implementors. Since the	e semantics guaranteed by atomic mode are stronger	30 31
	5	mic mode, an implementation is free to adhere to	32
		semantics for outstanding requests. (End of advice	33
	nplementors.)		34
			35
			36
MPI_FILE	_GET_ATOMICITY(fh, flag)		37 38
IN	fh	file handle (handle)	39
OUT	flag	true if atomic mode, false if nonatomic mode (logical)	40
			41
C bindin	g		42
int MPI_	File_get_atomicity(MPI_Fi	le fh, int *flag)	43
Fortran	2008 binding		44
	_get_atomicity(fh, flag,	ierror)	45
	(MPI_File), INTENT(IN) ::	fh	46 47
LOGI	CAL, INTENT(OUT) :: flag		48

```
1
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
2
 Fortran binding
3
 MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR)
4
 INTEGER FH, IERROR
5
 LOGICAL FLAG
6
7
 MPI_FILE_GET_ATOMICITY returns the current consistency semantics for data access
8
 operations on the set of file handles created by one collective open. If flag is true, atomic
9
 mode is enabled; if flag is false, nonatomic mode is enabled.
10
11
 MPI_FILE_SYNC(fh)
12
13
 INOUT
 fh
 file handle (handle)
14
15
 C binding
16
 int MPI_File_sync(MPI_File fh)
17
 Fortran 2008 binding
18
 MPI_File_sync(fh, ierror)
19
 TYPE(MPI_File), INTENT(IN) :: fh
20
21
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
 Fortran binding
23
 MPI_FILE_SYNC(FH, IERROR)
^{24}
 INTEGER FH, IERROR
25
26
 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
27
 device, then all such updates become visible to subsequent reads of fh by the calling process.
28
 MPI_FILE_SYNC may be necessary to ensure sequential consistency in certain cases (see
29
30
 above).
 MPI_FILE_SYNC is a collective operation.
^{31}
 The user is responsible for ensuring that all nonblocking requests and split collective
32
 operations on fh have been completed before calling MPI_FILE_SYNC—otherwise, the call
33
34
 to MPI_FILE_SYNC is erroneous.
35
36
 14.6.2
 Random Access vs. Sequential Files
37
 MPI distinguishes ordinary random access files from sequential stream files, such as pipes
38
 and tape files. Sequential stream files must be opened with the MPI_MODE_SEQUENTIAL
39
 flag set in the amode. For these files, the only permitted data access operations are shared
40
 file pointer reads and writes. Filetypes and etypes with holes are erroneous. In addition, the
41
 notion of file pointer is not meaningful; therefore, calls to MPI_FILE_SEEK_SHARED and
42
 MPI_FILE_GET_POSITION_SHARED are erroneous, and the pointer update rules specified
43
 for the data access routines do not apply. The amount of data accessed by a data access
44
 operation will be the amount requested unless the end of file is reached or an error is raised.
45
46
 Rationale. This implies that reading on a pipe will always wait until the requested
47
 amount of data is available or until the process writing to the pipe has issued an end
48
 of file. (End of rationale.)
```

 24 

Finally, for some sequential files, such as those corresponding to magnetic tapes or streaming network connections, writes to the file may be destructive. In other words, a write may act as a truncate (a MPI_FILE_SET_SIZE with size set to the current position) followed by the write.

#### 14.6.3 Progress

The progress rules of MPI are both a promise to users and a set of constraints on implementors. In cases where the progress rules restrict possible implementation choices more than the interface specification alone, the progress rules take precedence.

All blocking routines must complete in finite time unless an exceptional condition (such as resource exhaustion) causes an error.

Nonblocking data access routines inherit the following progress rule from nonblocking point-to-point communication: a nonblocking write is equivalent to a nonblocking send for which a receive is eventually posted, and a nonblocking read is equivalent to a nonblocking receive for which a send is eventually posted.

Finally, an implementation is free to delay progress of collective routines until all processes in the group associated with the collective call have invoked the routine. Once all processes in the group have invoked the routine, the progress rule of the equivalent noncollective routine must be followed.

#### 14.6.4 Collective File Operations

Collective file operations are subject to the same restrictions as collective communication operations. For a complete discussion, please refer to the semantics set forth in Section 6.14.

Collective file operations are collective over a duplicate of the communicator used to open the file—this duplicate communicator is implicitly specified via the file handle argument. Different processes can pass different values for other arguments of a collective routine unless specified otherwise.

#### 14.6.5 Nonblocking Collective File Operations

Nonblocking collective file operations are defined only for data access routines with explicit offsets and individual file pointers but not with shared file pointers.

Nonblocking collective file operations are subject to the same restrictions as blocking collective I/O operations. All processes belonging to the group of the communicator that was used to open the file must call collective I/O operations (blocking and nonblocking) in the same order. This is consistent with the ordering rules for collective operations in threaded environments. For a complete discussion, please refer to the semantics set forth in Section 6.14.

Nonblocking collective I/O operations do not match with blocking collective I/O operations. Multiple nonblocking collective I/O operations can be outstanding on a single file handle. High quality MPI implementations should be able to support a large number of pending nonblocking I/O operations.

All nonblocking collective I/O calls are local and return immediately, irrespective of the status of other processes. The call initiates the operation which may progress independently of any communication, computation, or I/O. The call returns a request handle, which must be passed to a completion call. Input buffers should not be modified and output buffers should not be accessed before the completion call returns. The same progress rules described 

for nonblocking collective operations apply for nonblocking collective I/O operations. For  $\mathbf{2}$ a complete discussion, please refer to the semantics set forth in Section 6.12.

3 4

5

6

7

8

9

1

#### Type Matching 14.6.6

The type matching rules for I/O mimic the type matching rules for communication with one exception: if etype is MPI_BYTE, then this matches any datatype in a data access operation. In general, the etype of data items written must match the etype used to read the items, and for each data access operation, the current etype must also match the type declaration of the data access buffer.

10 11

1213

14

1516

17

In most cases, use of MPI_BYTE as a wild card will defeat the Advice to users. file interoperability features of MPI. File interoperability can only perform automatic conversion between heterogeneous data representations when the exact datatypes accessed are explicitly specified. (End of advice to users.)

#### 14.6.7 Miscellaneous Clarifications

18 Once an I/O routine completes, it is safe to free any opaque objects passed as arguments 19to that routine. For example, the comm and info used in an MPI_FILE_OPEN, or the etype 20and filetype used in an MPI_FILE_SET_VIEW, can be freed without affecting access to the 21file. Note that for nonblocking routines and split collective operations, the operation must 22 be completed before it is safe to reuse data buffers passed as arguments.

23As in communication, datatypes must be committed before they can be used in file  24 manipulation or data access operations. For example, the etype and filetype must be com-25mitted before calling MPI_FILE_SET_VIEW, and the datatype must be committed before 26calling MPI_FILE_READ or MPI_FILE_WRITE.

2728

29

33

#### 14.6.8 MPI_Offset Type

MPI_Offset is an integer type of size sufficient to represent the size (in bytes) of the largest 30  31 file supported by MPI. Displacements and offsets are always specified as values of type MPI_Offset. 32

In Fortran, the corresponding integer is an integer with kind parameter

34MPI_OFFSET_KIND, which is defined in the mpi_f08 module, the mpi module and the mpif.h include file. 35

36 In Fortran 77 environments that do not support KIND parameters, MPI_Offset arguments 37 should be declared as an INTEGER of suitable size. The language interoperability implications for MPI_Offset are similar to those for addresses (see Section 19.3). 38

39 40

41

#### 14.6.9 Logical vs. Physical File Layout

MPI specifies how the data should be laid out in a virtual file structure (the view), not 42how that file structure is to be stored on one or more disks. Specification of the physical 43 file structure was avoided because it is expected that the mapping of files to disks will be 44 system specific, and any specific control over file layout would therefore restrict program 45 portability. However, there are still cases where some information may be necessary to 46 optimize file layout. This information can be provided as *hints* specified via info when a file 47is created (see Section 14.2.8). 48

The size of a file may be increased by writing to the file after the current end of file. The size may also be changed by calling MPI *size changing* routines, such as MPI_FILE_SET_SIZE. A call to a size changing routine does not necessarily change the file size. For example, calling MPI_FILE_PREALLOCATE with a size less than the current size does not change the size.

Consider a set of bytes that has been written to a file since the most recent call to a size changing routine, or since MPI_FILE_OPEN if no such routine has been called. Let the *high byte* be the byte in that set with the largest displacement. The file size is the larger of

- One plus the displacement of the high byte.
- The size immediately after the size changing routine, or MPI_FILE_OPEN, returned.

When applying consistency semantics, calls to MPI_FILE_SET_SIZE and MPI_FILE_PREALLOCATE are considered writes to the file (which conflict with operations that access bytes at displacements between the old and new file sizes), and MPI_FILE_GET_SIZE is considered a read of the file (which overlaps with all accesses to the file).

Advice to users. Any sequence of operations containing the collective routines MPI_FILE_SET_SIZE and MPI_FILE_PREALLOCATE is a write sequence. As such, sequential consistency in nonatomic mode is not guaranteed unless the conditions in Section 14.6.1 are satisfied. (*End of advice to users.*)

File pointer update semantics (i.e., file pointers are updated by the amount accessed) are only guaranteed if file size changes are sequentially consistent.

Advice to users. Consider the following example. Given two operations made by separate processes to a file containing 100 bytes: an MPI_FILE_READ of 10 bytes and an MPI_FILE_SET_SIZE to 0 bytes. If the user does not enforce sequential consistency between these two operations, the file pointer may be updated by the amount requested (10 bytes) even if the amount accessed is zero bytes. (*End of advice to users.*)

## 14.6.11 Examples

The examples in this section illustrate the application of the MPI consistency and semantics guarantees. These address

- conflicting accesses on file handles obtained from a single collective open, and
- all accesses on file handles obtained from two separate collective opens.

The simplest way to achieve consistency for conflicting accesses is to obtain sequential consistency by setting atomic mode. For the code below, process 1 will read either 0 or 10 integers. If the latter, every element of **b** will be 5. If nonatomic mode is set, the results of the read are undefined.

 31 

```
1
 /* Process 0 */
\mathbf{2}
 int i, a[10];
3
 int TRUE = 1;
4
\mathbf{5}
 for (i=0;i<10;i++)
6
 a[i] = 5;
7
8
 MPI_File_open(MPI_COMM_WORLD, "workfile",
9
 MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0);
10
 MPI_File_set_view(fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
11
 MPI_File_set_atomicity(fh0, TRUE);
12
 MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status);
13
 /* MPI_Barrier(MPI_COMM_WORLD); */
14
 /* Process 1 */
15
16
 int b[10];
17
 int TRUE = 1;
 MPI_File_open(MPI_COMM_WORLD, "workfile",
18
 MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1);
19
 MPI_File_set_view(fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
20
 MPI_File_set_atomicity(fh1, TRUE);
21
 /* MPI_Barrier(MPI_COMM_WORLD); */
22
 MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status);
23
^{24}
 A user may guarantee that the write on process 0 precedes the read on process 1 by imposing
25
 temporal order with, for example, calls to MPI_BARRIER.
26
27
 Advice to users. Routines other than MPI_BARRIER may be used to impose temporal
28
 order. In the example above, process 0 could use MPI_SEND to send a 0 byte message,
29
 received by process 1 using MPI_RECV. (End of advice to users.)
30
31
 Alternatively, a user can impose consistency with nonatomic mode set:
32
33
 /* Process 0 */
34
 int i, a[10];
35
 for (i=0;i<10;i++)</pre>
36
 a[i] = 5;
37
38
 MPI_File_open(MPI_COMM_WORLD, "workfile",
39
 MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0);
40
 MPI_File_set_view(fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
^{41}
 MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status);
42
 MPI_File_sync(fh0);
43
 MPI_Barrier(MPI_COMM_WORLD);
44
 MPI_File_sync(fh0);
45
46
 /* Process 1 */
47
 int b[10];
48
 MPI_File_open(MPI_COMM_WORLD, "workfile",
```

```
1
 MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1);
 \mathbf{2}
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);
 6
The "sync-barrier-sync" construct is required because:
 9
 • The barrier ensures that the write on process 0 occurs before the read on process 1.
 10
 • The first sync guarantees that the data written by all processes is transferred to the
 11
 storage device.
 12
 13
 • The second sync guarantees that all data which has been transferred to the storage
 14
 device is visible to all processes. (This does not affect process 0 in this example.)
 15
 The following program represents an erroneous attempt to achieve consistency by elim-
 16
inating the apparently superfluous second "sync" call for each process.
 17
 18
/* ----- THIS EXAMPLE IS ERRONEOUS ---
 ____ */
 19
/* Process 0 */
 20
int i, a[10];
 21
for (i=0;i<10;i++)</pre>
 22
 a[i] = 5;
 23
 ^{24}
MPI_File_open(MPI_COMM_WORLD, "workfile",
 25
 MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0);
 26
MPI_File_set_view(fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
 27
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status);
 28
MPI_File_sync(fh0);
 29
MPI_Barrier(MPI_COMM_WORLD);
 30
/* Process 1 */
 31
int b[10];
 32
MPI_File_open(MPI_COMM_WORLD, "workfile",
 33
 MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1);
 34
MPI_File_set_view(fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
 35
MPI_Barrier(MPI_COMM_WORLD);
 36
MPI_File_sync(fh1);
 37
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status);
 38
 39
 ----- THIS EXAMPLE IS ERRONEOUS ------ */
 40
 41
The above program also violates the MPI rule against out-of-order collective operations and
 42
```

will deadlock for implementations in which MPI_FILE_SYNC blocks.

Advice to users. Some implementations may choose to implement MPI_FILE_SYNC 44 as a temporally synchronizing function. When using such an implementation, the 45 "sync-barrier-sync" construct above can be replaced by a single "sync." The results of 46 using such code with an implementation for which MPI_FILE_SYNC is not temporally 47 synchronizing is undefined. (*End of advice to users.*) 48

#### Unofficial Draft for Comment Only

```
1
 Asynchronous I/O
\mathbf{2}
 The behavior of asynchronous I/O operations is determined by applying the rules specified
3
 above for synchronous I/O operations.
4
 The following examples all access a preexisting file "myfile." Word 10 in myfile initially
5
 contains the integer 2. Each example writes and reads word 10.
6
 First consider the following code fragment:
7
8
 int a = 4, b, TRUE=1;
9
 MPI_File_open(MPI_COMM_WORLD, "myfile",
10
 MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
11
 MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
12
 /* MPI_File_set_atomicity(fh, TRUE); Use this to set atomic mode. */
13
 MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]);
14
 MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, ®s[1]);
15
 MPI_Waitall(2, reqs, statuses);
16
 For asynchronous data access operations, MPI specifies that the access occurs at any time
17
 between the call to the asynchronous data access routine and the return from the corre-
18
 sponding request complete routine. Thus, executing either the read before the write, or the
19
 write before the read is consistent with program order. If atomic mode is set, then MPI
20
21
 guarantees sequential consistency, and the program will read either 2 or 4 into b. If atomic
 mode is not set, then sequential consistency is not guaranteed and the program may read
22
 something other than 2 or 4 due to the conflicting data access.
23
 Similarly, the following code fragment does not order file accesses:
^{24}
25
 int a = 4, b;
26
 MPI_File_open(MPI_COMM_WORLD, "myfile",
27
 MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
28
 MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
29
 /* MPI_File_set_atomicity(fh, TRUE); Use this to set atomic mode. */
30
 MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]);
31
 MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]);
32
 MPI_Wait(&reqs[0], &status);
33
 MPI_Wait(&reqs[1], &status);
34
35
 If atomic mode is set, either 2 or 4 will be read into b. Again, MPI does not guarantee
36
 sequential consistency in nonatomic mode.
37
 On the other hand, the following code fragment:
38
39
 int a = 4, b;
 MPI_File_open(MPI_COMM_WORLD, "myfile",
40
 MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
41
 MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
42
 MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]);
43
 MPI_Wait(&reqs[0], &status);
44
 MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]);
45
 MPI_Wait(®s[1], &status);
46
47
 defines the same ordering as:
48
```

```
1
int a = 4, b;
 \mathbf{2}
MPI_File_open(MPI_COMM_WORLD, "myfile",
 3
 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);
 6
Since
 • nonconcurrent operations on a single file handle are sequentially consistent, and
 10
 11
 • the program fragments specify an order for the operations.
 12
MPI guarantees that both program fragments will read the value 4 into b. There is no need
 13
to set atomic mode for this example.
 14
 Similar considerations apply to conflicting accesses of the form:
 15
 16
MPI_File_iwrite_all(fh,...);
 17
MPI_File_iread_all(fh,...);
 18
MPI_Waitall(...);
 19
 In addition, as mentioned in Section 14.6.5, nonblocking collective I/O operations have
 20
to be called in the same order on the file handle by all processes.
 21
 Similar considerations apply to conflicting accesses of the form:
 22
 23
MPI_File_write_all_begin(fh,...);
 ^{24}
MPI_File_iread(fh,...);
 25
MPI_Wait(fh,...);
 26
MPI_File_write_all_end(fh,...);
 27
 28
 Recall that constraints governing consistency and semantics are not relevant to the
 29
following:
 30
MPI_File_write_all_begin(fh,...);
 31
MPI_File_read_all_begin(fh,...);
 32
MPI_File_read_all_end(fh,...);
 33
MPI_File_write_all_end(fh,...);
 34
 35
since split collective operations on the same file handle may not overlap (see Section 14.4.5).
 36
 37
 I/O Error Handling
 38
14.7
 39
```

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.)

## Unofficial Draft for Comment Only

40

 $41 \\ 42$ 

43 44

45

46

47

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.

8 I/O error handling differs from communication error handling in another important 9 aspect. By default, the predefined error handler for file handles is MPI_ERRORS_RETURN. 10 The **default file error** handler has two purposes: when a new file handle is created (by 11MPI_FILE_OPEN), the error handler for the new file handle is initially set to the default 12file error handler, and I/O routines that have no valid file handle on which to raise an 13 error (e.g., MPI_FILE_OPEN or MPI_FILE_DELETE) use the default file error handler. The 14default file error handler can be changed by specifying MPI_FILE_NULL as the fh argument 15to MPI_FILE_SET_ERRHANDLER. The current value of the default file error handler can 16be determined by passing MPI_FILE_NULL as the fh argument to

¹⁷ 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.*)

23 24 25

26 27

28

29

30

31 32

33 34

35 36

37

38

18

19

20

21

22

# 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().

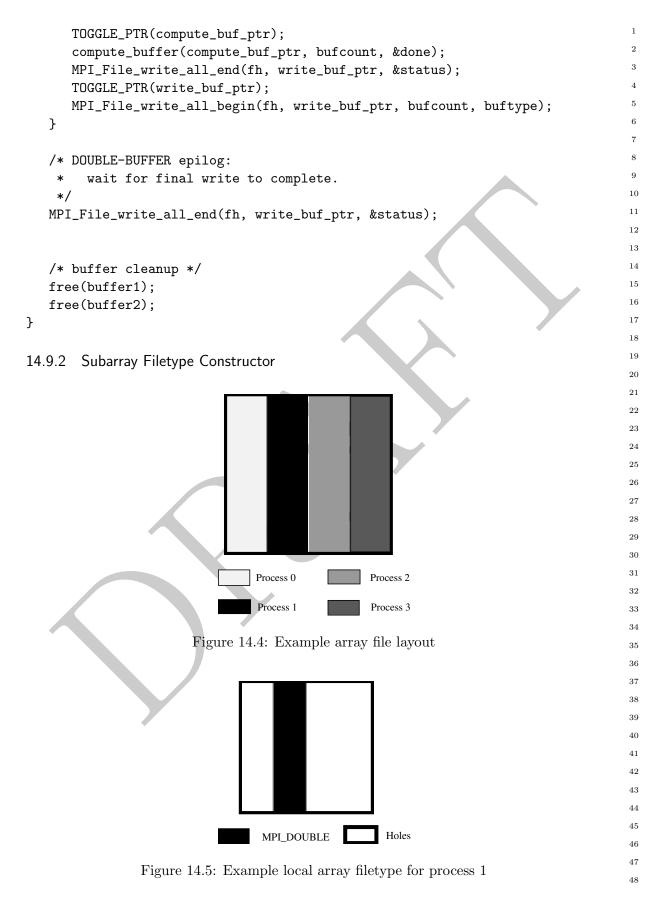
```
39
 /*
40
41
 Function:
 double_buffer
42
 *
43
 *
 Synopsis:
44
 *
 void double_buffer(
45
 MPI_File fh,
 IN
 *
 **
46
 *
 MPI_Datatype buftype,
 IN
47
 int bufcount
 ** IN
 *
48
)
 *
```

1

 $\mathbf{2}$ 

MPI_ERR_FILE	Invalid file handle	1
MPI_ERR_NOT_SAME	Collective argument not identical on all	2
	processes, or collective routines called in	3
	a different order by different processes	4
MPI_ERR_AMODE	Error related to the amode passed to	5
	MPI_FILE_OPEN	6
MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to	7
	MPI_FILE_SET_VIEW	8
MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on	9
	a file which supports sequential access only	10
MPI_ERR_NO_SUCH_FILE	File does not exist	11
MPI_ERR_FILE_EXISTS	File exists	12
MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)	13
MPI_ERR_ACCESS	Permission denied	14
MPI_ERR_NO_SPACE	Not enough space	15
MPI_ERR_QUOTA	Quota exceeded	16
MPI_ERR_READ_ONLY	Read-only file or file system	17
MPI_ERR_FILE_IN_USE	File operation could not be completed, as	18
	the file is currently open by some process	19
MPI_ERR_DUP_DATAREP	Conversion functions could not be regis-	20
	tered because a data representation identi-	21
	fier that was already defined was passed to	22
	MPI_REGISTER_DATAREP	23
MPI_ERR_CONVERSION	An error occurred in a user supplied data	24
	conversion function.	25
MPI_ERR_IO	Other I/O error	26
Table 14.5	: I/O Error Classes	27
10010 11.0		28
		29 30
		31
		32
		33
		34
		35
		36
		37
		38
		39
•		40
		41
		42
		43
		44
		45
		46
		47

```
1
 *
\mathbf{2}
 * Description:
3
 Performs the steps to overlap computation with a collective write
 *
4
 by using a double-buffering technique.
 *
5
6
 * Parameters:
7
 *
 fh
 previously opened MPI file handle
8
 buftype
 MPI datatype for memory layout
 *
9
 (Assumes a compatible view has been set on fh)
 *
10
 # buftype elements to transfer
 *
 bufcount
11
 -----/
12
13
 /* this macro switches which buffer "x" is pointing to */
14
 #define TOGGLE_PTR(x) (((x)==(buffer1)) ? (x=buffer2) : (x=buffer1))
15
16
 void double_buffer(MPI_File fh, MPI_Datatype buftype, int bufcount)
17
 {
18
19
 MPI_Status status;
 /* status for MPI calls */
20
 float *buffer1, *buffer2; /* buffers to hold results */
21
 float *compute_buf_ptr; /* destination buffer */
22
 /* for computing */
23
 float *write_buf_ptr; /* source for writing */
^{24}
 /* determines when to quit */
 int done;
25
26
 /* buffer initialization */
27
 buffer1 = (float *)
 malloc(bufcount*sizeof(float));
28
29
 buffer2 = (float *)
30
 malloc(bufcount*sizeof(float));
31
 compute_buf_ptr = buffer1; /* initially point to buffer1 */
32
 write_buf_ptr = buffer1; /* initially point to buffer1 */
33
34
35
 /* DOUBLE-BUFFER prolog:
36
 compute buffer1; then initiate writing buffer1 to disk
 *
37
 */
38
 compute_buffer(compute_buf_ptr, bufcount, &done);
39
 MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
40
41
 /* DOUBLE-BUFFER steady state:
42
 * Overlap writing old results from buffer pointed to by write_buf_ptr
43
 * with computing new results into buffer pointed to by compute_buf_ptr.
44
 *
45
 * There is always one write-buffer and one compute-buffer in use
46
 * during steady state.
47
 */
48
 while (!done) {
```



```
1
 Assume we are writing out a 100 \times 100 2D array of double precision floating point
\mathbf{2}
 numbers that is distributed among 4 processes such that each process has a block of 25
3
 columns (e.g., process 0 has columns 0–24, process 1 has columns 25–49, etc.; see Fig-
4
 ure 14.4). To create the filetypes for each process one could use the following C program
\mathbf{5}
 (see Section 5.1.3):
6
 double subarray[100][25];
7
 MPI_Datatype filetype;
8
 int sizes[2], subsizes[2], starts[2];
9
10
 int rank;
11
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
12
 sizes[0]=100; sizes[1]=100;
13
 subsizes[0]=100; subsizes[1]=25;
14
 starts[0]=0; starts[1]=rank*subsizes[1];
15
16
 MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C,
17
 MPI_DOUBLE, &filetype);
18
19
 Or, equivalently in Fortran:
20
21
 double precision subarray(100,25)
22
 integer filetype, rank, ierror
23
 integer sizes(2), subsizes(2), starts(2)
24
25
 call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
26
 sizes(1)
 = 100
27
 sizes(2)
 = 100
28
 subsizes(1) = 100
29
 subsizes(2) = 25
30
 starts(1)
 = 0
^{31}
 starts(2)
 = rank*subsizes(2)
32
33
 call MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts, &
34
 MPI_ORDER_FORTRAN, MPI_DOUBLE_PRECISION,
 &
35
 filetype, ierror)
36
37
 The generated filetype will then describe the portion of the file contained within the
38
 process's subarray with holes for the space taken by the other processes. Figure 14.5 shows
39
 the filetype created for process 1.
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 operation of MPI processes. Specifically, this chapter defines both the MPI profiling interface (Section 15.2), which supports the transparent interception and inspection of MPI calls, and the MPI tool information interface (Section 15.3), which supports the inspection and manipulation of MPI control and performance variables, as well as the registration of callbacks for MPI library events. The interfaces described in this chapter are all defined in the context of an MPI process, i.e., are callable from the same code that invokes other MPI functions.

## 15.2 Profiling Interface

#### 15.2.1 Requirements

To meet the requirements for the  $\mathsf{MPI}$  profiling interface, an implementation of the  $\mathsf{MPI}$  functions must

1. provide a mechanism through which all of the MPI defined functions, except those allowed as macros (See Section 2.6.4), may be accessed with a name shift. This requires, in C and Fortran, an alternate entry point name, with the prefix PMPI_ for each MPI function in each provided language binding and language support method. For routines implemented as macros, it is still required that the PMPI_ version be supplied and work as expected, but it is not possible to replace at link time the MPI_ version with a user-defined version.

For Fortran, the different support methods cause several specific procedure names. Therefore, several profiling routines (with these specific procedure names) are needed for each Fortran MPI routine, as described in Section 19.1.5.

- 2. ensure that those MPI functions that are not replaced may still be linked into an executable image without causing name clashes.
- 3. document the implementation of different language bindings of the MPI interface if
   they are layered on top of each other, so that the profiler developer knows whether
   she must implement the profile interface for each binding, or can economize by imple ⁴⁵
   ⁴⁶
   ⁴⁷
   ⁴⁷
   ⁴⁸

#### Unofficial Draft for Comment Only

4. where the implementation of different language bindings is done through a layered approach (e.g., the Fortran binding is a set of "wrapper" functions that call the C implementation), ensure that these wrapper functions are separable from the rest of the library.

This separability is necessary to allow a separate profiling library to be correctly implemented, since (at least with Unix linker semantics) the profiling library must contain these wrapper functions if it is to perform as expected. This requirement allows the person who builds the profiling library to extract these functions from the original MPI library and add them into the profiling library without bringing along any other unnecessary code.

11 12 13

15

1

2

3

4

5

6

7

8

9

10

5. provide a no-op routine MPI_PCONTROL in the MPI library.

¹⁴ 15.2.2 Discussion

The objective of the MPI profiling interface is to ensure that it is relatively easy for authors of profiling (and other similar) tools to interface their codes to MPI implementations on different machines.

Since MPI is a machine independent standard with many different implementations, it is unreasonable to expect that the authors of profiling tools for MPI will have access to the source code that implements MPI on any particular machine. It is therefore necessary to provide a mechanism by which the implementors of such tools can collect whatever performance information they wish *without* access to the underlying implementation.

We believe that having such an interface is important if MPI is to be attractive to end users, since the availability of many different tools will be a significant factor in attracting users to the MPI standard.

The profiling interface is just that, an interface. It says *nothing* about the way in which it is used. There is therefore no attempt to lay down what information is collected through the interface, or how the collected information is saved, filtered, or displayed.

While the initial impetus for the development of this interface arose from the desire to permit the implementation of profiling tools, it is clear that an interface like that specified may also prove useful for other purposes, such as "internetworking" multiple MPI implementations. Since all that is defined is an interface, there is no objection to it being used wherever it is useful.

As the issues being addressed here are intimately tied up with the way in which executable images are built, which may differ greatly on different machines, the examples given below should be treated solely as one way of implementing the objective of the MPI profiling interface. The actual requirements made of an implementation are those detailed in the Requirements section above, the whole of the rest of this section is only present as justification and discussion of the logic for those requirements.

The examples below show one way in which an implementation could be constructed to meet the requirements on a Unix system (there are doubtless others that would be equally valid).

44

⁴⁵ 15.2.3 Logic of the Design

⁴⁶ Provided that an MPI implementation meets the requirements above, it is possible for ⁴⁷ the implementor of the profiling system to intercept the MPI calls that are made by the user program. The profiling system implementor can then collect any required information before calling the underlying MPI implementation (through its name shifted entry points) to achieve the desired effects.

### 15.2.4 Miscellaneous Control of Profiling

There is a clear requirement for the user code to be able to control the profiler dynamically at run time. This capability is normally used for (at least) the purposes of

- Enabling and disabling profiling depending on the state of the calculation.
- Flushing trace buffers at non-critical 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 ⁴⁵ be enabled at the normal default level. (i.e., as if MPI_PCONTROL had just been called ⁴⁶ with the argument 1). This allows users to link with a profiling library and to obtain profile ⁴⁷ output without having to modify their source code at all. ⁴⁸

# Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

 $\frac{4}{5}$ 

6

7

8 9

10

11 12

13 14

15 16

17 18

19 20

21

22

23

24

25 26

27

28

29

30

31

32

33

34

35

36

37 38

39 40

41

42 43

The provision of MPI_PCONTROL as a no-op in the standard MPI library supports the collection of more detailed profiling information with source code that can still link against the standard MPI library.

**Example 15.1** A wrapper to accumulate the total amount of data sent by the MPI_SEND function, along with the total elapsed time spent in the function.

```
static int totalBytes = 0;
8
 static double totalTime = 0.0;
9
10
 int MPI_Send(const void* buffer, int count, MPI_Datatype datatype,
11
 int dest, int tag, MPI_Comm comm)
12
 {
13
 double tstart = MPI_Wtime();
 /* Pass on all arguments */
14
 int size;
15
 = PMPI_Send(buffer,count,datatype,dest,tag,comm);
 int result
16
17
 totalTime += MPI_Wtime() - tstart;
 /* and time
 */
18
19
 MPI_Type_size(datatype, &size); /* Compute size */
20
 totalBytes += count*size;
21
22
 return result;
23
 }
24
25
 MPI Library Implementation
26
 15.2.5
27
 If the MPI library is implemented in C on a Unix system, then there are various options,
28
 including the two presented here, for supporting the name-shift requirement. The choice
29
 between these two options depends partly on whether the linker and compiler support weak
30
 symbols.
^{31}
 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:

³⁴ Example 15.2 Library implementation using weak symbols.

```
36 #pragma weak MPI_Example = PMPI_Example
37
```

```
int PMPI_Example(/* appropriate args */)
```

/* Useful content */

40 41 {

}

38

39

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:

1

 $\mathbf{2}$ 

3

4

5

Example 15.3 Library implementation using C pre-processor macros.

```
#ifdef PROFILELIB
ifdef __STDC__
define FUNCTION(name) P##name
else
define FUNCTION(name) P/**/name
endif
#else
define FUNCTION(name) name
#endif
```

Each of the user visible functions in the library would then be declared thus

```
int FUNCTION(MPI_Example)(/* appropriate args */)
{
 /* Useful content */
}
```

The same source file can then be compiled to produce both versions of the library, depending on the state of the **PROFILELIB** macro symbol.

It is required that the standard MPI library be built in such a way that the inclusion of MPI functions can be achieved one at a time. This is a somewhat unpleasant requirement, since it may mean that each external function has to be compiled from a separate file. However this is necessary so that the author of the profiling library need only define those MPI functions that she wishes to intercept, references to any others being fulfilled by the normal MPI library. Therefore the link step can look something like this

# % cc ... -lmyprof -lpmpi -lmpi

Here libmyprof.a contains the profiler functions that intercept some of the MPI functions, libpmpi.a contains the "name shifted" MPI functions, and libmpi.a contains the normal definitions of the MPI functions.

# 15.2.6 Complications

# Multiple Counting

Since parts of the MPI library may themselves be implemented using more basic MPI func-tions (e.g., a portable implementation of the collective operations implemented using pointto-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).

# Unofficial Draft for Comment Only

 24 

# ¹ 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 7 achieved by using wrapper functions on top of the C implementation. The author of the 8 profile library then assumes that it is reasonable only to provide profile functions for the C 9 binding, since Fortran will eventually call these, and the cost of the wrappers is assumed 10 to be small. However, if the wrapper functions are not in the profiling library, then none 11 of the profiled entry points will be undefined when the profiling library is called. Therefore 12none of the profiling code will be included in the image. When the standard MPI library 13 is scanned, the Fortran wrappers will be resolved, and will also pull in the base versions of 14 the MPI functions. The overall effect is that the code will link successfully, but will not be 15profiled. 16

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**.

²² 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.

28 29

21

# 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

34 35 36

37

- 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.

⁴¹ Note, however, that it is possible to use the scheme above to implement a multi-level
⁴² system, since the function called by the user may call many different profiling functions
⁴³ before calling the underlying MPI function. This capability has been demonstrated in the
⁴⁴ P^NMPI tool infrastructure [58].

- 45
- 46
- 47
- 48

#### 15.3 The MPI Tool Information Interface

MPI implementations often use internal variables to control their operation and performance and rely on internal events for their implementation. Understanding and manipulating these variables and tracking these events can provide a more efficient execution environment or improve performance for many applications. This section describes the MPI tool information interface, which provides a mechanism for MPI implementors to expose variables, each of which represents a particular property, setting, or performance measurement from within the MPI implementation, as well as expose events that can be tracked by tools. The interface is split into three parts: the first part provides information about, and supports the setting of, control variables through which the MPI implementation tunes its configuration. The second part provides access to performance variables that can provide insight into internal performance information of the MPI implementation. The third part enables tools to query available events within an MPI implementation and register callbacks for them.

To avoid restrictions on the MPI implementation, the MPI tool information interface 15allows the implementation to specify which control variables, performance variables, and 16events exist. Additionally, the user of the MPI tool information interface can obtain meta-17 data about each available variable or event, such as its datatype, and a textual description. 18 The MPI tool information interface provides the necessary routines to find all variables and 19 events that exist in a particular MPI implementation; to query their properties; to retrieve 20descriptions about their meaning; to access and, if appropriate, to alter their values; and 21(in case of events) set callbacks triggered by them. 22

Variables, events, and categories across connected MPI processes with equivalent names 23 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 27processes in enumerations with the same meaning must also have the same meaning. In 28 order for variables and categories to have the same meaning, routines in the tools information 29interface that return details for those variables and categories have requirements on what 30 parameters must be identical. These requirements are specified in their respective sections. 31

Rationale. The intent of requiring the same meaning for entities with equivalent names is to enforce consistency across connected MPI processes. For example, variables describing the number of packets sent on different types of network devices should have different names to reflect their potentially different meanings. (End of rationale.)

The MPI tool information interface can be used independently from the MPI communication functionality. In particular, the routines of this interface can be called before MPI is initialized and after MPI is finalized. In order to support this behavior cleanly, the MPI tool information interface uses separate initialization and finalization routines. All identifiers used in the MPI tool information interface have the prefix MPI_T_.

On success, all MPI tool information interface routines return MPI_SUCCESS, otherwise they return an appropriate and unique return code indicating the reason why the call was not successfully completed. Details on return codes can be found in Section 15.3.10. However, unsuccessful calls to the MPI tool information interface are not fatal and do not impact the execution of subsequent MPI routines.

#### **Unofficial Draft for Comment Only**

1  $\mathbf{2}$ 

3

4

5

6

 $\overline{7}$ 

8

9

10

11

12

13

14

 24 

25

26

32

33

34

35

36

37 38

39

40

41

42

43

44

45

46

47

¹ Since the MPI tool information interface primarily focuses on tools and support li-² braries, MPI implementations are only required to provide C bindings for functions and ³ constants introduced in this section. Except where otherwise noted, all conventions and ⁴ principles governing the C bindings of the MPI API also apply to the MPI tool information ⁵ interface, which is available by including the mpi.h header file. All routines in this interface ⁶ have local semantics.

Advice to users. The number and type of control variables, performance variables, and events can vary between MPI implementations, platforms and different builds of the same implementation on the same platform as well as between runs. Hence, any application relying on a particular variable will not be portable. Further, there is no guarantee that the number of variables and variable indices are the same across connected MPI processes.

This interface is primarily intended for performance monitoring tools, support tools, and libraries controlling the application's environment. When maximum portability is desired, application programmers should either avoid using the MPI tool information interface or avoid being dependent on the existence of a particular control or performance variable or of a particular event. (*End of advice to users.*)

# 15.3.1 Verbosity Levels

The MPI tool information interface provides access to internal configuration and perfor-22 mance information through a set of control and performance variables defined by the MPI 23implementation. Since some implementations may export a large number of variables,  24 25variables are classified by a verbosity level that categorizes both their intended audience 26(end users, performance tuners or MPI implementors) and a relative measure of level of detail (basic, detailed or all). These verbosity levels are described by a single integer. 27Table 15.1 lists the constants for all possible verbosity levels. The values of the con-28stants are monotonic in the order listed in the table; i.e., MPI_T_VERBOSITY_USER_BASIC 29 < MPI_T_VERBOSITY_USER_DETAIL < ... < MPI_T_VERBOSITY_MPIDEV_ALL. 30

31		
32	MPI_T_VERBOSITY_USER_BASIC	Basic information of interest to users
33	MPI_T_VERBOSITY_USER_DETAIL	Detailed information of interest to users
34	MPI_T_VERBOSITY_USER_ALL	All remaining information of interest to users
35	MPI_T_VERBOSITY_TUNER_BASIC	Basic information required for tuning
36	MPI_T_VERBOSITY_TUNER_DETAIL	Detailed information required for tuning
37	MPI_T_VERBOSITY_TUNER_ALL	All remaining information required for tuning
38	MPI_T_VERBOSITY_MPIDEV_BASIC	Basic information for MPI implementors
39	MPI_T_VERBOSITY_MPIDEV_DETAIL	Detailed information for MPI implementors
40	MPI_T_VERBOSITY_MPIDEV_ALL	All remaining information for MPI implementors

- 41
- 42
- 43
- 44 45

46

Table 15.1: MPI tool information interface verbosity levels

# 15.3.2 Binding MPI Tool Information Interface Variables to MPI Objects

Each MPI tool information interface variable provides access to a particular control setting
 or performance property of the MPI implementation. A variable may refer to a specific

7

8

9

10

11

12

13 14

15

16

17

18

19 20

MPI object such as a communicator, datatype, or one-sided communication window, or the variable may refer more generally to the MPI environment of the process. Except for the last case, the variable must be bound to exactly one MPI object before it can be used. Table 15.2 lists all MPI object types to which an MPI tool information interface variable can be bound, together with the matching constant that MPI tool information interface routines return to identify the object type.

Constant	MPI object
MPI_T_BIND_NO_OBJECT	N/A; applies globally to entire MPI process
MPI_T_BIND_MPI_COMM	MPI communicators
MPI_T_BIND_MPI_DATATYPE	MPI datatypes
MPI_T_BIND_MPI_ERRHANDLER	MPI error handlers
MPI_T_BIND_MPI_FILE	MPI file handles
MPI_T_BIND_MPI_GROUP	MPI groups
MPI_T_BIND_MPI_OP	MPI reduction operators
MPI_T_BIND_MPI_REQUEST	MPI requests
MPI_T_BIND_MPI_WIN	MPI windows for one-sided communication
MPI_T_BIND_MPI_MESSAGE	MPI message object
MPI_T_BIND_MPI_INFO	MPI info object
MPI_T_BIND_MPI_SESSION	MPI session object

Table 15.2: Constants to identify associations of variables

*Rationale.* Some variables have meanings tied to a specific MPI object. Examples include the number of send or receive operations that use a particular datatype, the number of times a particular error handler has been called, or the communication protocol and "eager limit" used for a particular communicator. Creating a new MPI tool information interface variable for each MPI object would cause the number of variables to grow without bound, since they cannot be reused to avoid naming conflicts. By associating MPI tool information interface variables with a specific MPI object, the MPI implementation only must specify and maintain a single variable, which can then be applied to as many MPI objects of the respective type as created during the program's execution. (*End of rationale.*)

# 15.3.3 Convention for Returning Strings

Several MPI tool information interface functions return one or more strings. These functions have two arguments for each string to be returned: an OUT parameter that identifies a pointer to the buffer in which the string will be returned, and an INOUT parameter to pass the length of the buffer. The user is responsible for the memory allocation of the buffer and must pass the size of the buffer (n) as the length argument. Let n be the length value specified to the function. On return, the function writes at most n-1 of the string's characters into the buffer, followed by a null terminator. If the returned string's length is greater than or equal to n, the string will be truncated to n-1 characters. In this case, the length of the string plus one (for the terminating null character) is returned in the length argument. If the user passes the null pointer as the buffer argument or passes 0 as the length argument, the function does not return the string and only returns the length of the 

#### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

string plus one in the length argument. If the user passes the null pointer as the length argument, the buffer argument is ignored and nothing is returned.

³ MPI implementations behave as if they have an internal character array that is copied ⁴ to the output character array supplied by the user. Such output strings are only defined ⁵ to be equivalent if their notional source-internal character arrays are identical (up to and ⁶ including the null terminator), even if the output string is truncated due to a small input ⁷ length parameter n.

8 9

10

1

 $\mathbf{2}$ 

# 15.3.4 Initialization and Finalization

The MPI tool information interface requires a separate set of initialization and finalization
 routines.

 $13 \\ 14$ 

15 16 17

```
MPI_T_INIT_THREAD(required, provided)
```

6	IN	required	desired level of thread support (integer)
7	OUT	provided	provided level of thread support (integer)

18 19

20

21

42

# 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 22MPI tool information interface in the processes that will use the interface before calling 23any other of its routines. A user can initialize the MPI tool information interface by calling  24 MPI_T_INIT_THREAD, which can be called multiple times. In addition, this routine initial-25izes the thread environment for all routines in the MPI tool information interface. Calling 26this routine when the MPI tool information interface is already initialized has no effect 27beyond increasing the reference count of how often the interface has been initialized. The 28argument required is used to specify the desired level of thread support. The possible values 29 and their semantics are identical to the ones that can be used with MPI_INIT_THREAD 30 listed in Section 11.6. The call returns in provided information about the actual level of  31 thread support that will be provided by the MPI implementation for calls to MPI tool 32 information interface routines. It can be one of the four values listed in Section 11.6. 33

The MPI specification does not require all MPI processes to exist before MPI is 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.*)

 $47 \\ 48$ 

#### Unofficial Draft for Comment Only

1

2

3

4

5

6

 $\overline{7}$ 

12 13

14

15

16

17

18

19

20

21

22

23

 24 

25

26

27

28 29

30 31

32

33

34

35

36

37

38 39

40

41

42

43

44

45

1	MPI_INT
2	MPI_INT32_T
3	MPI_INT64_T
4	MPI_UNSIGNED
5	MPI_UNSIGNED_LONG
6	MPI_UNSIGNED_LONG_LONG
7	MPI_UINT32_T
8	MPI_UINT64_T
9	MPI_COUNT
10	MPI_CHAR
11	MPI_DOUBLE
12	
13	
14	Table 15.3: MPI datatypes that can be used by the MPI tool information interface
15	
16	The MPI tool information interface only relies on a subset of the basic MPI datatypes
17	and does not use any derived MPI datatypes. Table 15.3 lists all MPI datatypes that can
18	be returned by the MPI tool information interface to represent its variables.
19	The use of the datatype MPI_CHAR in the MPI tool information interface implies a null-
20	terminated character array, i.e., a string in the C language. If a variable has type MPI_CHAR,
21	the value of the count parameter returned by MPI_T_CVAR_HANDLE_ALLOC and
22	MPI_T_PVAR_HANDLE_ALLOC must be large enough to include any valid value, including
23	its terminating null character. The contents of returned MPI_CHAR arrays are only defined
24	from index 0 through the location of the first null character.
25	
26	Rationale. The MPI tool information interface requires a significantly simpler type
27	system than MPI itself. Therefore, only its required subset must be present before
28	MPI initialization and MPI implementations do not need to initialize the complete
29	MPI datatype system. (End of rationale.)
30	
31	For variables of type MPI_INT, an MPI implementation can provide additional informa-
32	tion by associating names with a fixed number of values. We refer to this information in
33	the following as an enumeration. In this case, the respective calls that provide additional
34	metadata for each control or performance variable, i.e., MPI_T_CVAR_GET_INFO (Sec-
35	tion 15.3.6), MPI_T_PVAR_GET_INFO (Section 15.3.7), and MPI_T_EVENT_GET_INFO
36	(Section 15.3.8), return a handle of type MPI_T_enum that can be passed to the follow-
37	ing functions to extract additional information. Thus, the MPI implementation can de-
38	scribe variables with a fixed set of values that each represents a particular state. Each
39	enumeration type can have $N$ different values, with a fixed $N$ that can be queried using
40	MPI_T_ENUM_GET_INFO.
41	
42	
43	
44	
45	
46	
47	
48	

IV		ow_GET_INTO(enumype, nu	in, name, name_ien)	
	IN	enumtype	enumeration to be queried (handle)	2
		enuntype	enumeration to be queried (nanule)	3
	OUT	num	number of discrete values represented by this	4
			enumeration (integer)	5
	OUT	name	buffer to return the string containing the name of the	6
			enumeration item (string)	7
	INOUT	name_len	length of the string and/or buffer for name (integer)	8
		hume_len	ing in or the string and or stiller for hame (integer)	9
				10

MPI_T_ENUM_GET_INFO(enumtype, num, name, name_len)

#### C binding

If enumtype is a valid enumeration, this routine returns the number of items represented by this enumeration type as well as its name. N must be greater than 0, i.e., the enumeration must represent at least one value.

The arguments name and name_len are used to return the name of the enumeration as described in Section 15.3.3.

The routine is required to return a name of at least length one. This name must be unique with respect to all other names for enumerations that the MPI implementation uses.

Names associated with individual values in each enumeration enumtype can be queried using MPI_T_ENUM_GET_ITEM.

MPI_T_ENUM_GET_ITEM	(enumtype, index,	value, na	me, name_len)
---------------------	-------------------	-----------	---------------

IN	enumtype	enumeration to be queried (handle)	26
IN	index		27
IIN	Index	number of the value to be queried in this enumeration (integer)	28
			29
OUT	value	variable value (integer)	30
OUT	name	buffer to return the string containing the name of the	31
		enumeration item (string)	32
INOUT	name_len	length of the string and/or buffer for name (integer)	33
INCOT	name_ien	rengen of the string and/or buller for hame (integer)	34

#### C binding

The arguments name and name_len are used to return the name of the enumeration item as described in Section 15.3.3.

If completed successfully, the routine returns the name/value pair that describes the enumeration at the specified index. The call is further required to return a name of at least length one. This name must be unique with respect to all other names of items for the same enumeration.

```
45
46
47
```

#### 15.3.6 **Control Variables**

2 The routines described in this section of the MPI tool information interface specification 3 focus on the ability to list, query, and possibly set control variables exposed by the MPI 4 implementation. These variables can typically be used by the user to fine tune properties 5and configuration settings of the MPI implementation. On many systems, such variables 6 can be set using environment variables, although other configuration mechanisms may be 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 9 limit," i.e., an upper bound on the size of messages sent or received using an eager protocol. 10

11 12

 24 

25

26

27

28 29 30

 31 32

33 34

35

36

37

7

8

# Control Variable Query Functions

13 An MPI implementation exports a set of N control variables through the MPI tool infor-14mation interface. If N is zero, then the MPI implementation does not export any control 15variables, otherwise the provided control variables are indexed from 0 to N-1. This index 16number is used in subsequent calls to identify the individual variables.

17An MPI implementation is allowed to increase the number of control variables during 18 the execution of an MPI application when new variables become available through dynamic 19loading. However, MPI implementations are not allowed to change the index of a control 20variable or to delete a variable once it has been added to the set. When a variable becomes 21inactive, e.g., through dynamic unloading, accessing its value should return a corresponding 22 error code. 23

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, ¹				
	desc_len, bind, scope)		- 3	
IN	cvar_index	index of the control variable to be queried, value between 0 and $num_cvar - 1$ (integer)	4	
OUT	name	buffer to return the string containing the name of the control variable (string)	6 7	
INOUT	name_len	length of the string and/or buffer for name (integer)	8	
OUT	verbosity	verbosity level of this variable (integer)	9	
	verbosity	verbosity level of this variable (integer)	10	
OUT	datatype	MPI datatype of the information stored in the control variable (handle)	11 12	
			12	
OUT	enumtype	optional descriptor for enumeration information (handle)	14	
OUT	desc	buffer to return the string containing a description of	15	
001	dese	the control variable (string)	16	
		( ),	17	
INOUT	desc_len	length of the string and/or buffer for $desc\xspace$ (integer)	18	
OUT	bind	type of MPI object to which this variable must be	19	
		bound (integer)	20	
OUT	scope	scope of when changes to this variable are possible	21	
001	scope	(integer)	22	
		(integer)	23	
			24	
C bindin			25	

# 

After a successful call to MPI_T_CVAR_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_CVAR_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 control 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 name must be unique with respect to all other names for control variables used by the MPI implementation.

The argument verbosity returns the verbosity level of the variable (see Section 15.3.1).

The argument datatype returns the MPI datatype that is used to represent the control 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.

The arguments desc and desc_len are used to return a description of the control variable 47 as described in Section 15.3.3. 48

#### **Unofficial Draft for Comment Only**

26

27

28

29 30

31

32 33

34

35

36 37

38 39

40 41

42

43 44

 $45 \\ 46$ 

Returning a description is optional. If an MPI implementation does not return a de scription, the first character for desc must be set to the null character and desc_len must
 be set to one at the return of this call.

⁴ The parameter bind returns the type of the MPI object to which the variable must be ⁵ bound or the value MPI_T_BIND_NO_OBJECT (see Section 15.3.2).

6 The scope of a variable determines whether changing a variable's value is either local 7to the MPI process or must be done by the user across multiple connected MPI processes. 8 The latter is further split into variables that require changes in a group of MPI processes 9 and those that require collective changes among all connected MPI processes. Both cases 10 can require variables on all participating MPI processes either to be set to consistent (but 11potentially different) values or to equal values. The description provided with the variable 12must contain an explanation about the requirements and/or restrictions for setting the 13particular variable.

On successful return from MPI_T_CVAR_GET_INFO, the argument scope will be set to
 one of the constants listed in Table 15.4.

¹⁶ If the name of a control variable is equivalent across connected MPI processes, the
 ¹⁷ following OUT parameters must be identical: verbosity, datatype, enumtype, bind, and scope.
 ¹⁸ The returned description must be equivalent.

19		
20	Scope Constant	Description
21	MPI_T_SCOPE_CONSTANT	read-only, value is constant
22	MPI_T_SCOPE_READONLY	read-only, cannot be written, but can change
23	MPI_T_SCOPE_LOCAL	may be writeable, writing is a local operation
24	MPI_T_SCOPE_GROUP	may be writeable, must be set to consistent values
25		across a group of connected MPI processes
26	MPI_T_SCOPE_GROUP_EQ	may be writeable, must be set to the same value
27		across a group of connected MPI processes
28	MPI_T_SCOPE_ALL	may be writeable, must be set to consistent values
29		across all connected MPI processes
30	MPI_T_SCOPE_ALL_EQ	may be writeable, must be set to the same value
31		across all connected MPI processes
32		
33	T-11.1	15 4. Comes for control consider
34	Table 1	15.4: Scopes for control variables
35		
36	Advice to users. The s	cope of a variable only indicates if a variable might
37		rantee that it can be changed at any time. (End of ad
38	to users.)	
9		
10		
11		
12 MF	PI_T_CVAR_GET_INDEX(nam	e, cvar_index)
¹³	N name	name of the control variable (string)
¹⁴	)UT cvar_index	
5		index of the control variable (integer)
16		
	binding	
₁₈ int	t MPI_T_cvar_get_index(co	nst 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 15.4 Querying and printing the names of all available control variables.

```
#include <stdio.h>
 20
 21
#include <stdlib.h>
 22
#include <mpi.h>
 23
 ^{24}
int main(int argc, char *argv[]) {
 25
 int i, err, num, namelen, bind, verbose, scope;
 26
 int threadsupport;
 27
 char name[100];
 28
 MPI_Datatype datatype;
 29
 err=MPI_T_init_thread(MPI_THREAD_SINGLE,&threadsupport);
 30
 31
 if (err!=MPI_SUCCESS)
 32
 return err;
 33
 34
 err=MPI_T_cvar_get_num(&num);
 35
 if (err!=MPI_SUCCESS)
 36
 return err;
 37
 for (i=0; i<num; i++) {</pre>
 38
 39
 namelen=100;
 40
 err=MPI_T_cvar_get_info(i, name, &namelen,
 41
 &verbose, &datatype, NULL,
 42
 NULL, NULL, /*no description */
 &bind, &scope);
 43
 44
 if (err!=MPI_SUCCESS && err!=MPI_T_ERR_INVALID_INDEX) return err;
 printf("Var %i: %s\n", i, name);
 45
 }
 46
 47
 48
 err=MPI_T_finalize();
```

1

 $\mathbf{2}$ 

3

4

5 6

7

8

9

10

11

12

13

14

15

16 17 18

1		!=MPI_SUCCESS)		
3	return 1; else			
4	return 0;			
5	}			
6	-			
7 8	Handle Allo	cation and Deallocation		
9 10		0	variable, a user must first allocate a handle of type ding it to an MPI object (see also Section $15.3.2$ ).	
11 12 13 14 15 16	befor partic	les used in the remaining parts $e$ MPI is initialized and after	MPI tool information interface are distinct from s of the MPI standard because they must be usable MPI is finalized. Further, accessing handles, in s, can be time critical and having a separate handle <i>of rationale.</i> )	
17				
18 19		AR HANDLE ALLOC(over in	dex, obj_handle, handle, count)	
20				
21	IN	cvar_index	index of control variable for which handle is to be allocated (index)	
22 23 24	IN	obj_handle	reference to a handle of the MPI object to which this variable is supposed to be bound (pointer)	
25	OUT	handle	allocated handle (handle)	
26 27 28	OUT	count	number of elements used to represent this variable (integer)	
29				
30	C binding		ar_index, void *obj_handle,	
31 32	IIIC MFI_I	MPI_T_cvar_handle *ha	C C	
33	This re	outine binds the control variable	le specified by the argument index to an MPI object.	
34			handle as an address to a local variable that stores	
35	•		handle is ignored if the MPI_T_CVAR_GET_INFO	
36			I_T_BIND_NO_OBJECT in the argument bind. The	
37			is returned in the argument handle. Upon success-	
38 39		AR_GET_INFO call) used to r	elements (of the datatype returned by a previous	
40		AR_GET_INFO can) used to r	epresent tills variable.	
41	Advid	te to users. The count can be	e different based on the MPI object to which the	
42	$\operatorname{contr}$	ol variable was bound. For e	xample, variables bound to communicators could	
43	have	a count that matches the size	of the communicator.	
44	It is a	not portable to pass reference	s to predefined MPI object handles, such as	
45			e, since their implementation depends on the $MPI$	
46			lles should be stored in a local variable and the	
47 48		ess of this local variable should of advice to users.)	d be passed into MPI_T_CVAR_HANDLE_ALLOC.	

is the number of available control van MPI_T_CVAR_GET_NUM. The type	in the range from 0 to $num_cvar - 1$ , where $num_cvar$ riables as determined from a prior call to of the MPI object it references must be consistent gument in a prior call to MPI_T_CVAR_GET_INFO.	1 2 3 4 5
		6
MPI_T_CVAR_HANDLE_FREE(handle	)	7
INOUT handle	handle to be freed (handle)	8
		9
C binding		10
int MPI_T_cvar_handle_free(MPI_T	C_cvar_handle *handle)	11 12
When a handle is no longer neede	d, a user of the MPI tool information interface should	13
_	free the handle and the associated resources in the	14
MPI implementation. On a successful	ul return, MPI sets the handle to	15
MPI_T_CVAR_HANDLE_NULL.		16
		17
Control Variable Access Functions		18
		19
		20 21
MPI_T_CVAR_READ(handle, buf)		21
IN handle	handle to the control variable to be read (handle)	23
OUT buf	initial address of storage location for variable value	24
	(choice)	25
		26
C binding		27
int MPI_T_cvar_read(MPI_T_cvar_h	andle handle, void *buf)	28 29
This routine queries the value of a	control variable identified by the argument handle and	29 30
stores the result in the buffer identified	by the parameter <b>buf</b> . The user must ensure that the	31
	the entire value of the control variable (based on the	32
	ior corresponding calls to MPI_T_CVAR_GET_INFO	33
and MPI_T_CVAR_HANDLE_ALLOC,	respectively).	34
		35
MPI_T_CVAR_WRITE(handle, buf)		36
IN handle	handle to the control variable to be written (handle)	37
IN buf		38 39
IN DUI	initial address of storage location for variable value (choice)	40
	(choice)	41
C binding		42
int MPI_T_cvar_write(MPI_T_cvar_	handle handle, const void *buf)	43
		44
	control variable identified by the argument handle to	45
	by the parameter <b>buf</b> . The user must ensure that the l the entire value of the control variable (based on the	46
Surfer is of the appropriate size to hold	the choice value of the control variable (based off the	47
		48

# Unofficial Draft for Comment Only

1	returned datatype and count from prior corresponding calls to MPI_T_CVAR_GET_INFO
2	and MPI_T_CVAR_HANDLE_ALLOC, respectively).
3	If the variable has a global scope (as returned by a prior corresponding
4	MPI_T_CVAR_GET_INFO call), any write call to this variable must be issued by the user
5	in all connected (as defined in Section 11.10.4) MPI processes. If the variable has group
6	scope, any write call to this variable must be issued by the user in all MPI processes in
7	the group, which must be described by the MPI implementation in the description by the
8	MPI_T_CVAR_GET_INFO.
9	In both cases, the user must ensure that the writes in all participating MPI processes
10	are consistent. If the scope is either MPI_T_SCOPE_ALL_EQ or MPI_T_SCOPE_GROUP_EQ
11	this means that the variable in all connected MPI processes or MPI processes of the group,
12	respectively, must be set to the same value.
13	If it is not possible to change the variable at the time the call is made, the function
14	returns either MPI_T_ERR_CVAR_SET_NOT_NOW, if there may be a later time at which the
15	variable could be set, or MPI_T_ERR_CVAR_SET_NEVER, if the variable cannot be set for the
16	remainder of the application's execution.
17	
18	<b>Example 15.5</b> Reading the value of a control variable.
19	<pre>int getValue_int_comm(int index, MPI_Comm comm, int *val) {</pre>
20	int err, count;
21	MPI_T_cvar_handle handle;
22	MFI_I_CVAL_HANGIE HANGIE,
23	/* This example assumes that the variable index */
24	/* can be bound to a communicator */
25	
26	err=MPI_T_cvar_handle_alloc(index, &comm, &handle, &count);
27	if (err!=MPI_SUCCESS) return err;
28	
29	/* The following assumes that the variable is $*/$
30	/* represented by a single integer */
31 32	,
33	<pre>err=MPI_T_cvar_read(handle,val);</pre>
34	if (err!=MPI_SUCCESS) return err;
35	
36	err=MPI_T_cvar_handle_free(&handle);
37	return err;
38	}
39	
40	15.3.7 Performance Variables
41	
42	The following section focuses on the ability to list and to query performance variables
43	provided by the MPI implementation. Performance variables provide insight into MPI
44	implementation-specific internals and can represent information such as the state of the

44 45 46

Rationale. The interface for performance variables is separate from the interface for control variables, since performance variables have different requirements and param-

for submodules, or queue sizes and lengths.

# Unofficial Draft for Comment Only

MPI implementation (e.g., waiting blocked, receiving, not active), aggregated timing data

eters. By keeping them separate, the interface provides cleaner semantics and allows for more performance optimization opportunities. (*End of rationale.*)

Some performance variables and classes refer to *events*. In general, such events describe state transitions within software or hardware related to the performance of an MPI application. The events offered through the callback-driven event-notification interface described in Section 15.3.8 also refer to such state transitions; however, the set of state transitions referred to by performance variables and events as described in Section 15.3.8 may not be identical.

#### Performance Variable Classes

Each performance variable is associated with a class that describes its basic semantics, possible datatypes, basic behavior, its starting value, whether it can overflow, and when and how an MPI implementation can change the variable's value. The starting value is the value that is assigned to the variable the first time that it is used or whenever it is reset.

Advice to users. If a performance variable belongs to a class that can overflow, it is up to the user to protect against this overflow, e.g., by frequently reading and resetting the variable value. (*End of advice to users.*)

Advice to implementors. MPI implementations should use large enough datatypes for each performance variable to avoid overflows under normal circumstances. (End of advice to implementors.)

The classes are defined by the following constants:

# • MPI_T_PVAR_CLASS_STATE

A performance variable in this class represents a set of discrete states. Variables of this class are represented by MPI_INT and can be set by the MPI implementation at any time. Variables of this type should be described further using an enumeration, as discussed in Section 15.3.5. The starting value is the current state of the implementation at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

#### MPI_T_PVAR_CLASS_LEVEL

A performance variable in this class represents a value that describes the utilization level of a resource. The value of a variable of this class can change at any time to match the current utilization level of the resource. Values returned from variables in this class are non-negative and represented by one of the following datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG, MPI_DOUBLE. The starting value is the current utilization level of the resource at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

• MPI_T_PVAR_CLASS_SIZE

A performance variable in this class represents a value that is the size of a resource. Values returned from variables in this class are non-negative and represented by one of the following datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG,

MPI_UNSIGNED_LONG_LONG, MPI_DOUBLE. The starting value is the current size of the resource at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow. 

#### Unofficial Draft for Comment Only

- 1 MPI_T_PVAR_CLASS_PERCENTAGE 2 3 4 56 7 8 implementations must ensure that variables of this class cannot overflow. 9 MPI_T_PVAR_CLASS_HIGHWATERMARK 10 11 1213 141516mentations must ensure that variables of this class cannot overflow. 17 MPI_T_PVAR_CLASS_LOWWATERMARK 19 202122 232425mentations must ensure that variables of this class cannot overflow. 26 MPI_T_PVAR_CLASS_COUNTER 2728 29 30 3132 33 of this class is 0. Variables of this class can overflow. 34 35MPI_T_PVAR_CLASS_AGGREGATE 36 The value of a performance variable in this class is an an aggregated value that 37 represents a sum of arguments processed during a specific event (e.g., the amount 38 of memory allocated by all memory allocations). This class is similar to the counter 39 class, but instead of counting individual events, the value can be incremented by 40 arbitrary amounts. The value of a variable of this class increases monotonically from 41 the initialization or reset of the performance variable. It must be non-negative and 42represented by one of the following datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG, 43 MPI_UNSIGNED_LONG_LONG, MPI_DOUBLE. The starting value for variables of this 44 class is 0. Variables of this class can overflow. 45
  - MPI_T_PVAR_CLASS_TIMER
  - The value of a performance variable in this class represents the aggregated time that the MPI implementation spends executing a particular event, type of event, or section

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

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 non-negative 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 imple-

18

46

47

48

A performance variable in this class represents a value that describes the low watermark utilization of a resource. The value of a variable of this class is non-negative and decreases 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 imple-

A performance variable in this class counts the number of occurrences of a specific event (e.g., the number of memory allocations within an MPI library). The value of a variable of this class increases monotonically from the initialization or reset of the performance variable by one for each specific event that is observed. Values must be non-negative and represented by one of the following datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG. The starting value for variables of the MPI library. This class has the same basic semantics as MPI_T_PVAR_CLASS_AGGREGATE, but explicitly records a timing value. The value of a variable of this class increases monotonically from the initialization or reset of the performance variable. It must be non-negative and represented by one of the following datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG, MPI_DOUBLE. The starting value for variables of this class is 0. If the type MPI_DOUBLE is used, the units that represent time in this datatype must match the units used by MPI_WTIME. Otherwise, the time units should be documented, e.g., in the description returned by MPI_T_PVAR_GET_INFO. Variables of this class can overflow.

• MPI_T_PVAR_CLASS_GENERIC

This class can be used to describe a variable that does not fit into any of the other classes. For variables in this class, the starting value is variable-specific and implementation-defined.

#### Performance Variable Query Functions

An MPI implementation exports a set of N performance variables through the MPI tool information interface. If N is zero, then the MPI implementation does not export any performance variables; otherwise the provided performance variables are indexed from 0 to N-1. This index number is used in subsequent calls to identify the individual variables.

An MPI implementation is allowed to increase the number of performance variables during the execution of an MPI application when new variables become available through dynamic loading. However, MPI implementations are not allowed to change the index of a performance variable or to delete a variable once it has been added to the set. When a variable becomes inactive, e.g., through dynamic unloading, accessing its value should return a corresponding error code.

The following function can be used to query the number of performance variables, num_pvar:

## MPI_T_PVAR_GET_NUM(num_pvar)

OUT num_pvar

returns number of performance variables (integer)

#### C binding

int MPI_T_pvar_get_num(int *num_pvar)

The function  $\mathsf{MPI_T_PVAR_GET_INFO}$  provides access to additional information for each variable.

 $\mathbf{2}$ 

 $\overline{7}$ 

 24 

1	MPI_T_P\	/AR_GET_INFO(pvar_index, na	ame, name_len, verbosity, var_class, datatype,
2		enumtype, desc, desc_len	, bind, readonly, continuous, atomic)
3 4 5	IN	pvar_index	index of the performance variable to be queried between 0 and $num_pvar - 1$ (integer)
6 7	OUT	name	buffer to return the string containing the name of the performance variable (string)
8	INOUT	name_len	length of the string and/or buffer for name (integer)
9	OUT	verbosity	verbosity level of this variable (integer)
10 11	OUT	var_class	class of performance variable (integer)
12 13	OUT	datatype	MPI datatype of the information stored in the performance variable (handle)
14 15	OUT	enumtype	optional descriptor for enumeration information (handle)
16 17 18	OUT	desc	buffer to return the string containing a description of the performance variable (string)
19	INOUT	desc_len	length of the string and/or buffer for desc (integer)
20 21 22	OUT	bind	type of MPI object to which this variable must be bound (integer)
22 23 24	OUT	readonly	flag indicating whether the variable can be written/reset (integer)
25 26	OUT	continuous	flag indicating whether the variable can be started and stopped or is continuously active (integer)
27 28 29	OUT	atomic	flag indicating whether the variable can be atomically read and reset (integer)
30	C binding	<u>r</u>	
31 32	•		index, char *name, int *name_len,
33			<pre>*var_class, MPI_Datatype *datatype,</pre>
34 35			, char *desc, int *desc_len, int *bind, continuous, int *atomic)
36 37 38 39	calls to thi information If any	s routine that query informat n. An MPI implementation is OUT parameter to MPI_T_PV	AR_GET_INFO for a particular variable, subsequent ion about the same variable must return the same not allowed to alter any of the returned values. /AR_GET_INFO is a NULL pointer, the implementa-
40 41 42	tion 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		
43 44 45	The a		verbosity level of the variable (see Section 15.3.1). le is returned in the parameter var_class. The class
46		ne of the constants defined in	-
47 48			he class of the performance variable must be unique rmance 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.

#### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 24 

Rationale. This routine is provided to enable fast retrieval of performance variables by a tool, assuming it knows the name of the variable for which it is looking. The number of variables exposed by the implementation can change over time, so it is not possible for the tool to simply iterate over the list of variables once at initialization. Although using MPI implementation specific variable names is not portable across MPI implementations, tool developers may choose to take this route for lower overhead at runtime because the tool will not have to iterate over the entire set of variables to find a specific one. (End of rationale.)

# ¹⁰ Performance Experiment Sessions

Within a single program, multiple components can use the MPI tool information interface. To avoid collisions with respect to accesses to performance variables, users of the MPI tool information interface must first create a performance experiment session. Subsequent calls that access performance variables can then be made within the context of this performance experiment session. Starting, stopping, reading, writing, or resetting a variable in one performance experiment session shall not influence whether a variable is started, stopped, read, written, or reset in another performance experiment session.

19 20

21

22 23

25 26

27

28

29 30 31

32

33 34

36 37

38

39

# MPI_T_PVAR_SESSION_CREATE(pe_session)

OUT pe_session identifier of performance experiment session (handle)

### ²⁴ 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 pe_session of type MPI_T_pvar_session.

# MPI_T_PVAR_SESSION_FREE(pe_session)

INOUT pe_session

identifier of performance experiment session (handle)

# 35 C binding

int MPI_T_pvar_session_free(MPI_T_pvar_session *pe_session)

This call frees an existing performance experiment session. Calls to the MPI tool information interface can no longer be made within the context of a performance experiment session after it is freed. On a successful return, MPI sets the performance experiment session identifier to MPI_T_PVAR_SESSION_NULL.

40 41 42

43

Handle Allocation and Deallocation

Before using a performance variable, a user must first allocate a handle of type MPI_T_pvar_handle for the variable by binding it to an MPI object (see also Section 15.3.2).

1

2

3

4

5

6

7

8

IN	pe_session	identifier of performance experiment session (handle)
IN	pvar_index	index of performance variable for which handle is to be allocated (integer)
IN	obj_handle	reference to a handle of the MPI object to which this variable is supposed to be bound (pointer)
OUT	handle	allocated handle (handle)
OUT	count	number of elements used to represent this variable (integer)

MPI_T_PVAR_HANDLE_ALLOC(pe_session, pvar_index, obj_handle, handle, count)

### C binding

# 

This routine binds the performance variable specified by the argument index to an MPI object in the performance experiment session identified by the parameter pe_session. The object is passed in the argument obj_handle as an address to a local variable that stores the object's handle. The argument obj_handle is ignored if the MPI_T_PVAR_GET_INFO call for this performance variable returned MPI_T_BIND_NO_OBJECT in the argument bind. The handle allocated to reference the variable is returned in the argument handle. Upon successful return, count contains the number of elements (of the datatype returned by a previous MPI_T_PVAR_GET_INFO call) used to represent this variable.

Advice to users. The count can be different based on the MPI object to which the performance variable was bound. For example, variables bound to communicators could have a count that matches the size of the communicator.

It is not portable to pass references to predefined MPI object handles, such as MPI_COMM_WORLD, to this routine, since their implementation depends on the MPI library. Instead, such an object handle should be stored in a local variable and the address of this local variable should be passed into MPI_T_PVAR_HANDLE_ALLOC. (*End of advice to users.*)

The value of index should be in the range from 0 to  $num_pvar - 1$ , where  $num_pvar$  is the number of available performance variables as determined from a prior call to MPI_T_PVAR_GET_NUM. The type of the MPI object it references must be consistent with the type returned in the bind argument in a prior call to MPI_T_PVAR_GET_INFO.

For all routines in the rest of this section that take both handle and pe_session as IN or INOUT arguments, if the handle argument passed in is not associated with the pe_session argument, MPI_T_ERR_INVALID_HANDLE is returned.

MPI_T_PVAR_HANDLE_FREE(pe_session, handle)

IN	pe_session	identifier of performance experiment session (handle)	44
INOUT	handle	handle to be freed (handle)	45

C binding

int MPI_T_pvar_handle_free(MPI_T_pvar_session pe_session,

#### Unofficial Draft for Comment Only

 24 

1		MPI_T_pvar_h	andle *ha	andle)
2 3 4 5 6	call MPI_ identified	T_PVAR_HANDLE_F	REE to free e_session an	a user of the MPI tool information interface should e the handle in the performance experiment session and the associated resources in the MPI implemen- the handle to MPI_T_PVAR_HANDLE_NULL.
7 8	Starting a	and Stopping of Perfor	mance Vari	ables
9 10 11 12 13 14	continuou any time stopped s	usly operating once a , but they cannot be	handle has started or le has been	ntinuous flag set during the query operation are s been allocated. Such variables may be queried at stopped by the user. All other variables are in a allocated; their values are not updated until they
15 16	MPI_T_F	VAR_START(pe_sess	ion, handle	
17	IN	pe_session		identifier of performance experiment session (handle)
18 19	IN	handle		handle of a performance variable (handle)
21 22 23	C bindini int MPI	ng _T_pvar_start(MPI_ MPI_T_pvar_h	-	
24 25 26 27 28 29 30 31 32 33	eter hand If th tion atte by the pa tine retu non-cont returned.	lle in the performance e constant MPI_T_PV mpts to start all var arameter pe_session f rns MPI_SUCCESS if inuous variables to b	e experimen /AR_ALL_H/ iables with or which h all variabl be started) es and var	e variable with the handle identified by the param- nt session identified by the parameter pe_session. ANDLES is passed in handle, the MPI implementa- in the performance experiment session identified andles have been allocated. In this case, the rou- es are started successfully (even if there are no , otherwise MPI_T_ERR_PVAR_NO_STARTSTOP is iables that are already started are ignored when
34				
35 36		VAR_STOP(pe_session	on, nandle)	
37	IN	pe_session		identifier of performance experiment session (handle)
38 39	IN	handle		handle of a performance variable (handle)
40	C bindi	ng		
41 42		_T_pvar_stop(MPI_T MPI_T_pvar_h	-	-
43 44 45 46 47 48	eter hand If th	lle in the performance e constant $MPI_T_N$	e experimer /AR_ALL_H/	e variable with the handle identified by the param- nt session identified by the parameter pe_session. ANDLES is passed in handle, the MPI implementa- in the performance experiment session identified

CHAPTER 15. TOOL SUPPORT

by the parameter pe_session for which handles have been allocated. In this case, the routine returns MPI_SUCCESS if all variables are stopped successfully (even if there are no non-continuous variables to be stopped), otherwise MPI_T_ERR_PVAR_NO_STARTSTOP is returned. Continuous variables and variables that are already stopped are ignored when MPI_T_PVAR_ALL_HANDLES is specified.

#### Performance Variable Access Functions

MPI_T_PVAR_READ(pe_session, handle, buf)

IN	pe_session	identifier of performance experiment session (handle)
IN	handle	handle of a performance variable (handle)
OUT	buf	initial address of storage location for variable value
		(choice)

#### C binding

The MPI_T_PVAR_READ call queries the value of the performance variable with the handle handle in the performance experiment session identified by the parameter pe_session and stores the result in the buffer identified by the parameter buf. The user is responsible to ensure that the buffer is of the appropriate size to hold the entire value of the performance variable (based on the datatype and count returned by the corresponding previous calls to MPI_T_PVAR_GET_INFO and MPI_T_PVAR_HANDLE_ALLOC, respectively).

The constant MPI_T_PVAR_ALL_HANDLES cannot be used as an argument for the function MPI_T_PVAR_READ.

		30
MPI_T_PVAR_WRITE(pe_session, handle	, buf)	31
IN pe_session	identifier of performance experiment session (handle)	32
IN handle	handle of a performance variable (handle)	33
		34
IN buf	initial address of storage location for variable value	35
	(choice)	36
		37
C binding		38
<pre>int MPI_T_pvar_write(MPI_T_pvar_se</pre>	ssion pe_session,	39

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).

#### Unofficial Draft for Comment Only

 24 

1	If it	is not possible to ch	ange the variable, the function returns
2		RR_PVAR_NO_WRITE.	
3			$_ALL_HANDLES$ cannot be used as an argument for the func-
4	tion MPI	_T_PVAR_WRITE.	
5			
6 7	MPI_T_F	VAR_RESET(pe_session	on, handle)
8	IN	pe_session	identifier of performance experiment session (handle)
9 10	IN	handle	handle of a performance variable (handle)
11	C bindi	na	
12		-	_pvar_session pe_session,
13 14	1110 111 1 <u>-</u>	MPI_T_pvar_ha	
15	The	MPI_T_PVAR_RESET	call sets the performance variable with the handle identified
16	by the pa	arameter handle to its s	starting value specified in Section 15.3.7. If it is not possible
17	to change	e the variable, the fund	tion returns MPI_T_ERR_PVAR_NO_WRITE.
18			<code>R_ALL_HANDLES</code> is passed in handle, the MPI implementation
19	-		within the performance experiment session identified by the
20	-		handles have been allocated. In this case, the routine returns
21 22			e reset successfully (even if there are no valid handles or all
23			ERR_PVAR_NO_WRITE is returned. Read-only variables are
24	ignored v	vnen MPI_I_PVAR_ALL	_HANDLES is specified.
25			
26	MPI_T_F	VAR_READRESET(pe	_session, handle, buf)
27 28	IN	pe_session	identifier of performance experiment session (handle)
29	IN	handle	handle of a performance variable (handle)
30 31	OUT	buf	initial address of storage location for variable value (choice)
32			
33	C bindi	ng	
34	int MPI_	T_pvar_readreset(M	PI_T_pvar_session pe_session,
35		MPI_T_pvar_ha	undle handle, void *buf)
36	This	call atomically combi	ines the functionality of MPI_T_PVAR_READ and
37		-	same semantics as if these two calls were called separately.
38			ariable are not supported, this routine returns
39		RR_PVAR_NO_ATOMIC.	
40			_ALL_HANDLES cannot be used as an argument for the func-
41 42	tion MPI	_T_PVAR_READRESE	T.
43	Adt	vice to implementors.	Sampling-based tools rely on the ability to call the $MPI$ tool
44			particular routines to start, stop, read, write, and reset per-
45			any program context, including asynchronous contexts such
46		-	mplementations should strive, if possible in their particular
47		,	nese usage scenarios for all or a subset of the routines men-
48	tior	ned above. If implement	nting only a subset, the read, write, and reset routines are

# Unofficial Draft for Comment Only

typically the most critical for sampling based tools. An MPI implementation should clearly document any restrictions on the program contexts in which the MPI tool information interface can be used. Restrictions might include guaranteeing usage outside of all signals or outside a specific set of signals. Any restrictions could be documented, for example, through the description returned by MPI_T_PVAR_GET_INFO. (*End of advice to implementors.*)

*Rationale.* All routines to read, to write or to reset performance variables require the performance experiement session argument. This requirement keeps the interface consistent and allows the use of MPI_T_PVAR_ALL_HANDLES where appropriate. Further, this opens up additional performance optimizations for the implementation of handles. (*End of rationale.*)

**Example 15.6** Detecting Receives with long unexpected message queues.

The following example shows a sample tool to identify receive operations that occur during times with long message queues. This examples assumes that the MPI implementation exports a variable with the name "MPI_T_UMQ_LENGTH" to represent the current length of the unexpected message queue. The tool is implemented as a PMPI tool using the MPI profiling interface.

The tool consists of three parts: (1) the initialization (by intercepting the call to MPI_INIT), (2) the test for long unexpected message queues (by intercepting calls to MPI_RECV), and (3) the clean-up phase (by intercepting the call to MPI_FINALIZE). To capture all receives, the example would have to be extended to have similar wrappers for all receive operations.

Part 1—Initialization: During initialization, the tool searches for the variable and, once the right index is found, allocates a performance experiment session and a handle for the variable with the found index, and starts the performance variable.

	30
#include <stdio.h></stdio.h>	31
#include <stdlib.h></stdlib.h>	32
<pre>#include <string.h></string.h></pre>	33
<pre>#include <assert.h></assert.h></pre>	34
#include <mpi.h></mpi.h>	35
	36
/* Global variables for the tool */	37
<pre>static MPI_T_pvar_session pe_session;</pre>	38
<pre>static MPI_T_pvar_handle handle;</pre>	39
*	40
int MPI_Init(int *argc, char ***argv ) {	41
<pre>int err, num, i, index, namelen, verbosity;</pre>	42
<pre>int var_class, bind, threadsup;</pre>	43
int readonly, continuous, atomic, count;	44
char name[18];	45
MPI_Comm comm;	46
MPI_Datatype datatype;	47
MPI_T_enum enumtype;	48

 $\mathbf{5}$ 

```
1
2
 err=PMPI_Init(argc, argv);
3
 if (err!=MPI_SUCCESS) return err;
4
5
 err=PMPI_T_init_thread(MPI_THREAD_SINGLE, &threadsup);
6
 if (err!=MPI_SUCCESS) return err;
7
8
 err=PMPI_T_pvar_get_num(&num);
9
 if (err!=MPI_SUCCESS) return err;
10
 index=-1;
11
 i=0;
12
 while ((i<num) && (index<0) && (err==MPI_SUCCESS)) {</pre>
13
 /* Pass a buffer that is at least one character longer than */
14
 /* the name of the variable being searched for to avoid */
15
 /* finding variables that have a name that has a prefix */
16
 /* equal to the name of the variable being searched. */
17
 namelen=18;
18
 err=PMPI_T_pvar_get_info(i, name, &namelen, &verbosity,
19
 &var_class, &datatype, &enumtype, NULL, NULL, &bind,
20
 &readonly, &continuous, &atomic);
21
 if (strcmp(name,"MPI_T_UMQ_LENGTH")==0) index=i;
22
 i++; }
23
 if (err!=MPI_SUCCESS) return err;
24
25
 /* this could be handled in a more flexible way for a generic tool */
26
 assert(index>=0);
27
 assert(var_class==MPI_T_PVAR_CLASS_LEVEL);
28
 assert(datatype==MPI_INT);
 assert(bind==MPI_T_BIND_MPI_COMM);
29
30
31
 /* Create a session */
32
 err=PMPI_T_pvar_session_create(&pe_session);
33
 if (err!=MPI_SUCCESS) return err;
34
35
 /* Get a handle and bind to MPI_COMM_WORLD */
36
 comm=MPI_COMM_WORLD;
37
 err=PMPI_T_pvar_handle_alloc(pe_session, index, &comm, &handle, &count);
38
 if (err!=MPI_SUCCESS) return err;
39
40
 /* this could be handled in a more flexible way for a generic tool */
41
 assert(count==1);
42
43
 /* Start variable */
44
 err=PMPI_T_pvar_start(pe_session, handle);
45
 if (err!=MPI_SUCCESS) return err;
46
47
 return MPI_SUCCESS;
48
 }
```

Part 2—Testing the Queue Lengths During Receives: During every receive operation, the tool reads the unexpected queue length through the matching performance variable and compares it against a predefined threshold.

}

Part 3—Termination: In the wrapper for MPI_FINALIZE, the MPI tool information interface is finalized.

```
int MPI_Finalize(void)
{
 int err;
 err=PMPI_T_pvar_handle_free(pe_session, &handle);
 err=PMPI_T_pvar_session_free(&pe_session);
 err=PMPI_T_finalize();
 return PMPI_Finalize();
}
```

# 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.

# Unofficial Draft for Comment Only

 $\mathbf{2}$ 

2 3 4 5		as to manage multiple state of		
6 7 8 9 10 11	Interface u logical entity part of the ordering of determined	e software and hardware system ses the concept of <i>sources</i> . A ity raising the event. A sour software or hardware system. If events across different compo- l or is too costly to enforce.	changes to be observed concurrently by different m, the event interface of the MPI Tool Information source in this context is a concept describing the rece may or may not directly represent a concrete This concept is used primarily to describe partia onents where total ordering cannot necessarily be o query the number of event sources, <i>num_sources</i>	n e d e
12 13	MPI_T_SO	URCE_GET_NUM(num_source	es)	
14	OUT	num_sources	returns number of event sources (integer)	
15 16 17	C binding int MPI_T	g _source_get_num(int *num_	_sources)	
19 20 21 22 23 24	MPI_T_SO sources du allowed to made visib	URCE_GET_NUM. An MPI in ring the execution of an MPI change the index of an event s	rces can be queried with a call to aplementation is allowed to increase the number of process. However, MPI implementations are not source or to delete an event source once it has been at sources become available via dynamic loading of mentation).	n n
26 27	MPI_T_SO	URCE_GET_INFO(source_inde ticks_per_second, max_tic	ex, name, name_len, desc, desc_len, ordering, cks, info)	
28 29 30	IN	source_index	index of the source to be queried between 0 and $num_sources - 1$ (integer)	
31 32	OUT	name	buffer to return the string containing the name of the source (string)	e
33	INOUT	name_len	length of the string and/or buffer for $name\xspace$ (integer)	
35	OUT	desc	buffer to return the string containing the description of the source (string)	1
37	INOUT	desc_len	length of the string and/or buffer for $desc$ (integer)	
38 39	OUT	ordering	flag indicating chronological ordering guarantees given by the source (integer)	
40 41	OUT	ticks_per_second	the number of ticks per second for the timer of this source (integer)	
43	OUT	max_ticks	the maximum count of ticks reported by this source before overflow occurs (integer)	
45 46	OUT	info	optional info object (handle)	
47 48	C binding	-	rea index char thank int theme las	
<ol> <li>13</li> <li>14</li> <li>15</li> <li>16</li> <li>17</li> <li>18</li> <li>19</li> <li>20</li> <li>21</li> <li>22</li> <li>23</li> <li>24</li> <li>25</li> <li>26</li> <li>27</li> <li>28</li> <li>29</li> <li>30</li> <li>31</li> <li>32</li> <li>33</li> <li>34</li> <li>35</li> <li>36</li> <li>37</li> <li>38</li> <li>39</li> <li>40</li> <li>41</li> <li>42</li> <li>43</li> <li>44</li> <li>45</li> <li>46</li> </ol>	OUT C binding int MPI_T The n MPI_T_SO sources du allowed to made visib additional MPI_T_SO IN OUT INOUT OUT INOUT OUT OUT OUT OUT OUT	<pre>num_sources g _source_get_num(int *num_ umber of available event sour URCE_GET_NUM. An MPI in ring the execution of an MPI change the index of an event s le to the user (e.g., if new ever components in the MPI imple URCE_GET_INFO(source_indee     ticks_per_second, max_tic     source_index     name     name_len     desc     desc_len     ordering     ticks_per_second     max_ticks     info g</pre>	returns number of event sources (integer) <b>sources</b> ) rces can be queried with a call to applementation is allowed to increase the number process. However, MPI implementations are n source or to delete an event source once it has be at sources become available via dynamic loading mentation). ex, name, name_len, desc, desc_len, ordering, cks, info) index of the source to be queried between 0 and num_sources – 1 (integer) buffer to return the string containing the name of t source (string) length of the string and/or buffer for name (integer buffer to return the string containing the description of the source (string) length of the string and/or buffer for desc (integer) flag indicating chronological ordering guarantees given by the source (integer) the number of ticks per second for the timer of this source (integer) the maximum count of ticks reported by this source before overflow occurs (integer)	lo en o hh h : )) or

char *desc, int *desc_len, MPI_T_source_order *ordering, MPI_Count *ticks_per_second, MPI_Count *max_ticks, MPI_Info *info)

A call to MPI_T_SOURCE_GET_INFO returns additional information on the source identified by the source_index argument.

The arguments name and name_len are used to return the name of the source as described in Section 15.3.3.

The arguments desc and desc_len are used to return the description of the source as described in Section 15.3.3.

The ordering argument returns whether event callbacks of this source will be invoked in chronological order, i.e., the timestamps reported by MPI_T_EVENT_GET_TIMESTAMP of subsequent events of the same source are monotonically increasing. The value of ordering can be MPI_T_SOURCE_ORDERED or MPI_T_SOURCE_UNORDERED.

The ticks_per_seconds argument returns the number of ticks elapsed in one second for the timer used for the specific source.

The max_ticks argument returns the largest number of ticks reported by this source as a timestamp before the value overflows.

Advice to users. As the size of MPI_Count is defined in relation to the types MPI_Aint and MPI_Offset, the effective size of MPI_Count may lead to overflows of the timestamp values reported. Users can use the argument max_ticks to mitigate resulting problems. (*End of advice to users.*)

MPI can optionally return an info object containing the default hints set for this source. If the argument to info provided by the user is the NULL pointer, this argument is ignored, otherwise an MPI implementation is required to return all hints that are supported by the implementation for this source 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 via MPI_INFO_FREE.

MPI_T_S	OURCE_GET_TIME	ESTAMP(source_index, timestamp)	34
IN	source_index	index of the source (integer)	35
			36
OUT	timestamp	current timestamp from specified source (integer)	37

#### 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

 $\mathbf{2}$ 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  $\mathbf{5}$ safety levels. 

7	
8	Safety Requirement
9	MPI_T_CB_REQUIRE_NONE
10	MPI_T_CB_REQUIRE_MPI_RESTRICTED
11	MPI_T_CB_REQUIRE_THREAD_SAFE
12	MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE
13	
14	Table 15.5: Hierarchy of safety requirement levels for event callback routines
15	
16	Table 15.5 provides the hierarchy of callback safety requirements levels within user-
17	defined callback functions. The MPI implementation provides the safety requirement as an
18	argument to the callback when it is invoked.
19	The level of MPI_T_CB_REQUIRE_NONE is the lowest level and does not impose any
20	restrictions on the callback function.
21	The level of MPI_T_CB_REQUIRE_MPI_RESTRICTED restricts the set of MPI functions
22	that can be called from inside the callback to all functions with the prefix MPI_T as well as
23	MPI_WTICK and MPI_WTIME.
24	
25	Advice to users. While some MPI functions are safe to be called inside a callback
26	function used in the MPI tool information interface—which may in some implemen-
27	tations be issued from asynchronous contexts such as signal handlers—this does not
28	imply that those MPI functions are generally safe to be called in asynchronous contexts
29	such as signal handlers. (End of advice to users.)
30	
31	The level of MPI_T_CB_REQUIRE_THREAD_SAFE includes all the limitations of
32	MPI_T_CB_REQUIRE_MPI_RESTRICTED and additionally requires the callback to be reen-
33	trant and thread-safe. This means the callback must allow its execution to be interrupted
34	by or happen concurrently with any other callback including itself.
35	The level of MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE includes all the limitations of
36	MPI_T_CB_REQUIRE_THREAD_SAFE and additionally requires the callback to meet the safety
37	requirements needed to support invocations from asynchronous contexts, such as signal
38	handlers.
39	Advice to users. It is always safe to assume the highest restrictions for a callback
40	
41	invocation (i.e., MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE). By evaluating the spe-
42	cific requirements at runtime, a tool may obtain more freedom of action within the
43	callback. (End of advice to users.)
44	Advice to implementors. A high-quality implementation will strive to set callback
45	safety requirements to the most permissive level for a given callback invocation. ( <i>End</i>
46	of advice to implementors.)
47	

MPI_T_EVENT_COPY	PMPI_T_EVENT_COPY	1
MPI_T_EVENT_GET_SOURCE	PMPI_T_EVENT_GET_SOURCE	2
MPI_T_EVENT_GET_TIMESTAMP	PMPI_T_EVENT_GET_TIMESTAMP	3
MPI_T_EVENT_READ	PMPI_T_EVENT_READ	4
MPI_T_PVAR_READ	PMPI_T_PVAR_READ	5
MPI_T_PVAR_READRESET	PMPI_T_PVAR_READRESET	6
MPI_T_PVAR_RESET	PMPI_T_PVAR_RESET	7
MPI_T_PVAR_START	PMPI_T_PVAR_START	8
MPI_T_PVAR_STOP	PMPI_T_PVAR_STOP	9
MPI_T_PVAR_WRITE	PMPI_T_PVAR_WRITE	10
MPI_T_SOURCE_GET_TIMESTAMP	PMPI_T_SOURCE_GET_TIMESTAMP	11

Table 15.6: List of MPI functions that when called from within a callback function may not return MPI_T_ERR_NOT_ACCESSIBLE

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.

A call may be implemented in a way that is not safe for all execution Rationale. 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*:

48

12

13

141516

17

18

19 20

21

22

23

2425

26

27

2829

30

31

32

33 34

3536

37

38

39

40

41

42

43

⁴⁵ 4647

	754		CHAPTER 15. TOOL SUPPORT
1	MPI_T_EVENT_GET_NUM(num_events)		
2 3	OUT	num_events	returns number of event types (integer)
4 5 6	C binding int MPI_T_event_get_num(int *num_events)		
7 8 9		function MPI_T_EVENT_GET pecific event type.	_INFO provides access to additional information
10 11 12 13	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)		
14 15	IN	event_index	index of the event type to be queried between 0 and num_events $-1$ (integer)
16 17 18	OUT	name	buffer to return the string containing the name of the event type (string)
19	INOUT	name_len	length of the string and/or buffer for name (integer)
20	OUT	verbosity	verbosity level of this event type (integer)
21 22 23	OUT	array_of_datatypes	array of MPI basic data types used to encode the event data (array of handles)
24 25	OUT	array_of_displacements	array of byte displacements of the elements in the event buffer (array of non-negative integers)
26 27 28	INOUT	num_elements	length of array_of_datatypes and array_of_displacements arrays (non-negative integer)
29 30	OUT	enumtype	optional descriptor for enumeration information (handle)
31	OUT	info	optional info object (handle)
32 33	Ουτ	desc	buffer to return the string containing a description of the event type (string)
$\frac{34}{35}$	INOUT	desc_len	length of the string and/or buffer for $desc$ (integer)
36	OUT	bind	type of $MPI$ object to which an event of this type
37			must be bound (integer)
38 39	C hindin		
40	C bindin		t index char *name int *name len
41	<pre>int MPI_T_event_get_info(int event_index, char *name, int *name_len,</pre>		
42	<ul> <li>MPI_Aint array_of_displacements[], int *num_elements,</li> <li>MPI_T_enum *enumtype, MPI_Info *info, char *desc,</li> <li>int *desc_len, int *bind)</li> </ul>		
44 45			
After a successful call to MPI_T_EVENT_GET_INFO for			· · · · · · · · · · · · · · · · · · ·
<ul> <li>sequent calls to this routine that query information about the same event type in the same information. If any INOUT or OUT argument to MPI_T_EVENT_GET</li> </ul>			
48	une same i	mormanon. It any moor of	

NULL pointer, the implementation will ignore the argument and not return a value for the specific argument.

The arguments name and name_len are used to return the name of the event type as described in Section 15.3.3. If completed successfully, the routine is required to return a name of at least length one. The name of the event type must be unique with respect to all other names for event types used by the MPI implementation.

The argument verbosity returns the verbosity level of the event type (see Section 15.3.1).

The argument array_of_datatypes returns an array of MPI datatype handles that describe the elements returned for an instance of the event type with index event_index. The event data can either be queried element by element with MPI_T_EVENT_READ or copied into a contiguous event buffer with MPI_T_EVENT_COPY. For the latter case, the argument array_of_displacements returns an array of byte displacements in the event buffer in ascending order starting with zero.

The user is responsible for the memory allocation for the array_of_datatypes and array_of_displacements arrays. The number of elements in each array is supplied by the user in num_elements. If the number of elements used by the event type is larger than the value of num_elements provided by the user, the number of datatype handles and displacements returned in the corresponding arrays is truncated to the value of num_elements passed in by the user. If the user passes the NULL pointer for array_of_datatypes or array_of_displacements, the respective arguments are ignored. Unless the user passes the NULL pointer for num_elements, the function returns the number of elements required for this event type. If the user passes the NULL pointer for num_elements, the arguments num_elements, array_of_datatypes, and array_of_displacements are ignored.

MPI can optionally return an enumeration identifier in the enumtype argument, describing the individual elements in the array_of_datatypes argument. Otherwise, enumtype is set to MPI_T_ENUM_NULL. If the argument to enumtype provided by the user is the MPI_T_ENUM_NULL pointer, no enumeration type is returned.

28MPI can optionally return an info object containing the default hints set for a registration handle for this event type. If the argument to info provided by the user is the NULL 2930 pointer, this argument is ignored, otherwise an MPI implementation is required to return  31 all hints that are supported by the implementation for a registration handle for this event type and have default values specified; any user-supplied hints that were not ignored by the 33 implementation; and any additional hints that were set by the implementation. If no such 34hints exist, a handle to a newly created info object is returned that contains no key/value pair. The user is responsible for freeing info via 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.

4546This routine returns MPI_SUCCESS on success and returns MPI_T_ERR_INVALID_INDEX 47if event_index does not match a valid event type index provided by the implementation at 48 the time of the call.

### **Unofficial Draft for Comment Only**

1

2

3

4

 $\mathbf{5}$ 

6  $\overline{7}$ 

8 9

10 11

12

13

14

15

16

17

18

19

20

21

22

23 24 

25

26

27

32

35

36 37

> 38 39

> 40 41

> 42

1	MPI_T_EV	/ENT_GET_INDEX(name, ever	nt_index)		
2	IN	name	name of the event type (string)		
3 4	OUT	event_index	index of the event type (integer)		
5					
6	C binding	5			
7	int MPI_T	_event_get_index(const ch	nar *name, int *event_index)		
8 9 10 11	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.				
12 13 14 15	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.				
16 17 18 19 20 21 22 23 23 24	a too numl is no initia porta for lo	ol, assuming it knows the nar per of event types exposed by ot possible for the tool to sin dization. Although using MP able across MPI implementation	ided to enable fast retrieval of an event index by ne of the event type for which it is looking. The y the implementation can change over time, so it mply iterate over the list of event types once at I implementation specific event type names is not ons, tool developers may choose to take this route cuse the tool will not have to iterate over the entire e one. ( <i>End of rationale.</i> )		
25 26	Handle Allocation and Deallocation				
27 28 29 30	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 $MPI_T_event_registration$ .				
31 32	MPI_T_EV	ENT_HANDLE_ALLOC(event	_index, obj_handle, info, event_registration)		
33 34	IN	event_index	index of event type for which the registration handle is to be allocated (integer)		
35 36 37	IN	obj_handle	reference to a handle of the MPI object to which this event is supposed to be bound (pointer)		
38	IN	info	info object (handle)		
39	OUT	event_registration	event registration (handle)		
40			2		
41 42 43 44	C binding int MPI_T		event_index, void *obj_handle, _event_registration *event_registration)		
45 46 47 48	MPI_INFO INFO, MPI_I_event_registration *event_registration) MPI_T_EVENT_HANDLE_ALLOC creates a <i>registration handle</i> for the event type iden- tified by event_index. Furthermore, if required by the event type, the registration handle is bound to the object referred to by the argument obj_handle. The argument obj_handle is ignored if the MPI_T_EVENT_GET_INFO call for this event type returned				

MPI_T_BIND_NO_OBJECT in the argument bind. The user can pass hints for the handle allocation to the MPI implementation via the info argument. The allocated event-registration handle is returned in the argument event_registration.

MPI_T_EVENT_HANDLE_SET_INFO(event_registration, info) IN event_registration event registration (handle) IN info info object (handle)

#### C binding

```
int MPI_T_event_handle_set_info(
 MPI_T_event_registration event_registration, MPI_Info info)
```

MPI_T_EVENT_HANDLE_SET_INFO updates the hints of the event-registration handle associated with event_registration 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_T_EVENT_HANDLE_SET_INFO.

Advice to users. Some info items that an implementation can use when it creates an event-registration handle cannot easily be changed once the registration handle is created. Thus, an implementation may ignore hints issued in this call that it would have accepted in a handle allocation call. An implementation may also be unable to update certain info hints in a call to MPI_T_EVENT_HANDLE_SET_INFO. MPI_T_EVENT_HANDLE_GET_INFO can be used to determine whether info changes were ignored by the implementation. (End of advice to users.)

		29
MPI_T_EVENT_HANDLE_GET_INFO(ev	vent_registration, info_used)	30
IN event_registration	event registration (handle)	31
OUT info_used	info object (handle)	32
		33
C binding		34
e binang		35

int MPI_T_event_handle_get_info( MPI_T_event_registration event_registration, MPI_Info *info_used)

MPI_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.

1

 $\mathbf{2}$ 

3

4 5

6

7

8

9 10

11

12

13 14

15

16

17

18

19

20

21

22

23 24 

25

26

2728

35

36

37

38

39

40 41

42

43

44

45

1 2	MPI_T_EV	ENT_REGISTER_CALLBACK( event_cb_function)	(event_registration, cb_safety, info, user_data,
3 4	IN	event_registration	event registration (handle)
5	IN	cb_safety	maximum callback safety level (integer)
6	IN	info	info object (handle)
7	IN	user_data	pointer to a user-controlled buffer
8 9	IN	event_cb_function	pointer to user-defined callback function (function)
10			
11	C binding		
12 13 14 15	int MPI_T	MPI_T_cb_safety cb_sa	tion event_registration, afety, MPI_Info info, void *user_data, ion event_cb_function)
16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 24	by event_cd safety level a given eve Registering registered of for the spe- association When the callback In situation callback fun At call memory reg The us	<b>b_function</b> with an allocated e supported by the callback fu s are defined in Table 15.5. A int-registration handle, potent a callback function for a spec- callback function pointer and cific callback safety level. If e of a callback function for tha an event is triggered, the implex with the lowest safety level ve us where the required callback inction is registered for a given back invocation time, the im- gion specified during callback	ACK associates a user-defined function pointed to event-registration handle. The maximum callback inction is passed in the argument cb_safety. The A user can register multiple callback functions for tially specifying one for each callback safety level. tific callback safety level overwrites any previously- info object associated with the event registration event_cb_function is the NULL pointer, an existing t callback safety level is removed. Hementation will select from all registered callbacks alid in the context in which the callback is invoked. A safety level exceeds the highest level for which a registration handle, the event instance is dropped. plementation with the argument user_data. tistration of the specified callback function to the ent.
34 35 36 37 38 39	is ass callba lower	ociated with the first callback ack safety guarantees should	es can be raised as soon as the registration handle k function, the callback function with the highest be registered before any further registrations for o avoid dropped events due to insufficient callback to users.)
40	The ca	allback function passed to MF	PI_T_EVENT_REGISTER_CALLBACK in the argu-
41 42	ment event	_cb_function needs to have the	e following type:
42 43 44 45 46	typedef v	MPI_T_even	ction)( nt_instance event_instance, nt_registration event_registration, cafety cb_safety,
47 48		void *user	

The argument event_instance corresponds to a handle for the opaque event-instance object of type MPI_T_event_instance. This handle is only valid inside the corresponding invocation of the function to which it is passed. The argument event_registration corresponds to the event-registration handle returned by MPI_T_EVENT_HANDLE_ALLOC for the user function to the same event type and bound object combination. The handle can be used to identify the specific event registration information, such as event type and bound object, or even to deallocate the handle from within the callback invocation. The argument cb_safety describes the safety requirements the callback function must fulfill in the current invocation. The argument user_data is the pointer to user-allocated memory that was passed to the MPI implementation during callback registration.

MPI_T_E	VENT_CALLBACK_SET	_INFO(event_registration, cb_safety, info)
IN	event_registration	event registration (handle)
IN	cb_safety	callback safety level (integer)
IN	info	info object (handle)
C bindir	ıg	

int MPI_T_event_callback_set_info(

```
MPI_T_event_registration event_registration,
MPI_T_cb_safety cb_safety, MPI_Info info)
```

MPI_T_EVENT_CALLBACK_SET_INFO updates 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 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_T_EVENT_CALLBACK_SET_INFO.

MPI_T_EVENT_CALLBACK_GET_INFO(event_registration, cb_safety, info_used)			
IN	event_registration	event registration (handle)	
IN	cb_safety	callback safety level (integer)	
OUT	info_used	info object (handle)	

### C binding

MPI_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

### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

5

6 7

8 9

10

20

21

22 23

 24 

25

26

27

28

29 30 31

36 37

38

39

40

1 no such hints exist, a handle to a newly created info object is returned that contains no  $\mathbf{2}$ key/value pairs. The user is responsible for freeing info_used via MPI_INFO_FREE. 3 To stop the MPI implementation from raising events for a specific registration, a user 4 needs to free the corresponding event-registration handle. 56 MPI_T_EVENT_HANDLE_FREE(event_registration, user_data, free_cb_function) 7 8 event registration (handle) IN event_registration 9 IN user_data pointer to a user-controlled buffer 10 free_cb_function pointer to user-defined callback function (function) IN 11 1213 C binding int MPI_T_event_handle_free(MPI_T_event_registration event_registration, 1415void *user_data, 16MPI_T_event_free_cb_function free_cb_function) 17 MPI_T_EVENT_HANDLE_FREE returns MPI_SUCCESS when deallocation of the handle 18 was initiated successfully and returns MPI_T_ERR_INVALID_HANDLE if 19 event_registration does not match a valid allocated event-registration handle at the time 20of the call. The callback function free_cb_function is called by the MPI implementation, 21when it is able to guarantee that no further event instances for the corresponding event-22 registration handle will be raised. If the pointer to free_cb_function is the NULL pointer, no 23user function is invoked after successful deallocation of the event registration handle. The 24pointer to user-controlled memory provided in the user_data argument will be passed to the 25function provided in the free_cb_function on invocation. 2627Advice to users. A free-callback function associated with a registration handle should 28always be prepared to postpone any pending actions, should the provided callback 29 safety requirements exceed those required by the pending actions. (End of advice to 30 users.) 3132 The callback function passed to MPI_T_EVENT_HANDLE_FREE in the argument 33 free_cb_function needs to have the following type: 3435 typedef void (*MPI_T_event_free_cb_function)( 36 MPI_T_event_registration event_registration, 37 MPI_T_cb_safety cb_safety, 38 void *user_data); 39 40 Handling Dropped Events 41 Events may occur at times when the MPI implementation cannot invoke the user function 42corresponding to a matching event handle. An implementation is allowed to buffer such 43 events and delay the callback invocation. If an event occurs at times when the corresponding 44 callback function cannot be called and the corresponding data cannot be buffered, or no 45callback function meeting the required callback safety level is registered, the event data may 46 be dropped. To discover such data loss, the user can set a handler function for a specific 47

IN	event_registration	valid event registration (handle)
IN	dropped_cb_function	pointer to user-defined callback function (function)
C bind	ing	
nt MPI	 T_event_set_dropped_han	dler(
	MPI_T_event_regist	cration event_registration,
	$MPI_T_event_dropped$	ed_cb_function dropped_cb_function)
MP	I T EVENT SET DROPPED	_HANDLER registers the function
		the MPI implementation when event information is
		pecified in event_registration. Subsequent calls to
PI_T_	EVENT_SET_DROPPED_HA	NDLER with the same registration handle will replace
		ns for that registration handle. If the pointer to
	_	ter, no data loss is recorded or reported until a new
alid cal	llback function is registered.	
Ad	<i>lvice to users.</i> The invocation	on of the dropped handler callback function may not
ne		time the event was actually lost. (End of advice to
us	ers.)	
The	e callback function passed to	MPI_T_EVENT_SET_DROPPED_HANDLER in the
	nt dropped_cb_function needs	
-		
ypedei		<pre>ped_cb_function)(int count, vent_registration event_registration,</pre>
		rce_index,
		b_safety cb_safety,
		<pre>ser_data);</pre>
m	· ·	
		corresponds to the event registration handle to which
-		argument <b>count</b> provides a best effort estimation of ered event callback corresponding to <b>event_registration</b>
		istration of the dropped-callback handler or the last
		allback handler. The source_index provides the index
		sponding event information. The argument cb_safety
		callback function must fulfill in the current invocation.
he pos	sible values for cb_safety are	described in Table 15.5. The argument $user_data$ is
-		v that was passed to the MPI implementation during
llback	registration.	
A	<i>lvice to users.</i> A callback fu	unction for dropped events associated with a registra-
		repared to postpone any pending actions, should the
		ements exceed those required by the pending actions.
-	End of advice to users.)	• • • • •
`	,	

Advice to implementors.A high-quality implementation will strive to invoke a 45 callback function for dropped events associated with a registration handle at times 46 that provide as much freedom of action to the function as possible. (End of advice to 47 implementors.) 48 

### Unofficial Draft for Comment Only

¹ If events are dropped for a specific source, the corresponding handler callback function ² must be called before other events are raised for this source. This means in a sequence of five ³ events E1 to E5 from the same source, where E3 and E4 were dropped, any handler func-⁴ tion set through MPI_T_EVENT_SET_DROPPED_HANDLER for event-registration handles ⁵ 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	event_instance	event-instance handle provided to the callback function (handle)
IN	element_index	index into the array of datatypes of the item to be queried (integer)
OUT	buffer	pointer to a memory location to store the item data (choice)
C bindin int MPI_1	g [_event_read(MPI_T_event_;	

int element_index, void *buffer)

⁴⁴ MPI_T_EVENT_READ allows users to copy one element of the event data to a user-⁴⁵ 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

6 7

8

20

21

22

23

24

25

26 27

28

29 30 31

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		event instance provided to the callback function			
		(handle)	7		
OUT	buffer	user-allocated buffer for event data (choice)	8		
001	buildi		9		
C hindin	~		10		
C binding		and a second instance and thuffer)	11		
int MP1_1	_event_copy(MP1_1_event_1	instance event_instance, void *buffer)	12		
MPI_	$\Gamma_EVENT_COPY$ copies the e ⁻	vent data as a whole into the user-provided buffer.	13		
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 corresp	oonding call to MPI_T_EVENT	_GET_INFO. The data may include padding bytes	16		
between in	idividual elements of the ever	t data in the buffer. A user can reconstruct the	17		
location an	nd size of the data contained :	in the buffer through the information returned by	18		
MPI_T_EVENT_GET_INFO.					
Advi	ce to implementors. An imp	lementation should strive to use an appropriately	21		
comp	pact representation when copy	ring event instance data to a user buffer via	22		
MPI_	T_EVENT_COPY to reduce t	he amount of memory required for the user buffer.	23		
(End	l of advice to implementors.)		24		
			25		
Reading Ev	ent Meta Data		26		
A 1 1.4 · · · · · · ·			27		
	-	oded by each event type, supplemental information	28		
available across all event types can be queried.					

MPI_T_EVENT_GET_TIMESTAMP(event_instance, event_tin	ent_timestamp)
-----------------------------------------------------	----------------

IN	event_instance	Y	event instance provided to the callback function (handle)
OUT	event_timestamp		timestamp the event was observed (integer)

### C binding

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 48

### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

38

39

40

 41 

42

43

 $44 \\ 45$ 

1			tured during callback function invocation. (End of advice
2	to u	users.)	
3 4	1 da	vice to implementance A	high-quality implementation will return a timestamp as
4 5			est time the event was observed by the MPI implementa-
6		n. (End of advice to impl	
7	0101		
8	An e	event may be raised from	different components acting as event sources in the MPI
9			context is an abstract concept that helps to define partial
10	-		source provides its own ordering guarantees. A source
11	describes	the entity that raises the	e event, rather than the origin of the data.
12	To ic	dentify the source of an e	vent instance, the user can query the index of the source
13	within th	e corresponding event ca	llback function invocation.
14			
15		-	n excessive number of event sources may negatively impact
16	-		o per-source overhead in event handling. (End of advice
17	to i	implementors.)	
18			
19			
20	MPI_T_E	EVENT_GET_SOURCE(ev	vent_instance, source_index)
21	IN	event_instance	event instance provided to the callback function
22			(handle)
23	OUT	source_index	index identifying the source (integer)
24 25	001	Source_macx	index identifying the source (integer)
26	C bindi	ng and	
27		e e e e e e e e e e e e e e e e e e e	PI_T_event_instance event_instance,
28	1110 III I <u></u>	int *source_ind	
29			
30			identifies the event instance to query. It is erroneous
31	-		nce handle to the call than the one passed by the MPI
32	-		nction in which the source is queried.
33			eturns the index of the source of the event instance. It ation on the source using MPI_T_SOURCE_GET_INFO.
34	can be us	sed to query more morm	ation on the source using Wir1_1_SOURCE_GET_INTO.
35	Rat	tionale. Event callback f	unction invocations are associated with a source to enable
36			vents on the tool side, when required, while retaining low
37			MPI implementation. (End of rationale.)
38 39			
40	15.3.9	Variable Categorization	
41	MPI impl	ementations can optional	ly group performance and control variables into categories
42			tween various variables. For example, an MPI implemen-
43	-		performance variables that refer to message transfers in
44		o .	reby distinguish them from variables that refer to local
45		implementation and the	
		-	ons or other interactions with the operating system.
46	resources	such as memory allocati	ons or other interactions with the operating system. her categories to form a hierarchical grouping. Categories
46 47	resources Cate	such as memory allocati gories can also contain ot	

### Unofficial Draft for Comment Only

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 num_c	at cur	rent number	of categ	gories (i	integer)	
-----------	--------	-------------	----------	-----------	----------	--

### C binding

int MPI_T_category_get_num(int *num_cat)

Individual category information can then be queried by calling the following function:

1 2	MPI_T_CA	TEGORY_GET_INFO(cat_ind num_pvars, num_categor	ex, name, name_len, desc, desc_len, num_cvars, ies)			
3	IN	cat_index	index of the category to be queried (integer)			
4 5 6	OUT	name	buffer to return the string containing the name of the category (string)			
7	INOUT	name_len	length of the string and/or buffer for name (integer)			
8 9 10	OUT	desc	buffer to return the string containing the description of the category (string)			
11	INOUT	desc_len	length of the string and/or buffer for $desc$ (integer)			
12	OUT	num_cvars	number of control variables in the category (integer)			
13 14 15	OUT	num_pvars	number of performance variables in the category (integer)			
16 17	Ουτ	num_categories	number of categories contained in the category (integer)			
18						
19         20         21         22         23         24         25         26         27         28         29         30         31         32         33         34         35         36         37         38         39         40	<pre>C binding int MPI_T_category_get_info(int cat_index, char *name, int *name_len,</pre>					
41	MPI_T_CA	TEGORY_GET_NUM_EVENT	S(cat_index, num_events)			
42 43	IN	cat_index	index of the category to be queried (integer)			
43 44 45	OUT	num_events	number of event types in the category (integer)			
46 47 48	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.

		,		
$MPI_T_C$	ATEGORY	_GET_INDEX(name,	cat_index)	

IN	name	the name of the category (string)
OUT	cat_index	the index of the category (integer)

#### 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`	)
			cut_mack,	ien,	marces	,

		cut_m		33
IN	cat_index		index of the category to be queried, in the range	34
			from 0 to $num_cat - 1$ (integer)	35
IN	len		the length of the indices array (integer)	36
OUT	indices		an integer array of size len, indicating control	37
001	indices		variable indices (array of integers)	38
			variable indices (array of integers)	39

#### 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.

 $\mathbf{2}$ 

```
1
 MPI_T_CATEGORY_GET_PVARS(cat_index, len, indices)
2
 IN
 cat_index
 index of the category to be queried, in the range
3
 from 0 to num_cat -1 (integer)
4
 IN
 len
 the length of the indices array (integer)
5
6
 OUT
 indices
 an integer array of size len, indicating performance
7
 variable indices (array of integers)
8
9
 C binding
10
 int MPI_T_category_get_pvars(int cat_index, int len, int indices[])
11
 MPI_T_CATEGORY_GET_PVARS can be used to query which performance variables
12
 are contained in a particular category. A category contains zero or more performance
13
 variables.
14
15
16
 MPI_T_CATEGORY_GET_EVENTS(cat_index, len, indices)
17
 index of the category to be queried, in the range
 IN
 cat_index
18
 from 0 to num_cat - 1 (integer)
19
20
 IN
 len
 the length of the indices array (integer)
21
 OUT
 indices
 an integer array of size len, indicating event type
22
 indices (array of integers)
23
24
 C binding
25
 int MPI_T_category_get_events(int cat_index, int len, int indices[])
26
27
 MPI_T_CATEGORY_GET_EVENTS can be used to query which event types are con-
28
 tained in a particular category. A category contains zero or more event types.
29
30
 MPI_T_CATEGORY_GET_CATEGORIES(cat_index, len, indices)
31
32
 IN
 cat_index
 index of the category to be queried, in the range
33
 from 0 to num_cat - 1 (integer)
34
 IN
 len
 the length of the indices array (integer)
35
 OUT
 indices
36
 an integer array of size len, indicating category
37
 indices (array of integers)
38
39
 C binding
40
 int MPI_T_category_get_categories(int cat_index, int len, int indices[])
41
 MPI_T_CATEGORY_GET_CATEGORIES can be used to query which other categories
42
 are contained in a particular category. A category contains zero or more other categories.
43
 As mentioned above, MPI implementations can grow the number of categories as well
44
 as the number of variables or other categories within a category. In order to allow users
45
 of the MPI tool information interface to check quickly whether new categories have been
46
 added or new variables or categories have been added to a category, MPI maintains a
47
48
```

virtual timestamp. This timestamp is monotonically increasing during the execution and is returned by the following function:

### MPI_T_CATEGORY_CHANGED(stamp)

OUT stamp

a virtual time stamp to indicate the last change to the categories (integer)

#### C binding

int MPI_T_category_changed(int *stamp)

If two subsequent calls to this routine return the same timestamp, it is guaranteed that the category information has not changed between the two calls. If the timestamp retrieved from the second call is higher, then some categories have been added or expanded.

Advice to users. The timestamp value is purely virtual and only intended to check for changes in the category information. It should not be used for any other purpose. (End of advice to users.)

The index values returned in indices by MPI_T_CATEGORY_GET_CVARS,
MPI_T_CATEGORY_GET_PVARS, MPI_T_CATEGORY_GET_EVENTS, and
MPI_T_CATEGORY_GET_CATEGORIES can be used as input to
MPI_T_CVAR_GET_INFO, MPI_T_PVAR_GET_INFO, MPI_T_EVENT_GET_INFO, and
MPI_T_CATEGORY_GET_INFO, respectively.

The user is responsible for allocating the arrays passed into the functions MPI_T_CATEGORY_GET_CVARS, MPI_T_CATEGORY_GET_PVARS, MPI_T_CATEGORY_GET_EVENTS, and MPI_T_CATEGORY_GET_CATEGORIES. Starting from array index 0, each function writes up to len elements into the array. If the category contains more than len elements, the function returns an arbitrary subset of size len. Otherwise, the entire set of elements is returned in the beginning entries of the array, and any remaining array entries are not modified.

### 15.3.10 Return Codes for the MPI Tool Information Interface

All functions defined as part of the MPI tool information interface return an integer error code (see Table 15.7) to indicate whether the function was completed successfully or was aborted. In the latter case, the error code indicates the reason for not completing the routine. Such errors neither impact the execution of the MPI process nor invoke MPI error handlers. The MPI process continues executing regardless of the return code from the call. The MPI implementation is not required to check all user-provided parameters; if a user passes invalid parameter values to any routine the behavior of the implementation is undefined.

All error codes with the prefix MPI_T_ must be unique values and cannot overlap with any other error codes or error classes returned by the MPI implementation. Further, they shall be treated as MPI error classes as defined in Section 9.4 and follow the same rules and restrictions. In particular, they must satisfy:

$$0 = MPI_SUCCESS < MPI_T_ERR_XXX \le MPI_ERR_LASTCODE$$

#### **Unofficial Draft for Comment Only**

1

2

3 4

> 5 6

9

10

11

12

13

14 15

16

17

25

26

27

28

29

30

31 32

33 34

35

36

37

38

39

40

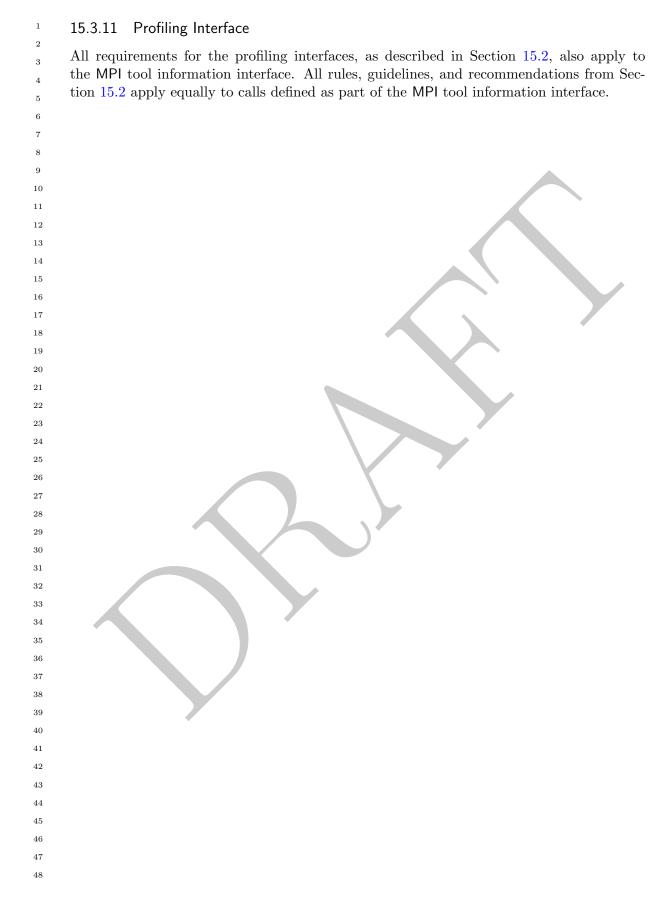
41

42

43

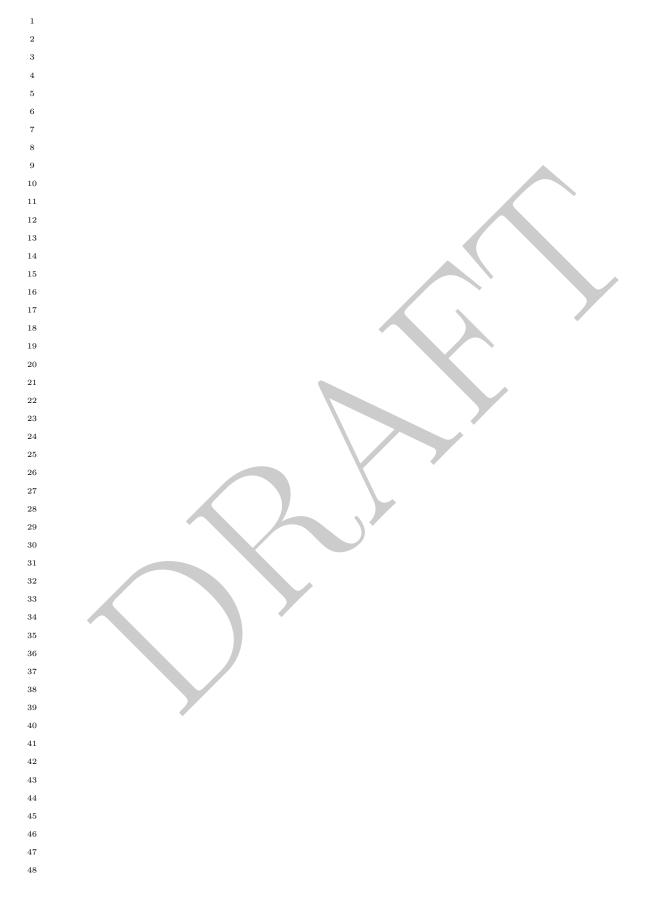
44

45



Return Code	Description
Return Codes for All Functions in th	he MPI Tool Information Interface
MPI_SUCCESS	Call completed successfully
MPI_T_ERR_INVALID	Invalid or bad parameter value(s)
MPI_T_ERR_MEMORY	Out of memory
MPI_T_ERR_NOT_INITIALIZED	Interface not initialized
MPI_T_ERR_CANNOT_INIT	Interface not in the state to be initialized
MPI_T_ERR_NOT_ACCESSIBLE	Requested functionality not accessible
Return Codes for Datatype Function	as: MPI_T_ENUM_*
MPI_T_ERR_INVALID_INDEX	The enumeration index is invalid
Return Codes for Variable, Category	y, and Event Query Functions: MPI_T_*_GET_*
MPI_T_ERR_INVALID_INDEX	The variable or category index is invalid
MPI_T_ERR_INVALID_NAME	The variable or category name is invalid
Return Codes for Handle Functions:	MPI_T_*_{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 Expe	eriment 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 A	Access Functions: MPI_T_CVAR_{READ WRITE}
MPI_T_ERR_CVAR_SET_NOT_NOW	Variable cannot be set at this moment
MPI_T_ERR_CVAR_SET_NEVER	Variable cannot be set until end of execution
MPI_T_ERR_INVALID_HANDLE	The handle is invalid
Return Codes for Performance Varia	able Access and Control:
MPI_T_PVAR_{START STOP READ	) 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)
MPI_T_ERR_INVALID_INDEX	The source index is invalid
MPI_T_ERR_NOT_SUPPORTED	Requested functionality not supported
Return Codes for Category Function	
	The category index is invalid

Table 15.7: Return codes used in functions of the MPI tool information interface



# Chapter 16

# **Deprecated Interfaces**

## 16.1 Deprecated since MPI-2.0

The following function is deprecated and is superseded by MPI_COMM_CREATE_KEYVAL in MPI-2.0. The language independent definition of the deprecated function is the same as that of the new function, except for the function name and a different behavior in the C/Fortran language interoperability, see Section 19.3.7. The language bindings are modified.

MPI_KE	YVAL_CREATE(copy_fn	n, delete_fn, keyval, extra_state)	22
IN	copy_fn	Copy callback function for keyval	23
IN	delete_fn	Delete callback function for keyval	24 25
OUT	keyval	key value for future access (integer)	26
IN	extra_state	Extra state for callback functions	27 28
			29
C bindi	ng		30
int MPI		Copy_function *copy_fn,	31
		nction *delete_fn, int *keyval,	32
	void *extra_s	tate)	33
For this :	routine, an interface wi	thin the mpi_f08 module was never defined.	34
		• -	35
	binding		36
		DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)	37
	ERNAL COPY_FN, DELET		38
TN11	EGER KEYVAL, EXTRA_S	STATE, IERRUR	39
The	copy_fn function is in	voked when a communicator is duplicated by	40
MPI_CO	MM_DUP. copy_fn shou	Id be of type MPI_Copy_function, which is defined as follows:	41
			42
typedef	int MPI_Copy_funct:	ion(MPI_Comm oldcomm, int keyval,	43
	void *extra_s	tate, void *attribute_val_in,	44
	void *attribu	<pre>te_val_out, int *flag);</pre>	45
ΛE	ortron declaration for a	uch a function is as follows:	46
			47
FOI UIIS	iouine, an interface wi	thin the mpi_f08 module was never defined.	48

1 2 3 4 5 6	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
7 8 9 10 11 12 13 14 15 16 17	<pre>copy_fn may be specified as MPI_NULL_COPY_FN or MPI_DUP_FN from either C or Fortran; MPI_NULL_COPY_FN is a function that does nothing other than returning flag = 0 and MPI_SUCCESS. MPI_DUP_FN is a simple-minded copy function that sets flag = 1, returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS. Note that MPI_NULL_COPY_FN and MPI_DUP_FN are also deprecated. Analogous to copy_fn is a callback deletion function, defined as follows. The delete_fn function is invoked when a communicator is deleted by MPI_COMM_FREE or when a call is made explicitly to MPI_ATTR_DELETE. delete_fn should be of type MPI_Delete_function, which is defined as follows: typedef int MPI_Delete_function(MPI_Comm comm, int keyval, void *attribute_val, void *extra_state);</pre>
18 19	A Fortran declaration for such a function is as follows: For this routine, an interface within the mpi_f08 module was never defined.
20 21 22	SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR) INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR
23 24 25 26 27 28 29 30	delete_fn may be specified as MPI_NULL_DELETE_FN from either C or Fortran; MPI_NULL_DELETE_FN is a function that does nothing, other than returning MPI_SUCCESS. Note that MPI_NULL_DELETE_FN is also deprecated. The following function is deprecated and is superseded by MPI_COMM_FREE_KEYVAL in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.
31	MPI_KEYVAL_FREE(keyval)
32 33 34 35 36	INOUT keyval Frees the integer key value (integer) C binding int MPI_Keyval_free(int *keyval)
37	For this routine, an interface within the mpi_f08 module was never defined.
38 39 40 41	Fortran binding MPI_KEYVAL_FREE(KEYVAL, IERROR) INTEGER KEYVAL, IERROR
42 43 44 45 46 47 48	The following function is deprecated and is superseded by MPI_COMM_SET_ATTR in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

MPI_ATTR_PUT(comm, keyval, attribute_val)			1
INOUT	comm	communicator to which attribute will be attached	2
moor	comm	(handle)	3
		(nancie)	4
IN	keyval	key value, as returned by $MPI_KEYVAL_CREATE$	5
		(integer)	6
IN	attribute_val	attribute value	7
			8
C binding			
, i i i i i i i i i i i i i i i i i i i	int MPI_Attr_put(MPI_Comm comm, int keyval, void *attribute_val)		
INC IN I_A	int in i_kttl_put(in i_comm comm, int keyvai, void wattlibute_vai)		
For this routine, an interface within the mpi_f08 module was never defined.			12

MPI_ATTR_PUT	(comm, keyval,	attribute_val)
--------------	----------------	----------------

### 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 of the new function, except of the function name. The language bindings are modified.

## MPI_ATTR_GET(comm, keyval, attribute_val, flag)

IN	comm	communicator to which attribute is attached (handle)	23
IN	keyval	key value (integer)	24
	5		25
OUT	attribute_val	attribute value, unless $flag = false$	26
OUT	flag	true if an attribute value was extracted; false if no	27
		attribute is associated with the key	28

### C binding

```
int MPI_Attr_get(MPI_Comm comm, int keyval, void *attribute_val, int *flag)
For this routine, an interface within the mpi_f08 module was never defined.
Fortran binding
MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
 INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
 LOGICAL FLAG
```

The following function is deprecated and is superseded by MPI_COMM_DELETE_ATTR in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

## **Unofficial Draft for Comment Only**

```
1
 MPI_ATTR_DELETE(comm, keyval)
2
 INOUT
 comm
 communicator to which attribute is attached (handle)
3
 IN
 keyval
 The key value of the deleted attribute (integer)
4
5
6
 C binding
7
 int MPI_Attr_delete(MPI_Comm comm, int keyval)
8
 For this routine, an interface within the mpi_f08 module was never defined.
9
10
 Fortran binding
11
 MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)
12
 INTEGER COMM, KEYVAL, IERROR
13
14
 16.2
 Deprecated since MPI-2.2
15
16
 The entire set of C++ language bindings have been removed. See Chapter 17, Removed
17
 Interfaces for more information.
18
 The following function typedefs have been deprecated and are superseded by new
19
 names. Other than the typedef names, the function signatures are exactly the same; the
20
 names were updated to match conventions of other function typedef names.
21
22
 Deprecated Name
 New Name
23
 MPI_Comm_errhandler_fn
 MPI_Comm_errhandler_function
24
 MPI_File_errhandler_fn
 MPI_File_errhandler_function
25
 MPI_Win_errhandler_fn
 MPI Win_errhandler_function
26
27
 Deprecated since MPI-4.0
 16.3
28
29
 Cancelling a send request by calling MPI_CANCEL has been deprecated and may be removed
30
 in a future version of the MPI specification.
^{31}
 The following function is deprecated and is superseded by the new
32
 MPI_INFO_GET_STRING call in MPI-4.0.
33
34
35
 MPI_INFO_GET(info, key, valuelen, value, flag)
36
 IN
 info
37
 info object (handle)
38
 IN
 key
 key (string)
39
 valuelen
 IN
 length of value arg (integer)
40
 OUT
 value
 value (string)
41
42
 OUT
 flag
 true if key defined, false if not (boolean)
43
44
 C binding
45
 int MPI_Info_get(MPI_Info info, const char *key, int valuelen, char *value,
46
 int *flag)
47
48
```

```
1
Fortran 2008 binding
 2
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
 10
MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)
 11
 INTEGER INFO, VALUELEN, IERROR
 12
 CHARACTER*(*) KEY, VALUE
 13
 LOGICAL FLAG
 14
 15
 The following function is deprecated and is superseded by the new
 16
MPI_INFO_GET_STRING call in MPI-4.0.
 17
 18
MPI_INFO_GET_VALUELEN(info, key, valuelen, flag)
 19
 20
 IN
 info
 info object (handle)
 21
 IN
 key
 key (string)
 22
 OUT
 valuelen
 length of value arg (integer)
 23
 ^{24}
 OUT
 true if key defined, false if not (boolean)
 flag
 25
 26
C binding
 27
int MPI_Info_get_valuelen(MPI_Info info, const char *key, int *valuelen,
 28
 int *flag)
 29
Fortran 2008 binding
 30
MPI_Info_get_valuelen(info, key, valuelen, flag, ierror)
 31
 TYPE(MPI_Info), INTENT(IN) :: info
 32
 CHARACTER(LEN=*), INTENT(IN) :: key
 33
 INTEGER, INTENT(OUT) :: valuelen
 34
 LOGICAL, INTENT(OUT) :: flag
 35
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 36
 37
Fortran binding
 38
MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
 39
 INTEGER INFO, VALUELEN, IERROR
 40
 CHARACTER*(*) KEY
 41
 LOGICAL FLAG
 42
 The following return class has been deprecated and is superseded by a new name.
 43
 44
 Deprecated Name
 Replacement Name
 45
 MPI_T_ERR_INVALID_ITEM
 MPI_T_ERR_INVALID_INDEX
 46
 47
 The following Fortran subroutines are deprecated because the Fortran language
```

storage_size() and c_sizeof() intrinsic functions provide similar functionality. Note that while

### Unofficial Draft for Comment Only

```
1
 MPI_SIZEOF and c_sizeof() return the size in bytes, storage_size() provides the size in bits.
^{2}
3
4
 MPI_SIZEOF(x, size)
\mathbf{5}
6
 IN
 a Fortran variable of numeric intrinsic type (choice)
 х
\overline{7}
 OUT
 size
 size of machine representation of that type (integer)
8
9
 Fortran 2008 binding
10
 MPI_Sizeof(x, size, ierror)
11
 TYPE(*), DIMENSION(..) :: x
12
 INTEGER, INTENT(OUT) :: size
13
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
 Fortran binding
16
 MPI_SIZEOF(X, SIZE, IERROR)
17
 <type> X
18
 INTEGER SIZE, IERROR
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
```

# 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	27
			28
_	Removed	MPI-2 Replacement	29
	MPI_ADDRESS	MPI_GET_ADDRESS	30
	MPI_ERRHANDLER_CREATE	MPI_COMM_CREATE_ERRHANDLER	31
	MPI_ERRHANDLER_GET	MPI_COMM_GET_ERRHANDLER	32
	MPI_ERRHANDLER_SET	MPI_COMM_SET_ERRHANDLER	33
	MPI_TYPE_EXTENT	MPI_TYPE_GET_EXTENT	34
	MPI_TYPE_HINDEXED	MPI_TYPE_CREATE_HINDEXED	35
	MPI_TYPE_HVECTOR	MPI_TYPE_CREATE_HVECTOR	36
	MPI_TYPE_LB	MPI_TYPE_GET_EXTENT	37
	MPI_TYPE_STRUCT	MPI_TYPE_CREATE_STRUCT	38
	MPI_TYPE_UB	MPI_TYPE_GET_EXTENT	39
		<u> </u>	40
			10

### 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.

 $41 \\ 42$ 

 $45 \\ 46$ 

Table 17.2: Removed MPI-1 datatypes. The indicated routine may be used for changing the lower and upper bound respectively.

4	Removed MPI-2 Replacement		
5	MPI_LB MPI_TYPE_CREATE_RESIZED		
6	MPI_UB MPI_TYPE_CREATE_RESIZED		
7			
8			
9 10	Table 17.3: Removed MPI-1 constants		
11			
12	Removed MPI-1 Constants		
13	C type: const int (or unnamed enum)		
14	Fortran type: INTEGER		
15	MPI_COMBINER_HINDEXED_INTEGER		
16	MPI_COMBINER_HVECTOR_INTEGER		
17	MPI_COMBINER_STRUCT_INTEGER		
18			
19			
20	17.1.5 Removed MPI-1 Callback Prototypes		
21	Table 17.4 shows the removed MPI-1 callback prototypes and their replacements.		
22			
23 24	Table 17.4: Removed MPI-1 callback prototypes and their replacements		
25	Table 11.4. Removed Will I camback prototypes and then replacements		
26	Removed MPI-2 Replacement		
27	MPI_Handler_function MPI_Comm_errhandler_function		
28			
29			
30			
31	17.2 C++ Bindings		
32			
33	The C++ bindings were deprecated as of MPI-2.2. The C++ bindings are removed in		
34	MPI-3.0. The namespace is still reserved, however, and bindings may only be provided by		
35	an implementation as described in the MPI-2.2 standard.		
36			
37			
38			
39			
40			
41			
42			
43			
44			
45			
46			
47			
48			

1

 $\mathbf{2}$ 

# Chapter 18

# **Backward Incompatibilities**

### 18.1 Backward Incompatibilities 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.

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 was done to avoid exceeding the limit on some compilation systems.

**Unofficial Draft for Comment Only** 

*Rationale.* For Fortran, this limit was already relaxed for the Fortran specific function names, see Section 19.1.5, and the Fortran language specification 2003 requires support for a minimum of 63 characters for internal and external identifiers. Starting with the ISO/IEC 9899:1999 C programming language standard, support for a minimum of 63 characters is required for internal identifiers, but only 31 characters are required to be significant for external identifiers. At the time of the release of MPI-4.0, most or nearly all compilers allow external identifiers longer than 31 characters. Therefore, the restriction is removed. (*End of rationale.*)

 24 



# Chapter 19

# Language Bindings

### 19.1 Support for Fortran

### 19.1.1 Overview

The Fortran MPI language bindings have been designed to be compatible with the Fortran 90 standard with additional features from Fortran 2003 and Fortran 2008 [45] + TS 29113 [46].

Rationale. Fortran 90 contains numerous features designed to make it a more "modern" language than Fortran 77. It seems natural that MPI should be able to take advantage of these new features with a set of bindings tailored to Fortran 90. In Fortran 2008 + TS 29113, the major new language features used are the ASYNCHRONOUS attribute to protect nonblocking MPI operations, and assumed-type and assumed-rank dummy arguments for choice buffer arguments. Further requirements for compiler support are listed in Section 19.1.7. (*End of rationale.*)

MPI defines three methods of Fortran support:

- 1. USE mpi_f08: This method is described in Section 19.1.2. It requires compile-time argument checking with unique MPI handle types and provides techniques to fully solve the optimization problems with nonblocking calls. This is the only Fortran support method that is consistent with the Fortran standard (Fortran 2008 + TS 29113 and later). This method is highly recommended for all MPI applications.
- 2. USE mpi: This method is described in Section 19.1.3 and requires compile-time argument checking. Handles are defined as INTEGER. This Fortran support method is inconsistent with the Fortran standard, and its use is therefore not recommended. It exists only for backwards compatibility.
- 3. **INCLUDE 'mpif.h':** This method is described in Section 19.1.4. The use of the include file mpif.h is strongly discouraged starting with MPI-3.0, because this method neither guarantees compile-time argument checking nor provides sufficient techniques to solve the optimization problems with nonblocking calls, and is therefore inconsistent with the Fortran standard. It exists only for backwards compatibility with legacy MPI applications.

1 Compliant MPI-3 implementations providing a Fortran interface must provide one or  $\mathbf{2}$ both of the following: 3 • The USE mpi_f08 Fortran support method. 4 5• The USE mpi and INCLUDE 'mpif.h' Fortran support methods. 6  $\overline{7}$ Section 19.1.6 describes restrictions if the compiler does not support all the needed features. 8 Application subroutines and functions may use either one of the modules or the mpif.h 9 include file. An implementation may require the use of one of the modules to prevent type 10mismatch errors. 11Advice to users. Users are advised to utilize one of the MPI modules even if mpif.h 12enforces type checking on a particular system. Using a module provides several poten-13 tial advantages over using an include file; the mpi_f08 module offers the most robust 14and complete Fortran support. (End of advice to users.) 1516In a single application, it must be possible to link together routines which USE mpi_f08, 17 USE mpi, and INCLUDE 'mpif.h'. 18 The LOGICAL compile-time constant MPI_SUBARRAYS_SUPPORTED is set to .TRUE. if 19all buffer choice arguments are defined in explicit interfaces with assumed-type and assumed-20rank [46]; otherwise it is set to .FALSE.. The LOGICAL compile-time constant 21MPI_ASYNC_PROTECTS_NONBLOCKING is set to .TRUE. if the ASYNCHRONOUS attribute was 22added to the choice buffer arguments of all nonblocking interfaces and the underlying 23Fortran compiler supports the ASYNCHRONOUS attribute for MPI communication (as part of  24 TS 29113), otherwise it is set to .FALSE.. These constants exist for each Fortran support 25method, but not in the C header file. The values may be different for each Fortran support 26method. All other constants and the integer values of handles must be the same for each 27Fortran support method. 28Section 19.1.2 through 19.1.4 define the Fortran support methods. The Fortran in-29terfaces of each MPI routine are shorthands. Section 19.1.5 defines the corresponding 30 full interface specification together with the specific procedure names and implications for  31 the profiling interface. Section 19.1.6 the implementation of the MPI routines for differ-32 ent versions of the Fortran standard. Section 19.1.7 summarizes major requirements for 33 valid MPI-3.0 implementations with Fortran support. Section 19.1.8 and Section 19.1.9 de-34scribe additional functionality that is part of the Fortran support. MPI_F_SYNC_REG is 35 needed for one of the methods to prevent register optimization problems. A set of functions 36 provides additional support for Fortran intrinsic numeric types, including parameterized 37 types: MPI_TYPE_MATCH_SIZE, MPI_TYPE_CREATE_F90_INTEGER, 38 MPI_TYPE_CREATE_F90_REAL and MPI_TYPE_CREATE_F90_COMPLEX. In the context 39 of MPI, parameterized types are Fortran intrinsic types which are specified using KIND type 40 parameters. Sections 19.1.10 through 19.1.19 give an overview and details on known prob-41 lems when using Fortran together with MPI; Section 19.1.20 compares the Fortran problems 42with those in C. 43

43 44

### 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

### Unofficial Draft for Comment Only

of the set of two Fortran routine interface specifications is provided by this module. This module must:

- Define all named MPI constants.
- Declare MPI functions that return a value.
- Provide explicit interfaces according to the Fortran routine interface specifications. This module therefore guarantees compile-time argument checking for all arguments which are not TYPE(*), with the following exception:

Only one Fortran interface is defined for functions that are deprecated as of MPI-3.0. This interface must be provided as an explicit interface according to the rules defined for the mpi module, see Section 19.1.3.

Advice to users. It is strongly recommended that developers substitute calls to deprecated routines when upgrading from mpif.h or the mpi module to the mpi_f08 module. (End of advice to users.)

- Define the derived type MPI_Status, and define all MPI handles with uniquely named handle types (instead of INTEGER handles, as in the mpi module). This is reflected in the first Fortran binding in each MPI function definition throughout this document (except for the deprecated routines).
- Overload the operators .EQ. and .NE. to allow the comparison of these MPI handles with .EQ., .NE., == and /=.
- Use the ASYNCHRONOUS attribute to protect the buffers of nonblocking operations, and set the LOGICAL compile-time constant MPI_ASYNC_PROTECTS_NONBLOCKING to .TRUE. if the underlying Fortran compiler supports the ASYNCHRONOUS attribute for MPI communication (as part of TS 29113). See Section 19.1.6 for older compiler versions.

• Set the LOGICAL compile-time constant MPI_SUBARRAYS_SUPPORTED to .TRUE. and declare choice buffers using the Fortran 2008 TS 29113 features assumed-type and assumed-rank, i.e., TYPE(*), DIMENSION(..) in all nonblocking, split collective and persistent communication routines, if the underlying Fortran compiler supports it. With this, non-contiguous sub-arrays can be used as buffers in nonblocking routines.

*Rationale.* In all blocking routines, i.e., if the choice-buffer is not declared as ASYNCHRONOUS, the TS 29113 feature is not needed for the support of non-contiguous buffers because the compiler can pass the buffer by in-and-out-copy through a contiguous scratch array. (*End of rationale.*)

- Set the MPI_SUBARRAYS_SUPPORTED compile-time constant to .FALSE. and declare choice buffers with a compiler-dependent mechanism that overrides type checking if the underlying Fortran compiler does not support the Fortran 2008 TS 29113 assumed-type and assumed-rank notation. In this case, the use of non-contiguous sub-arrays as buffers in nonblocking calls may be invalid. See Section 19.1.6 for details.
- Declare each argument with an INTENT of IN, OUT, or INOUT as defined in this standard.

### Unofficial Draft for Comment Only

 $\frac{4}{5}$ 

 31 

Rationale. For these definitions in the mpi_f08 bindings, in most cases, INTENT(IN) is used if the C interface uses call-by-value. For all buffer arguments and for OUT and INOUT dummy arguments that allow one of the non-ordinary Fortran constants (see MPI_BOTTOM, etc. in Section 2.5.4) as input, an INTENT is not specified. (End of rationale.)

Advice to users. If a dummy argument is declared with INTENT(OUT), then the Fortran standard stipulates that the actual argument becomes undefined upon invocation of the MPI routine, i.e., it may be overwritten by some other values, e.g. zeros; according to [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., COMM_COPY_ATTR_FUNCTION) and predefined callbacks (e.g., MPI_COMM_NULL_COPY_FN).

*Rationale.* For user-defined callback functions (e.g., COMM_COPY_ATTR_FUNCTION) and their predefined callbacks (e.g., MPI_COMM_NULL_COPY_FN), the ierror argument is not optional. The MPI library must always call these routines with an actual ierror argument. Therefore, these user-defined functions need not check whether the MPI library calls these routines with or without an actual ierror output argument. (*End of rationale.*)

The MPI Fortran bindings in the mpi_f08 module are designed based on the Fortran 2008 standard [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..

### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

For an MPI implementation that fully supports nonblocking calls Advice to users. with the ASYNCHRONOUS attribute for choice buffers, an existing MPI-2.2 application may fail to compile even if it compiled and executed with expected results with an MPI-2.2 implementation. One reason may be that the application uses "contiguous" but not "simply contiguous" ASYNCHRONOUS arrays as actual arguments for choice buffers of nonblocking routines, e.g., by using subscript triplets with stride one or specifying (1:n) for a whole dimension instead of using (:). This should be fixed to fulfill the Fortran constraints for ASYNCHRONOUS dummy arguments. This is not considered a violation of backward compatibility because existing applications can not use the ASYNCHRONOUS attribute to protect nonblocking calls. Another reason may be that the application does not conform either to MPI-2.2, or to MPI-3.0, or to the Fortran standard, typically because the program forces the compiler to perform copy-in/out for a choice buffer argument in a nonblocking MPI call. This is also not a violation of backward compatibility because the application itself is non-conforming. See Section 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 non-standardized 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., non-contiguous sub-arrays can be used as buffers in nonblocking routines.
  - Set the MPI_SUBARRAYS_SUPPORTED compile-time constant to .FALSE. and declare choice buffers with a compiler-dependent mechanism that overrides type checking if the underlying Fortran compiler does not support the TS 29113 assumed-type and assumed-rank features. In this case, the use of non-contiguous sub-arrays in non-blocking calls may be disallowed. See Section 19.1.6 for details.

An MPI implementation may provide other features in the mpi module that enhance the usability of MPI while maintaining adherence to the standard. For example, it may provide INTENT information in these interface blocks.

Advice to implementors. The appropriate INTENT may be different from what is given in the MPI language-neutral bindings. Implementations must choose INTENT so that the function adheres to the MPI standard, e.g., by defining the INTENT as provided in the mpi_f08 bindings. (End of advice to implementors.)

Rationale. The intent given by the MPI generic interface is not precisely defined
 and does not in all cases correspond to the correct Fortran INTENT. For instance,
 receiving into a buffer specified by a datatype with absolute addresses may require
 associating MPI_BOTTOM with a dummy OUT argument. Moreover, "constants" such
 MPI_BOTTOM and MPI_STATUS_IGNORE are not constants as defined by Fortran,
 but "special addresses" used in a nonstandard way. Finally, the MPI-1 generic intent

### Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21 22

23

24

25

26

27 28

29

30

31

32

33 34

35

36

37 38

39

40

41

was changed in several places in MPI-2. For instance, MPI_IN_PLACE changes the intent of an OUT argument to be INOUT. (*End of rationale.*)

Advice to implementors. The Fortran 2008 standard illustrates in its Note 5.17 that "INTENT(OUT) means that the value of the argument after invoking the procedure is entirely the result of executing that procedure. If an argument should retain its value rather than being redefined, INTENT(INOUT) should be used rather than INTENT(OUT), even if there is no explicit reference to the value of the dummy argument. Furthermore, INTENT(INOUT) is not equivalent to omitting the IN-TENT attribute, because INTENT(INOUT) always requires that the associated actual argument is definable." Applications that include mpif.h may not expect that INTENT(OUT) is used. In particular, output array arguments are expected to keep their 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.

### Unofficial Draft for Comment Only

 $\overline{7}$ 

 $46 \\ 47$ 

	790 CHAPTER 19. LANGUAGE BINDINGS		
1 2	<ul> <li>Declaration of a status as an INTEGER variable instead of an INTEGER array with size MPI_STATUS_SIZE.</li> </ul>		
$\frac{3}{4}$	<ul> <li>Incorrect argument positions; e.g., interchanging the count and datatype arguments.</li> </ul>		
5 6	<ul> <li>Passing incorrect MPI handles; e.g., passing a datatype instead of a commu- nicator.</li> </ul>		
7 8 9 10	• 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.		
11 12 13	• 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.		
14 15	(End of advice to users.)		
16 17 18 19 20	Rationale. With MPI-3.0, the mpif.h include file was not deprecated in order to retain strong backward compatibility. Internally, mpif.h and the mpi module may be implemented so that essentially the same library implementation of the MPI routines can be used. ( <i>End of rationale.</i> )		
21 22	19.1.5 Interface Specifications, Procedure Names, and the Profiling Interface		
23 24 25 26 27 28 29 30 31 32 33 34	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. <i>Rationale.</i> This section was introduced in MPI-3.0 on Sep. 21, 2012. The major goals for implementing the three Fortran support methods have been:		
35 36	• Portable implementation of the wrappers from the MPI Fortran interfaces to the		
37 38 39	<ul> <li>MPI routines in C.</li> <li>Binary backward compatible implementation path when switching MPI_SUBARRAYS_SUPPORTED from .FALSE. to .TRUE</li> </ul>		
40 41 42 43	• 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.		
43 44	• No performance drawbacks.		
45	• Consistency between all three Fortran support methods.		
46 47 48	• Consistent with Fortran $2008 + TS 29113$ .		

No.	Specific pro- cedure name	Calling convention
1A	MPI_Isend_f08	Fortran interface and arguments, as in Annex A.4, except that in routines with a choice buffer dummy argument, this dummy argument is implemented with non-standard extensions like <b>!\$PRAGMA IGNORE_TKR</b> , which provides a call-by-reference argument without type, kind, and dimension checking.
1B	MPI_Isend_f08ts	Fortran interface and arguments, as in Annex A.4, but only for routines with one or more choice buffer dummy arguments; these dummy arguments are implemented with TYPE(*), DIMENSION().
2A	MPI_ISEND	Fortran interface and arguments, as in Annex A.5, except that in routines with a choice buffer dummy argument, this dummy argument is implemented with non-standard extensions like <b>!\$PRAGMA IGNORE_TKR</b> , which provides a call-by-reference argument without type, kind, and dimension checking.
2B	MPI_ISEND_FTS	Fortran interface and arguments, as in Annex A.5, but only for routines with one or more choice buffer dummy arguments; these dummy arguments are implemented with TYPE(*), DIMENSION(). In mpif.h only, the postfix "_FTS" for MPI_NEIGHBOR_ALLGATHERV_INIT, MPI_NEIGHBOR_ALLTOALLV_INIT, and MPI_NEIGHBOR_ALLTOALLW_INIT is shortened to "_F".

Table 19.1: Specific Fortran procedure names and related calling conventions. MPI_ISEND is used as an example. For routines without choice buffers, only 1A and 2A apply.

The design expected that all dummy arguments in the MPI Fortran interfaces are interoperable with C according to Fortran 2008 + TS 29113. This expectation was not fulfilled. The LOGICAL arguments are not interoperable with C, mainly because the internal representations for .FALSE. and .TRUE. are compiler dependent. The provided interface was mainly based on BIND(C) interfaces and therefore inconsistent with Fortran. To be consistent with Fortran, the BIND(C) had to be removed from the callback procedure interfaces and the predefined callbacks, e.g., MPI_COMM_DUP_FN. Non-BIND(C) procedures are also not interoperable with C, and therefore the BIND(C) had to be removed from all routines with PROCEDURE arguments, e.g., from MPI_OP_CREATE.

Therefore, this section was rewritten as an erratum to MPI-3.0. (*End of rationale.*)

A Fortran call to an MPI routine shall result in a call to a procedure with one of the specific procedure names and calling conventions, as described in Table 19.1. Case is not significant in the names.

Note that for the deprecated routines in Section 16.1, which are reported only in Annex A.5, scheme 2A is utilized in the mpi module and mpif.h, and also in the mpi_f08

#### Unofficial Draft for Comment Only

1 module.

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

 $\overline{7}$ 

9

10

11

12

27

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.

6 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 8 multiple implementation schemes from Table 19.1.

Advice to implementors. This may be desirable for backwards binary compatibility in the scope of a single MPI implementation, for example. (End of advice to imple*mentors.*)

13 After a compiler provides the facilities from TS 29113, i.e., TYPE(*), Rationale. 14DIMENSION(...), it is possible to change the bindings within a Fortran support method 15to support subarrays without recompiling the complete application provided that the 16previous interfaces with their specific procedure names are still included in the li-17 brary. Of course, only recompiled routines can benefit from the added facilities. 18 There is no binary compatibility conflict because each interface uses its own spe-19 cific procedure names and all interfaces use the same constants (except the value of 20MPI_SUBARRAYS_SUPPORTED and MPI_ASYNC_PROTECTS_NONBLOCKING) and type 21definitions. After a compiler also ensures that buffer arguments of nonblocking MPI 22 operations can be protected through the ASYNCHRONOUS attribute, and the proce-23dure declarations in the mpi_f08 and mpi module and the mpif.h include file declare 24choice buffers with the ASYNCHRONOUS attribute, then the value of

- 25MPI_ASYNC_PROTECTS_NONBLOCKING can be switched to .TRUE. in the module def-26inition and include file. (End of rationale.)
- Partial recompilation of user applications when upgrading MPI Advice to users. 28implementations is a highly complex and subtle topic. Users are strongly advised to 29 consult their MPI implementation's documentation to see exactly what is—and what 30 is not—supported. (End of advice to users.) 31

32 Within the mpi_f08 and mpi modules and mpif.h, for all MPI procedures, a second 33 procedure with the same calling conventions shall be supplied, except that the name is 34 modified by prefixing with the letter "P", e.g., PMPI_lsend. The specific procedure names 35 for these PMPI_XXXX procedures must be different from the specific procedure names for 36 the MPI_Xxxx procedures and are not specified by this standard.

37 A user-written or middleware profiling routine should provide the same specific Fortran 38 procedure names and calling conventions, and therefore can interpose itself as the MPI 39 library routine. The profiling routine can internally call the matching

MPI_{COMM|WIN|TYPE}_{SET|GET}_ATTR). In this case, the profiling software should 43 invoke the corresponding PMPI routine using the same Fortran support method as used in 44the calling application program, because the C, mpi_f08 and mpi callback prototypes are 45different or the meaning of the choice buffer or attribute_val arguments are different. 46

47Advice to users. Although for each support method and MPI routine (e.g., 48 MPI_ISEND in mpi_f08), multiple routines may need to be provided to intercept

⁴⁰ PMPI routine with any of its existing bindings, except for routines that have callback routine 41 dummy arguments, choice buffer arguments, or that are attribute caching routines ( 42

the specific procedures in the MPI library (e.g., MPI_lsend_f08 and MPI_lsend_f08ts), each profiling routine itself uses only one support method (e.g., mpi_f08) and calls the real MPI routine through the one PMPI routine defined in this support method (i.e., PMPI_lsend in this example). (*End of advice to users.*)

Advice to implementors. If all of the following conditions are fulfilled:

- the handles in the mpi_f08 module occupy one Fortran numerical storage unit (same as an INTEGER handle),
- the internal argument passing mechanism used to pass an actual ierror argument to a non-optional ierror dummy argument is binary compatible to passing an actual ierror argument to an ierror dummy argument that is declared as OPTIONAL,
- the internal argument passing mechanism for ASYNCHRONOUS and non-ASYNCHRONOUS arguments is the same,
- the internal routine call mechanism is the same for the Fortran and the C compilers for which the MPI library is compiled,
- the compiler does not provide TS 29113,

then the implementor may use the same internal routine implementations for all Fortran support methods but with several different specific procedure names. If the accompanying Fortran compiler supports TS 29113, then the new routines are needed only for routines with choice buffer arguments. (*End of advice to implementors.*)

Advice to implementors. In the Fortran support method mpif.h, compile-time argument checking can be also implemented for all routines. For mpif.h, the argument names are not specified through the MPI standard, i.e., only positional argument lists are defined, and not key-word based lists. Due to the rule that mpif.h must be valid for fixed and free source form, the subroutine declaration is restricted to one line with 72 characters. To keep the argument lists short, each argument name can be shortened to a minimum of one character. With this, the 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

TNTERFACE	PMPI_NEIGHBOR_ALLTOALLW_INIT	41
		42
SOBROOTINE	<pre>PMPI_NEIGHBOR_ALLTOALLW_INIT_F(a,b,c,d,e,f,g,h,i,j,j,k)</pre>	49
INTERFACE	PMPI_NEIGHBOR_ALLGATHERV_INIT	45
SUBROUTINE	<pre>PMPI_NEIGHBOR_ALLGATHERV_INIT_F(a,b,c,d,e,f,g,h,i,j,k)</pre>	44
	PMPI_RGET_ACCUMULATE	45
		46
SORKOOLINE	<pre>PMPI_RGET_ACCUMULATE_FTS(a,b,c,d,e,f,g,h,i,j,k,l,m,n)</pre>	47
		·± /

 31 

1		with 72, 71, and 70 characters. In principle, continuation lines would be possible
2		n mpif.h (spaces in columns 73–131, & in column 132, and in column 6 of the
3		continuation line) but this would not be valid if the source line length is extended
4		with a compiler flag to 132 characters. Column 133 is also not available for the
5		continuation character because lines longer than 132 characters are invalid with some
6		compilers by default.
7		
8		The longest specific procedure name is PMPI_Reduce_scatter_block_init_c_f08ts with
9		88 characters in the mpi_f08 module.
10		For example, the interface specifications together with the specific procedure names
11	С	an be implemented with
12	М	IODULE mpi_f08
13		TYPE, BIND(C) :: MPI_Comm
14		INTEGER :: MPI_VAL
15		END TYPE MPI_Comm
16		
17		<pre>INTERFACE MPI_Comm_rank ! (as defined in Chapter 6) SUBROUTINE MPI_Comm_rank_f08(comm, rank, ierror)</pre>
18		IMPORT :: MPI_Comm
19		TYPE(MPI_Comm), INTENT(IN) :: comm
20		INTEGER, INTENT(OUT) :: rank
21		INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22		END SUBROUTINE
23	F	END INTERFACE END MODULE mpi_f08
24	L	WD WDDOFF mb1-100
25	Μ	IODULE mpi
26		INTERFACE MPI_Comm_rank ! (as defined in Chapter 6)
27		SUBROUTINE MPI_Comm_rank(comm, rank, ierror)
28		INTEGER, INTENT(IN) :: comm ! The INTENT may be added although
29		INTEGER, INTENT(OUT) :: rank ! it is not defined in the INTEGER, INTENT(OUT) :: ierror ! official routine definition.
30		END SUBROUTINE
31 32		END INTERFACE
33	E	IND MODULE mpi
34		
35		And if interfaces are provided in mpif.h, they might look like this (outside of any nodule and in fixed source format):
36	1.	noutre and in fixed source format).
37	!	23456789012345678901234567890123456789012345678901234567890123456789012
38		INTERFACE MPI_Comm_rank ! (as defined in Chapter 6)
39		SUBROUTINE MPI_Comm_rank(comm, rank, ierror)
40		INTEGER, INTENT(IN) :: comm ! The argument names may be
41		INTEGER, INTENT(OUT) :: rank  ! shortened so that the INTEGER, INTENT(OUT) :: ierror ! subroutine line fits to the
42		END SUBROUTINE ! maximum of 72 characters.
43		END INTERFACE
44		
45	(	End of advice to implementors.)
46		Advise to users. The following is an exemple of how a very switter or with
47		<i>Advice to users.</i> The following is an example of how a user-written or middleware profiling routine can be implemented:
48	ł	stomme round our be implemented.

CHAPTER 19. LANGUAGE BINDINGS

SUBROUTINE MPI_Isend_f08ts(buf,count,datatype,dest,tag,comm,request,ierror)						
USE :: mpi_f08, my_noname => MPI_Isend_f08ts						
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf						
INTEGER, INTENT(IN) :: count, dest, tag						
TYPE(MPI_Datatype), INTENT(IN) :: datatype						
TYPE(MPI_Comm), INTENT(IN) :: comm						
TYPE(MPI_Request), INTENT(OUT) :: request						
INTEGER, OPTIONAL, INTENT(OUT) :: ierror						
! some code for the begin of profiling						
call PMPI_Isend (buf, count, datatype, dest, tag, comm, request, ierror)						
! some code for the end of profiling						
END SUBROUTINE MPI_Isend_f08ts						

Note that this routine is used to intercept the existing specific procedure name MPI_lsend_f08ts in the MPI library. This routine must not be part of a module. This routine itself calls PMPI_lsend. The USE of the mpi_f08 module is needed for definitions of handle types and the interface for PMPI_lsend. However, this module also contains an interface definition for the specific procedure name MPI_lsend_f08ts that conflicts with the definition of this profiling routine (i.e., the name is doubly defined). Therefore, the USE here specifically excludes the interface from the module by renaming the unused routine name in the mpi_f08 module into "my_noname" in the scope of this routine. (*End of advice to users.*)

The PMPI interface allows intercepting MPI routines. For exam-Advice to users. ple, an additional MPI_ISEND profiling wrapper can be provided that is called by the application and internally calls PMPI_ISEND. There are two typical use cases: a profiling layer that is developed independently from the application and the MPI library, and profiling routines that are part of the application and have access to the application data. With MPI-3.0, new Fortran interfaces and implementation schemes were introduced that have several implications on how Fortran MPI routines are internally implemented and optimized. For profiling layers, these schemes imply that several internal interfaces with different specific procedure names may need to be intercepted, as shown in the example code above. Therefore, for wrapper routines that are part of a Fortran application, it may be more convenient to make the name shift within the application, i.e., to substitute the call to the MPI routine (e.g., MPI_ISEND) by a call to a user-written profiling wrapper with a new name (e.g., X_MPI_ISEND) and to call the Fortran MPI_ISEND from this wrapper, instead of using the PMPI interface. (End of advice to users.)

Advice to implementors. An implementation that provides a Fortran interface must provide a combination of MPI library and module or include file that uses the specific procedure names as described in Table 19.1 so that the MPI Fortran routines are interceptable as described above. (*End of advice to implementors.*)

## 19.1.6 MPI for Different Fortran Standard Versions

This section describes which Fortran interface functionality can be provided for different versions of the Fortran standard.

• For Fortran 77 with some extensions:

#### Unofficial Draft for Comment Only

 24 

 $46 \\ 47$ 

1	- MPI identifiers may be up to 30 characters (31 with the profiling interface).
2	– MPI identifiers may contain underscores after the first character.
3	- An MPI subroutine with a choice argument may be called with different argument
5	types.
6	- Although not required by the $MPI$ standard, the <code>INCLUDE</code> statement should be
7	available for including mpif.h into the user application source code.
8	Only MPI-1.1, MPI-1.2, and MPI-1.3 can be implemented. The use of absolute ad-
9 10	dresses from MPI_ADDRESS and MPI_BOTTOM may cause problems if an address
11	does not fit into the memory space provided by an INTEGER. (In MPI-2.0 this problem
12	is solved with MPI_GET_ADDRESS, but not for Fortran 77.)
13	• For Fortran 90:
14	The major additional features that are needed from Fortran 90 are:
15 16	- The MODULE and INTERFACE concept.
17	- The KIND= and SELECTED_XXX_KIND concept.
18	- Fortran derived TYPEs and the SEQUENCE attribute.
19 20	- The OPTIONAL attribute for dummy arguments.
20	- Cray pointers, which are a non-standard compiler extension, are needed for the
22	use of MPI_ALLOC_MEM.
23	
24	With these features, MPI-1.1 – MPI-2.2 can be implemented without restrictions. MPI-3.0 can be implemented with some restrictions. The Fortran support methods
25 26	are abbreviated with $S1 = \text{the mpi}_f08 \text{ module}$ , $S2 = \text{the mpi} \text{ module}$ , and $S3 = \text{the}$
27	mpif.f include file. If not stated otherwise, restrictions exist for each method which
28	prevent implementing the complete semantics of MPI-3.0.
29	MDL CUDADDAYC CUDDODTED single EALGE is subscript trights and some
30	<ul> <li>MPI_SUBARRAYS_SUPPORTED equals .FALSE., i.e., subscript triplets and non- contiguous subarrays cannot be used as buffers in nonblocking routines, RMA,</li> </ul>
31 32	or split-collective I/O.
33	- S1, S2, and S3 can be implemented, but for S1, only a preliminary implementa-
34	tion is possible.
35	– In this preliminary interface of S1, the following changes are necessary:
36	* TYPE(*), DIMENSION() is substituted by non-standardized extensions
37 38	like !\$PRAGMA IGNORE_TKR.
39	* The ASYNCHRONOUS attribute is omitted.
40	* PROCEDURE() callback declarations are substituted by EXTERNAL.
41	- The specific procedure names are specified in Section 19.1.5.
42	- Due to the rules specified in Section 19.1.5, choice buffer declarations should be
43 44	implemented only with non-standardized extensions like <b>!\$PRAGMA IGNORE_TKR</b>
45	(as long as $F2008+TS$ 29113 is not available).
46	In S2 and S3: Without such extensions, routines with choice buffers should be
47	provided with an implicit interface, instead of overloading with a different MPI
48	function for each possible buffer type (as mentioned in Section 19.1.11). Such

overloading would also imply restrictions for passing Fortran derived types as choice buffer, see also Section 19.1.15.

Only in S1: The implicit interfaces for routines with choice buffer arguments imply that the ierror argument cannot be defined as OPTIONAL. For this reason, it is recommended not to provide the  $mpi_f08$  module if such an extension is not available.

- The ASYNCHRONOUS attribute can not be used in applications to protect buffers in nonblocking MPI calls (S1–S3).
- The TYPE(C_PTR) binding of the MPI_ALLOC_MEM and MPI_WIN_ALLOCATE routines is not available.
- In S1 and S2, the definition of the handle types (e.g., TYPE(MPI_Comm) and the status type TYPE(MPI_Status) must be modified: The SEQUENCE attribute must be used instead of BIND(C) (which is not available in Fortran 90/95). This restriction implies that the application must be fully recompiled if one switches to an MPI library for Fortran 2003 and later because the internal memory size of the handles may have changed. For this reason, an implementor may choose not to provide the mpi_f08 module for Fortran 90 compilers. In this case, the mpi_f08 handle types and all routines, constants and types related to TYPE(MPI_Status) (see Section 19.3.5) are also not available in the mpi module and mpif.h.

handle types and all routines, constants and types related to TYPE(MPI_Status)	19
(see Section 19.3.5) are also not available in the mpi module and mpif.h.	20
	21
For Fortran 95:	22
The quality of the MPI interface and the restrictions are the same as with Fortran 90.	23
For Fortran 2003:	24
	25
The major features that are needed from Fortran 2003 are:	26
– Interoperability with C, i.e.,	27
* BIND(C) derived types.	28
	29
* The ISO_C_BINDING intrinsic type C_PTR and routine C_F_POINTER.	30
- The ability to define an ABSTRACT $$ INTERFACE and to use it for PROCEDURE dummy	31
arguments.	32
- The ability to overload the operators .EQ. and .NE. to allow the comparison of	33
derived types (used in MPI-3.0 for MPI handles).	34
- The ASYNCHRONOUS attribute is available to protect Fortran asynchronous I/O.	35
This feature is not yet used by MPI, but it is the basis for the enhancement for	36
MPI communication in the TS 29113.	37
Wit i communication in the 15 25115.	38
With these features (but still without the features of TS 29113), MPI-1.1 – MPI-2.2	39
can be implemented without restrictions, but with one enhancement:	40
	41
$-$ The user application can use <code>TYPE(C_PTR)</code> together with <code>MPI_ALLOC_MEM</code> as	42
long as $MPI_ALLOC_MEM$ is defined with an implicit interface because a $C_PTR$	43
and an $INTEGER(KIND=MPI_ADDRESS_KIND)$ argument must both map to a	44
void * argument.	45
	46

MPI-3.0 can be implemented with the following restrictions:

- MPI_SUBARRAYS_SUPPORTED equals .FALSE..

 $\mathbf{2}$ 

1	- For S1, only a preliminary implementation is possible. The following changes are
2	necessary:
3	* TYPE(*), DIMENSION() is substituted by non-standardized extensions
4	like !\$PRAGMA IGNORE_TKR.
5	- The specific procedure names are specified in Section 19.1.5.
6	
7	- With S1, the ASYNCHRONOUS is required as specified in the second Fortran inter-
8	faces. With $S2$ and $S3$ the implementation can also add this attribute if explicit
9	interfaces are used.
10	- The ASYNCHRONOUS Fortran attribute can be used in applications to try to protect
11	buffers in nonblocking MPI calls, but the protection can work only if the compiler
12	is able to protect asynchronous Fortran $I/O$ and makes no difference between such
13	asynchronous Fortran I/O and MPI communication.
14	- The TYPE(C_PTR) binding of the MPI_ALLOC_MEM, MPI_WIN_ALLOCATE,
15	MPI_WIN_ALLOCATE_SHARED, and MPI_WIN_SHARED_QUERY routines can
16	be used only for Fortran types that are C compatible.
17	
18	- The same restriction as for Fortran 90 applies if non-standardized extensions like
19	!\$PRAGMA IGNORE_TKR are not available.
20	• For Fortran $2008 + TS 29113$ and later and
21	For Fortran $2003 + TS 29113$ :
22	The major feature that are needed from TS 29113 are:
23	
24	- TYPE(*), DIMENSION() is available.
25	- The ASYNCHRONOUS attribute is extended to protect also nonblocking MPI com-
26	munication.
27	- The array dummy argument of the ISO_C_BINDING intrinsic C_F_POINTER is not
28	restricted to Fortran types for which a corresponding type in C exists.
29 30	resoluted to rotatian types for which a corresponding type in C exists.
31	Using these features, $MPI-3.0$ can be implemented without any restrictions.
32	With S1 MDI CURADRAYS SUDDORTED ocuals TRUE The
33	- With S1, MPI_SUBARRAYS_SUPPORTED equals .TRUE The ASYNCHRONOUS attribute can be used to protect buffers in nonblocking MPI calls.
34	The TYPE(C_PTR) binding of the MPI_ALLOC_MEM, MPI_WIN_ALLOCATE,
35	MPI_WIN_ALLOCATE_SHARED, and MPI_WIN_SHARED_QUERY routines can
36	be used for any Fortran type.
37	
38	- With S2 and S3, the value of MPI_SUBARRAYS_SUPPORTED is implementation
39	dependent. A high quality implementation will also provide
40	MPI_SUBARRAYS_SUPPORTED set to .TRUE. and will use the ASYNCHRONOUS at-
41	tribute in the same way as in S1.
42	$-$ If non-standardized extensions like <b>!\$PRAGMA IGNORE_TKR</b> are not available then
43	S2 must be implemented with TYPE(*), DIMENSION().
44	
45	Advice to implementors. If $MPI_SUBARRAYS_SUPPORTED ==.FALSE.$ , the choice
46	argument may be implemented with an explicit interface using compiler directives,
47	for example:
48	

```
INTERFACE
SUBROUTINE MPI_...(buf, ...)
!DEC$ ATTRIBUTES NO_ARG_CHECK :: buf
!$PRAGMA IGNORE_TKR buf
!DIR$ IGNORE_TKR buf
!IBM* IGNORE_TKR buf
REAL, DIMENSION(*) :: buf
... ! declarations of the other arguments
END SUBROUTINE
END INTERFACE
```

(End of advice to implementors.)

## 19.1.7 Requirements on Fortran Compilers

MPI-3.0 (and later) compliant Fortran bindings are not only a property of the MPI library itself, but rather a property of an MPI library together with the Fortran compiler suite for which it is compiled.

Advice to users. Users must take appropriate steps to ensure that proper options are specified to compilers. MPI libraries must document these options. Some MPI libraries are shipped together with special compilation scripts (e.g., mpif90, mpicc) that set these options automatically. (End of advice to users.)

An MPI library together with the Fortran compiler suite is only compliant with MPI-3.0 (and later), as referred by MPI_GET_VERSION, if all the solutions described in Sections 19.1.11 through 19.1.19 work correctly. Based on this rule, major requirements for all three Fortran support methods (i.e., the mpi_f08 and mpi modules, and mpif.h) are:

- The language features assumed-type and assumed-rank from Fortran 2008 TS 29113 [46] are available. This is required only for mpi_f08. As long as this requirement is not supported by the compiler, it is valid to build an MPI library that implements the mpi_f08 module with MPI_SUBARRAYS_SUPPORTED set to .FALSE..
- "Simply contiguous" arrays and scalars must be passed to choice buffer dummy arguments of nonblocking routines with call by reference. This is needed only if one of the support methods does not use the ASYNCHRONOUS attribute. See Section 19.1.12 for more details.
- SEQUENCE and BIND(C) derived types are valid as actual arguments passed to choice buffer dummy arguments, and, in the case of MPI_SUBARRAYS_SUPPORTED== .FALSE., they are passed with call by reference, and passed by descriptor in the case of .TRUE..
- All actual arguments that are allowed for a dummy argument in an implicitly defined and separately compiled Fortran routine with the given compiler (e.g., CHARACTER(LEN=*) strings and array of strings) must also be valid for choice buffer dummy arguments with all Fortran support methods.
- The array dummy argument of the ISO_C_BINDING intrinsic module procedure C_F_POINTER is not restricted to Fortran types for which a corresponding type in C exists.

## Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 24 

 $45 \\ 46$ 

• The Fortran compiler shall not provide TYPE(*) unless the ASYNCHRONOUS attribute protects MPI communication as described in TS 29113. Specifically, the TS 29113 must be implemented as a whole.

The following rules are required at least as long as the compiler does not provide the extension of the ASYNCHRONOUS attribute as part of TS 29113 and there still exists a Fortran 6 support method with MPI_ASYNC_PROTECTS_NONBLOCKING set to .FALSE.. Observation of these rules by the MPI application developer is especially recommended for backward compatibility of existing applications that use the mpi module or the mpif.h include file. The rules are as follows: 10

- Separately compiled empty Fortran routines with implicit interfaces and separately compiled empty C routines with BIND(C) Fortran interfaces (e.g., MPI_F_SYNC_REG on page 822 and Section 19.1.8, and DD on page 823) solve the problems described in Section 19.1.17.
- The problems with temporary data movement (described in detail in Section 19.1.18) 16are solved as long as the application uses different sets of variables for the nonblocking 17 communication (or nonblocking or split collective I/O) and the computation when overlapping communication and computation. 19
  - 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-3.0 are beyond the scope of TS 29113. (End of advice to *implementors.*)

#### Additional Support for Fortran Register-Memory-Synchronization 19.1.8

34As described in Section 19.1.17, a dummy call may be necessary to tell the compiler that 35 registers are to be flushed for a given buffer or that accesses to a buffer may not be moved 36 across a given point in the execution sequence. Only a Fortran binding exists for this call. 37

```
39
 MPI_F_SYNC_REG(buf)
40
 INOUT
 buf
 initial address of buffer (choice)
41
42
 Fortran 2008 binding
43
 MPI_F_sync_reg(buf)
44
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
45
46
 Fortran binding
47
 MPI_F_SYNC_REG(BUF)
48
 <type> BUF(*)
```

1

 $\mathbf{2}$ 

3

4

5

7

8

9

11

12

13

14

15

18

20

21

22

23

2425

26

27

28

29

30

 31 32

33

This routine has no executable statements. It must be compiled in the MPI library in such a manner that a Fortran compiler cannot detect in the module that the routine has an empty body. It is used only to force the compiler to flush a cached register value of a variable or buffer back to memory (when necessary), or to invalidate the register value.

*Rationale.* This function is not available in other languages because it would not be useful. This routine has no ierror return argument because there is no operation that can fail. (*End of rationale.*)

Advice to implementors. This routine can be bound to a C routine to minimize the risk that the Fortran compiler can learn that this routine is empty (and that the call to this routine can be removed as part of an optimization). However, it is explicitly allowed to implement this routine within the mpi_f08 module according to the definition for the mpi module or mpif.h to circumvent the overhead of building the internal dope vector to handle the assumed-type, assumed-rank argument. (End of advice to implementors.)

Rationale. This routine is not defined with TYPE(*), DIMENSION(*), i.e., assumed size instead of assumed rank, because this would restrict the usability to "simply contiguous" arrays and would require overloading with another interface for scalar arguments. (*End of rationale.*)

Advice to users. If only a part of an array (e.g., defined by a subscript triplet) is used in a nonblocking routine, it is recommended to pass the whole array to MPI_F_SYNC_REG anyway to minimize the overhead of this no-operation call. Note that this routine need not be called if MPI_ASYNC_PROTECTS_NONBLOCKING is .TRUE. and the application fully uses the facilities of ASYNCHRONOUS arrays. (*End of advice to users*.)

## 19.1.9 Additional Support for Fortran Numeric Intrinsic Types

MPI provides a small number of named datatypes that correspond to named intrinsic types supported by C and Fortran. These include MPI_INTEGER, MPI_REAL, MPI_INT, MPI_DOUBLE, etc., as well as the optional types MPI_REAL4, MPI_REAL8, etc. There is a one-to-one correspondence between language declarations and MPI types.

Fortran (starting with Fortran 90) provides so-called KIND-parameterized types. These 35 types are declared using an intrinsic type (one of INTEGER, REAL, COMPLEX, LOGICAL, and 36 CHARACTER) with an optional integer KIND parameter that selects from among one or more 37 variants. The specific meaning of different KIND values themselves are implementation 38 dependent and not specified by the language. Fortran provides the KIND selection functions 39 selected_real_kind for REAL and COMPLEX types, and selected_int_kind for INTEGER 40 types that allow users to declare variables with a minimum precision or number of digits. 41 These functions provide a portable way to declare KIND-parameterized REAL, COMPLEX, and 42INTEGER variables in Fortran. This scheme is backward compatible with Fortran 77. REAL 43 and INTEGER Fortran variables have a default KIND if none is specified. Fortran DOUBLE 44PRECISION variables are of intrinsic type REAL with a non-default KIND. The following two 45declarations are equivalent: 46

double precision x
real(KIND(0.0d0)) x

1

 $\mathbf{2}$ 

3

4

5

6

7

8 9

10

11

12

13

14

15

16 17

18

19

20

21

22

23

 24 

25

26

27 28

29 30

 31 

32

33

34

47

1 MPI provides two orthogonal methods for handling communication buffers of numeric  $\mathbf{2}$ intrinsic types. The first method (see the following section) can be used when variables have 3 been declared in a portable way—using default KIND or using KIND parameters obtained 4 with the selected_int_kind or selected_real_kind functions. With this method, MPI  $\mathbf{5}$ automatically selects the correct data size (e.g., 4 or 8 bytes) and provides representation 6 conversion in heterogeneous environments. The second method (see "Support for size- $\overline{7}$ specific MPI Datatypes" on page 806) gives the user complete control over communication 8 by exposing machine representations.

9 10

11

28

29

30

31 32

33

34

35

36

37

38

39

## Parameterized Datatypes with Specified Precision and Exponent Range

¹² MPI provides named datatypes corresponding to standard Fortran 77 numeric types:

 $_{\rm 13}$   $\,$  MPI_INTEGER, MPI_COMPLEX, MPI_REAL, MPI_DOUBLE_PRECISION and

¹⁴ MPI_DOUBLE_COMPLEX. MPI automatically selects the correct data size and provides rep-¹⁵ resentation conversion in heterogeneous environments. The mechanism described in this ¹⁶ section extends this model to support portable parameterized numeric types.

The model for supporting portable parameterized types is as follows. Real variables 17are declared (perhaps indirectly) using selected_real_kind(p, r) to determine the KIND 18 parameter, where p is decimal digits of precision and r is an exponent range. Implicitly 19 MPI maintains a two-dimensional array of predefined MPI datatypes D(p, r). D(p, r) is 20defined for each value of (p, r) supported by the compiler, including pairs for which one 21value is unspecified. Attempting to access an element of the array with an index (p, r) not 22 supported by the compiler is erroneous. MPI implicitly maintains a similar array of COMPLEX 23datatypes. For integers, there is a similar implicit array related to selected_int_kind and 24indexed by the requested number of digits r. Note that the predefined datatypes contained 25in these implicit arrays are not the same as the named MPI datatypes MPI_REAL, etc., but 26a new set. 27

Advice to implementors. The above description is for explanatory purposes only. It is not expected that implementations will have such internal arrays. (End of advice to implementors.)

Advice to users. selected_real_kind() maps a large number of (p,r) pairs to a much smaller number of KIND parameters supported by the compiler. KIND parameters are not specified by the language and are not portable. From the language point of view intrinsic types of the same base type and KIND parameter are of the same type. In order to allow interoperability in a heterogeneous environment, MPI is more stringent. The corresponding MPI datatypes match if and only if they have the same (p,r) value (REAL and COMPLEX) or r value (INTEGER). Thus MPI has many more datatypes than there are fundamental language types. (End of advice to users.)

#### MPI_TYPE_CREATE_F90_REAL(p, r, newtype)

IN	р	precision, in decimal digits (integer)
IN	r	decimal exponent range (integer)
OUT	newtype	the requested MPI datatype (handle)

#### C binding

int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)

#### 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 805.

It is erroneous to supply values for p and r not supported by the compiler.

## MPI_TYPE_CREATE_F90_COMPLEX(p, r, newtype)

	E_CREATE_F90_COMPLEX(p	, r, newtype)	30
IN	р	precision, in decimal digits (integer)	31
IN	r	decimal exponent range (integer)	32
OUT			33
OUT	newtype	the requested MPI datatype (handle)	34
			35
C bindin	g		36
int MPI_7	<pre>Fype_create_f90_complex(i</pre>	nt p, int r, MPI_Datatype *newtype)	37
Fortran 2	2008 binding		38
	Ũ		39
• 1	_create_f90_complex(p, r,	newtype, lerror)	40
	GER, INTENT(IN) :: p, r		41
TYPE	(MPI_Datatype), INTENT(OU	T) :: newtype	42
INTEC	GER, OPTIONAL, INTENT(OUT	) :: ierror	
			43
Fortran b	binding		44
MPI_TYPE_	_CREATE_F90_COMPLEX(P, R,	NEWTYPE, IERROR)	45
INTEC	GER P, R, NEWTYPE, IERROR		46

 24 

```
1
 This function returns a predefined MPI datatype that matches a COMPLEX variable of
\mathbf{2}
 KIND selected_real_kind(p, r). Either p or r may be omitted from calls to
3
 selected_real_kind(p, r) (but not both). Analogously, either p or r may be set to
4
 MPI_UNDEFINED. Matching rules for datatypes created by this function are analogous to the
\mathbf{5}
 matching rules for datatypes created by MPI_TYPE_CREATE_F90_REAL. Restrictions on
6
 using the returned datatype with the "external32" data representation are given on page 805.
7
 It is erroneous to supply values for p and r not supported by the compiler.
8
9
 MPI_TYPE_CREATE_F90_INTEGER(r, newtype)
10
11
 IN
 decimal exponent range, i.e., number of decimal
 r
12
 digits (integer)
13
 OUT
 newtype
 the requested MPI datatype (handle)
14
15
 C binding
16
 int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)
17
18
 Fortran 2008 binding
19
 MPI_Type_create_f90_integer(r, newtype, ierror)
20
 INTEGER, INTENT(IN) :: r
21
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
22
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
 Fortran binding
24
 MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR)
25
 INTEGER R, NEWTYPE, IERROR
26
27
 This function returns a predefined MPI datatype that matches an INTEGER variable of
28
 KIND selected_int_kind(r). Matching rules for datatypes created by this function are
29
 analogous to the matching rules for datatypes created by MPI_TYPE_CREATE_F90_REAL.
30
 Restrictions on using the returned datatype with the "external32" data representation are
^{31}
 given on page 805.
32
 It is erroneous to supply a value for r that is not supported by the compiler.
33
 Example:
34
 integer
 longtype, quadtype
35
 integer, parameter :: long = selected_int_kind(15)
36
 integer(long) ii(10)
37
 real(selected_real_kind(30)) x(10)
38
 call MPI_TYPE_CREATE_F90_INTEGER(15, longtype, ierror)
39
 call MPI_TYPE_CREATE_F90_REAL(30, MPI_UNDEFINED, quadtype, ierror)
40
41
 . . .
42
 call MPI_SEND(ii, 10, longtype, ...)
43
 call MPI_SEND(x, 10, quadtype, ...)
44
45
 Advice to users.
 The datatypes returned by the above functions are predefined
46
 datatypes. They cannot be freed; they do not need to be committed; they can be
47
 used with predefined reduction operations. There are two situations in which they
48
```

behave differently syntactically, but not semantically, from the MPI named predefined datatypes.

- 1. MPI_TYPE_GET_ENVELOPE returns special combiners that allow a program to retrieve the values of p and r.
- 2. Because the datatypes are not named, they cannot be used as compile-time initializers or otherwise accessed before a call to one of the MPI_TYPE_CREATE_F90_XXX routines.

If a variable was declared specifying a non-default KIND value that was not obtained with selected_real_kind() or selected_int_kind(), the only way to obtain a matching MPI datatype is to use the size-based mechanism described in the next section. (*End of advice to users.*)

Advice to implementors. An application may often repeat a call to MPI_TYPE_CREATE_F90_XXX with the same combination of (XXX,p,r). The application is not allowed to free the returned predefined, unnamed datatype handles. To prevent the creation of a potentially huge amount of handles, a high quality MPI implementation should return the same datatype handle for the same (REAL/COMPLEX/ INTEGER,p,r) combination. Checking for the combination (p,r) in the preceding call to MPI_TYPE_CREATE_F90_XXX and using a hash table to find formerly generated handles should limit the overhead of finding a previously generated datatype with same combination of (XXX,p,r). (*End of advice to implementors.*)

*Rationale.* The MPI_TYPE_CREATE_F90_REAL/COMPLEX/INTEGER interface needs as input the original range and precision values to be able to define useful and compiler-independent external (Section 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
```

 31 

```
1
 For MPI_TYPE_CREATE_F90_COMPLEX: twice the size as for
\mathbf{2}
 MPI_TYPE_CREATE_F90_REAL.
3
 For MPI_TYPE_CREATE_F90_INTEGER:
4
 (r > 38) then external32 representation is undefined
 if
5
 else if (r > 18) then external32_size =
 16
6
 else if (r >
 9) then
 external32_size =
 8
7
 else if (r >
 4) then
 external32_size =
 4
8
 else if (r > 2) then
 external32_size =
 2
9
 else
 external32_size =
 1
10
 If the "external32" representation of a datatype is undefined, the result of using the datatype
11
 directly or indirectly (i.e., as part of another datatype or through a duplicated datatype)
12
 in operations that require the "external32" representation is undefined. These operations in-
13
 clude MPI_PACK_EXTERNAL, MPI_UNPACK_EXTERNAL, and many MPI_FILE functions,
14
 when the "external32" data representation is used. The ranges for which the "external32"
15
16
 representation is undefined are reserved for future standardization.
17
18
 Support for Size-specific MPI Datatypes
19
 MPI provides named datatypes corresponding to optional Fortran 77 numeric types that
20
 contain explicit byte lengths—MPI_REAL4, MPI_INTEGER8, etc. This section describes a
21
 mechanism that generalizes this model to support all Fortran numeric intrinsic types.
22
 We assume that for each typeclass (integer, real, complex) and each word size there is
23
 a unique machine representation. For every pair (type class, n) supported by a compiler,
24
 MPI must provide a named size-specific datatype. The name of this datatype is of the form
25
 MPI_<TYPE>n in C and Fortran where <TYPE> is one of REAL, INTEGER and COMPLEX, and
26
 n is the length in bytes of the machine representation. This datatype locally matches all
27
 variables of type (typeclass, n) in Fortran. The list of names for such types includes:
28
29
 MPI REAL4
30
 MPI_REAL8
^{31}
 MPI_REAL16
 MPI_COMPLEX8
32
33
 MPI_COMPLEX16
34
 MPI_COMPLEX32
35
 MPI_INTEGER1
36
 MPI_INTEGER2
37
 MPI_INTEGER4
38
 MPI_INTEGER8
39
 MPI_INTEGER16
40
 One datatype is required for each representation supported by the Fortran compiler.
41
42
 Rationale. Particularly for the longer floating-point types, C and Fortran may use
43
 different representations. For example, a Fortran compiler may define a 16-byte REAL
44
 type with 33 decimal digits of precision while a C compiler may define a 16-byte long
45
 double type that implements an 80-bit (10 byte) extended precision floating point
46
 value. Both of these types are 16 bytes long, but they are not interoperable. Thus,
47
 these types are defined by Fortran, even though C may define types of the same length.
48
 (End of rationale.)
```

To be backward compatible with the interpretation of these types in MPI-1, we assume that the nonstandard declarations REAL*n, INTEGER*n, always create a variable whose representation is of size n. These datatypes may also be used for variables declared with KIND=INT8/16/32/64 or KIND=REAL32/64/128, which are defined in the ISO_FORTRAN_ENV intrinsic module. Note that the MPI datatypes and the REAL*n, INTEGER*n declarations count bytes whereas the Fortran KIND values count bits. All these datatypes are predefined.

The following function allows a user to obtain a size-specific MPI datatype for any intrinsic Fortran type.

Μ	PI_IYPE	_MATCH_SIZE(typeclass, size,	datatype)
	IN	typeclass	generic type specifier (integer)
	IN	size	size, in bytes, of representation (integer)
	OUT	datatype	datatype with correct type, size (handle)

#### C binding

int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *datatype)
Fortran 2008 binding

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

#### Fortran binding

MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, DATATYPE, IERROR) INTEGER TYPECLASS, SIZE, DATATYPE, IERROR

typeclass is one of MPI_TYPECLASS_REAL, MPI_TYPECLASS_INTEGER and MPI_TYPECLASS_COMPLEX, corresponding to the desired **typeclass**. The function returns an MPI datatype matching a local variable of type (**typeclass**, **size**).

This function returns a reference (handle) to one of the predefined named datatypes, not a duplicate. This type cannot be freed. MPI_TYPE_MATCH_SIZE can be used to obtain a size-specific type that matches a Fortran numeric intrinsic type by first calling storage_size() in order to compute the variable size in bits, dividing it by eight, and then calling MPI_TYPE_MATCH_SIZE to find a suitable datatype. In C, one can use the C function sizeof() (which returns the size in bytes) instead of storage_size() (which returns the size in bits). In addition, for variables of default kind the variable's size can be computed by a call to MPI_TYPE_GET_EXTENT, if the typeclass is known. It is erroneous to specify a size not supported by the compiler.

*Rationale.* This is a convenience function. Without it, it can be tedious to find the correct named type. See note to implementors below. (*End of rationale.*)

Advice to implementors. This function could be implemented as a series of tests.

int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *rtype)
{

switch(typeclass) {

 $\overline{7}$ 

 $45 \\ 46$ 

```
1
 case MPI_TYPECLASS_REAL: switch(size) {
2
 case 4: *rtype = MPI_REAL4; return MPI_SUCCESS;
3
 case 8: *rtype = MPI_REAL8; return MPI_SUCCESS;
4
 default: error(...);
 }
5
6
 case MPI_TYPECLASS_INTEGER: switch(size) {
7
 case 4: *rtype = MPI_INTEGER4; return MPI_SUCCESS;
8
 case 8: *rtype = MPI_INTEGER8; return MPI_SUCCESS;
9
 default: error(...);
10
 }
11
 etc. ...
 . . .
12
 }
13
14
 return MPI_SUCCESS;
15
 }
16
 (End of advice to implementors.)
17
18
19
 Communication With Size-specific Types
20
 The usual type matching rules apply to size-specific datatypes: a value sent with datatype
21
 MPL_{TYPE>n} can be received with this same datatype on another process. Most modern
22
 computers use two's complement for integers and IEEE format for floating point. Thus,
23
 communication using these size-specific datatypes will not entail loss of precision or trun-
24
 cation errors.
25
26
 Advice to users. Care is required when communicating in a heterogeneous environ-
27
 ment. Consider the following code:
28
29
 real(selected_real_kind(5)) x(100)
30
 size = storage_size(x) / 8
31
 call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
32
 if (myrank .eq. 0) then
33
 ... initialize x ...
34
 call MPI_SEND(x, xtype, 100, 1, ...)
35
 else if (myrank .eq. 1) then
36
 call MPI_RECV(x, xtype, 100, 0, ...)
37
 endif
38
39
 This may not work in a heterogeneous environment if the value of size is not the
40
 same on process 1 and process 0. There should be no problem in a homogeneous
41
 environment. To communicate in a heterogeneous environment, there are at least four
42
 options. The first is to declare variables of default type and use the MPI datatypes
43
 for these types, e.g., declare a variable of type REAL and use MPI_REAL. The second
44
 is to use selected_real_kind or selected_int_kind and with the functions of the
45
 previous section. The third is to declare a variable that is known to be the same
46
 size on all architectures (e.g., selected_real_kind(12) on almost all compilers will
47
 result in an 8-byte representation). The fourth is to carefully check representation
48
 size before communication. This may require explicit conversion to a variable of size
```

that can be communicated and handshaking between sender and receiver to agree on a size.

Note finally that using the "external32" representation for I/O requires explicit attention to the representation sizes. Consider the following code:

```
real(selected_real_kind(5)) x(100)
size = storage_size(x) / 8
call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
if (myrank .eq. 0) then
 call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo',
 &
 MPI_MODE_CREATE+MPI_MODE_WRONLY,
 &
 MPI_INFO_NULL, fh, ierror)
 'external32',&
 call MPI_FILE_SET_VIEW(fh, zero, xtype, xtype,
 MPI_INFO_NULL, ierror)
 call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
 call MPI_FILE_CLOSE(fh, ierror)
endif
call MPI_BARRIER(MPI_COMM_WORLD, ierror)
if (myrank .eq. 1) then
 call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo', MPI_MODE_RDONLY,
 X.
 MPI_INFO_NULL, fh, ierror)
 call MPI_FILE_SET_VIEW(fh, zero, xtype, xtype, 'external32',&
 MPI_INFO_NULL, ierror)
 call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
 call MPI_FILE_CLOSE(fh, ierror)
endif
```

If processes 0 and 1 are on different machines, this code may not work as expected if the size is different on the two machines. (*End of advice to users.*)

#### 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 new semantics defined in TS 29113, most violations are resolved, and this is hinted at in an addendum to each item. The violations were originally adopted and have been retained because they are important for the usability of MPI. The rest of this section describes the potential problems in detail.

The following MPI features are inconsistent with Fortran 90 and Fortran 77.

#### **Unofficial Draft for Comment Only**

2

3

4

5 6

9 10

11

12

13

14

15

16

17

18

19 20

21 22

23

24

25

26

27

28

29

30 31 32

33

34 35

36 37

38

39

40

41

42

43

44

45

46

47

- 1. An MPI subroutine with a choice argument may be called with different argument types. When using the mpi_f08 module together with a compiler that supports Fortran 2008 + TS 29113, this problem is resolved.
- 2. An MPI subroutine with an assumed-size dummy argument may be passed an actual scalar argument. This is only solved for choice buffers through the use of DIMENSION(...).
- 3. Nonblocking and split-collective MPI routines assume that actual arguments are passed by address or descriptor and that arguments and the associated data are not copied on entrance to or exit from the subroutine. This problem is solved with the use of the ASYNCHRONOUS attribute.
- 4. An MPI implementation may read or modify user data (e.g., communication buffers used by nonblocking communications) concurrently with a user program that is executing outside of MPI calls. This problem is resolved by relying on the extended semantics of the ASYNCHRONOUS attribute as specified in TS 29113.
- 5. Several named "constants," such as MPI_BOTTOM, MPI_IN_PLACE,
   MPI_STATUS_IGNORE, MPI_STATUSES_IGNORE, MPI_ERRCODES_IGNORE,
   MPI_UNWEIGHTED, MPI_WEIGHTS_EMPTY, MPI_ARGV_NULL, and MPI_ARGVS_NULL
   are not ordinary Fortran constants and require a special implementation. See Section 2.5.4 for more information.
- 6. The memory allocation routine MPI_ALLOC_MEM cannot be used from Fortran 77/90/95 without a language extension (for example, Cray pointers) that allows the allocated memory to be associated with a Fortran variable. Therefore, address sized integers were used in MPI-2.0 – MPI-2.2. In Fortran 2003, TYPE(C_PTR) entities were added, which allow a standard-conforming implementation of the semantics of MPI_ALLOC_MEM. In MPI-3.0 and later, MPI_ALLOC_MEM has an additional, overloaded interface to support this language feature. The use of Cray pointers is deprecated. The mpi_f08 module only supports TYPE(C_PTR) pointers.
  - Additionally, MPI is inconsistent with Fortran 77 in a number of ways, as noted below.
  - MPI identifiers exceed 6 characters.
  - MPI identifiers may contain underscores after the first character.
  - MPI requires an include file, mpif.h. On systems that do not support include files, the implementation should specify the values of named constants.
  - Many routines in MPI have KIND-parameterized integers (e.g., MPI_ADDRESS_KIND and MPI_OFFSET_KIND) that hold address information. On systems that do not support Fortran 90-style parameterized types, INTEGER*8 or INTEGER should be used instead.
- ⁴⁴ MPI-1 contained several routines that take address-sized information as input or return
   ⁴⁵ 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

## Unofficial Draft for Comment Only

 $45 \\ 46$ 

KIND=MPI_ADDRESS_KIND. A number of MPI-2 functions also take INTEGER arguments of non-default KIND. See Section 2.6 and Section 5.1.1 for more information.

Sections 19.1.11 through 19.1.19 describe several problems in detail which concern the interaction of MPI and Fortran as well as their solutions. Some of these solutions require special capabilities from the compilers. Major requirements are summarized in Section 19.1.7.

#### 19.1.11 Problems Due to Strong Typing

All MPI functions with choice arguments associate actual arguments of different Fortran datatypes with the same dummy argument. This is not allowed by Fortran 77, and in Fortran 90, it is technically only allowed if the function is overloaded with a different function for each type (see also Section 19.1.6). In C, the use of void* formal arguments avoids these problems. Similar to C, with Fortran 2008 + TS 29113 (and later) together with the mpi_f08 module, the problem is avoided by declaring choice arguments with TYPE(*), DIMENSION(..), i.e., as assumed-type and assumed-rank dummy arguments.

Using INCLUDE 'mpif.h', the following code fragment is technically invalid and may generate a compile-time error.

```
integer i(5)
real x(5)
...
call mpi_send(x, 5, MPI_REAL, ...)
call mpi_send(i, 5, MPI_INTEGER, ...)
```

In practice, it is rare for compilers to do more than issue a warning. When using either the mpi_f08 or mpi module, the problem is usually resolved through the assumed-type and assumed-rank declarations of the dummy arguments, or with a compiler-dependent mechanism that overrides type checking for choice arguments.

It is also technically invalid in Fortran to pass a scalar actual argument to an array dummy argument that is not a choice buffer argument. Thus, when using the mpi_f08 or mpi module, the following code fragment usually generates an error since the dims and periods arguments to MPI_CART_CREATE are declared as assumed size arrays INTEGER :: DIMS(*) and LOGICAL :: PERIODS(*).

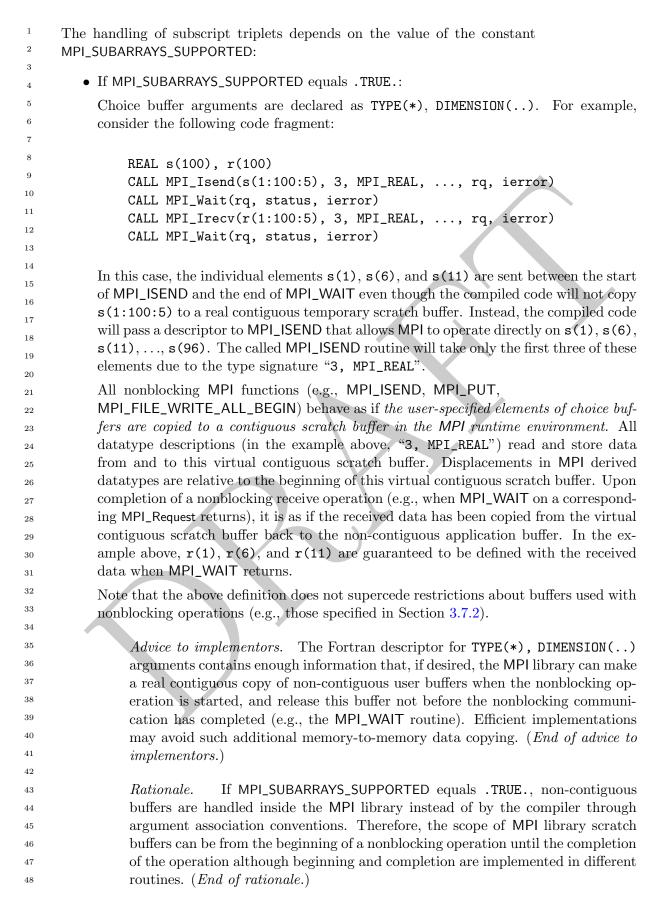
```
USE mpi_f08 ! or USE mpi
INTEGER size
CALL MPI_Cart_create(comm_old, 1, size, .TRUE., .TRUE., comm_cart, ierror)
```

Although this is a non-conforming MPI call, compiler warnings are not expected (but may occur) when using INCLUDE 'mpif.h' and this include file does not use Fortran explicit interfaces.

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, ...)



• If MPI_SUBARRAYS_SUPPORTED equals .FALSE.:

In this case, the use of Fortran arrays with subscript triplets as actual choice buffer arguments in any nonblocking MPI operation (which also includes persistent request, and split collectives) may cause undefined behavior. They may, however, be used in blocking MPI operations.

Implicit in MPI is the idea of a contiguous chunk of memory accessible through a linear address space. MPI copies data to and from this memory. An MPI program specifies the location of data by providing memory addresses and offsets. In the C language, sequence association rules plus pointers provide all the necessary low-level structure.

In Fortran, array data is not necessarily stored contiguously. For example, the array section A(1:N:2) involves only the elements of A with indices 1, 3, 5, .... The same is true for a pointer array whose target is such a section. Most compilers ensure that an array that is a dummy argument is held in contiguous memory if it is declared with an explicit shape (e.g., B(N)) or is of assumed size (e.g., B(*)). If necessary, they do this by making a copy of the array into contiguous memory.¹

Because MPI dummy buffer arguments are assumed-size arrays if MPI_SUBARRAYS_SUPPORTED equals .FALSE., this leads to a serious problem for a nonblocking call: the compiler copies the temporary array back on return but MPI continues to copy data to the memory that held it. For example, consider the following code fragment:

```
real a(100)
call MPI_IRECV(a(1:100:2), MPI_REAL, 50, ...)
```

Since the first dummy argument to MPI_IRECV is an assumed-size array (<type> buf(*)), the array section a(1:100:2) is copied to a temporary before being passed to MPI_IRECV, so that it is contiguous in memory. MPI_IRECV returns immediately, and data is copied from the temporary back into the array a. Sometime later, MPI may write to the address of the deallocated temporary. Copying is also a problem for MPI_ISEND since the temporary array may be deallocated before the data has all been sent from it.

Most Fortran 90 compilers do not make a copy if the actual argument is the whole of an explicit-shape or assumed-size array or is a "simply contiguous" section such as A(1:N) of such an array. ("Simply contiguous" is defined in the next paragraph.) Also, many compilers treat allocatable arrays the same as they treat explicit-shape arrays in this regard (though we know of one that does not). However, the same is not true for assumed-shape and pointer arrays; since they may be discontiguous, copying is often done. It is this copying that causes problems for MPI as described in the previous paragraph.

According to the Fortran 2008 Standard, Section 6.5.4, a "simply contiguous" array section is

name ( [:,]... [<subscript>]:[<subscript>] [,<subscript>]... )

 $\overline{7}$ 

¹Technically, the Fortran standard is worded to allow non-contiguous storage of any array data, unless the dummy argument has the CONTIGUOUS attribute.

1	That is, there are zero or more dimensions that are selected in full, then one dimension
2	selected without a stride, then zero or more dimensions that are selected with a simple
3	subscript. The compiler can detect from analyzing the source code that the array is
4	contiguous. Examples are
5	
6	A(1:N), A(:,N), A(:,1:N,1), A(1:6,N), A(:,:,1:N)
7	
8	Because of Fortran's column-major ordering, where the first index varies fastest, a
9	"simply contiguous" section of a contiguous array will also be contiguous.
10	The same problem can occur with a scalar argument. A compiler may make a copy of
11	scalar dummy arguments within a called procedure when passed as an actual argument
12	to a choice buffer routine. That this can cause a problem is illustrated by the example
13	
14	
15	real :: a
16	call user1(a,rq)
17 18	call MPI_WAIT(rq,status,ierr)
19	write (*,*) a
20	
20	subroutine user1(buf,request)
22	call MPI_IRECV(buf,,request,)
23	end
24	
25	If a is copied, MPI_IRECV will alter the copy when it completes the communication
26	and will not alter a itself.
27	Note that copying will almost certainly occur for an argument that is a non-trivial
28	expression (one with at least one operator or function call), a section that does not
29	select a contiguous part of its parent (e.g., $A(1:n:2)$ ), a pointer whose target is such
30	a section, or an assumed-shape array that is (directly or indirectly) associated with
31	such a section.
32	If a compiler option exists that inhibits copying of arguments, in either the calling or
33	called procedure, this must be employed.
34	If a compiler makes copies in the calling procedure of arguments that are explicit-
35	shape or assumed-size arrays, "simply contiguous" array sections of such arrays, or
36	scalars, and if no compiler option exists to inhibit such copying, then the compiler
37	cannot be used for applications that use MPI_GET_ADDRESS, or any nonblocking
$\frac{38}{39}$	MPI routine. If a compiler copies scalar arguments in the called procedure and there
40	is no compiler option to inhibit this, then this compiler cannot be used for applications
41	that use memory references across subroutine calls as in the example above.
42	
43	19.1.13 Problems Due to Data Copying and Sequence Association with Vector Subscripts
44	Fortran arrays with <b>vector</b> subscripts describe subarrays containing a possibly irregular
45	set of elements
46	
47	REAL a(100)
48	CALL MPI_Send(A((/7,9,23,81,82/)), 5, MPI_REAL,)

CHAPTER 19. LANGUAGE BINDINGS

Fortran arrays with a vector subscript must not be used as actual choice buffer arguments in any nonblocking or split collective MPI operations. They may, however, be used in blocking MPI operations.

#### 19.1.14 Special Constants

MPI requires a number of special "constants" that cannot be implemented as normal Fortran constants, e.g., MPI_BOTTOM. The complete list can be found in Section 2.5.4. In C, these are implemented as constant pointers, usually as NULL and are used where the function prototype calls for a pointer to a variable, not the variable itself.

In Fortran, using special values for the constants (e.g., by defining them through **parameter** statements) is not possible because an implementation cannot distinguish these values from valid data. Typically these constants are implemented as predefined static variables (e.g., a variable in an MPI-declared COMMON block), relying on the fact that the target compiler passes data by address. Inside the subroutine, the address of the actual choice buffer argument can be compared with the address of such a predefined static variable.

These special constants also cause an exception with the usage of Fortran INTENT: with USE mpi_f08, the attributes INTENT(IN), INTENT(OUT), and INTENT(INOUT) are used in the Fortran interface. In most cases, INTENT(IN) is used if the C interface uses call-by-value. For all buffer arguments and for dummy arguments that may be modified and allow one of these special constants as input, an INTENT is not specified.

## 19.1.15 Fortran Derived Types

MPI supports passing Fortran entities of BIND(C) and SEQUENCE derived types to choice dummy arguments, provided no type component has the ALLOCATABLE or POINTER attribute.

The following code fragment shows some possible ways to send scalars or arrays of interoperable derived type in Fortran. The example assumes that all data is passed by address.

```
type, BIND(C) :: mytype
 integer :: i
 real :: x
 double precision :: d
 logical :: 1
end type mytype
type(mytype) :: foo, fooarr(5)
integer :: blocklen(4), type(4)
integer(KIND=MPI_ADDRESS_KIND) :: disp(4), base, lb, extent
call MPI_GET_ADDRESS(foo%i, disp(1), ierr)
call MPI_GET_ADDRESS(foo%x, disp(2), ierr)
call MPI_GET_ADDRESS(foo%d, disp(3), ierr)
call MPI_GET_ADDRESS(foo%1, disp(4), ierr)
base = disp(1)
disp(1) = disp(1) - base
disp(2) = disp(2) - base
```

```
1
 disp(3) = disp(3) - base
\mathbf{2}
 disp(4) = disp(4) - base
3
4
 blocklen(1) = 1
5
 blocklen(2) = 1
6
 blocklen(3) = 1
\overline{7}
 blocklen(4) = 1
8
9
 type(1) = MPI_INTEGER
10
 type(2) = MPI_REAL
11
 type(3) = MPI_DOUBLE_PRECISION
12
 type(4) = MPI_LOGICAL
13
14
 call MPI_TYPE_CREATE_STRUCT(4, blocklen, disp, type, newtype, ierr)
15
 call MPI_TYPE_COMMIT(newtype, ierr)
16
17
 call MPI_SEND(foo%i, 1, newtype, dest, tag, comm, ierr)
18
 ! or
19
 call MPI_SEND(foo, 1, newtype, dest, tag, comm, ierr)
20
 ! expects that base == address(foo%i) == address(foo)
21
22
 call MPI_GET_ADDRESS(fooarr(1), disp(1), ierr)
23
 call MPI_GET_ADDRESS(fooarr(2), disp(2), ierr)
^{24}
 extent = disp(2) - disp(1)
25
 1b = 0
26
 call MPI_TYPE_CREATE_RESIZED(newtype, lb, extent, newarrtype, ierr)
27
 call MPI_TYPE_COMMIT(newarrtype, ierr)
28
29
 call MPI_SEND(fooarr, 5, newarrtype, dest, tag, comm, ierr)
30
 Using the derived type variable foo instead of its first basic type element foo%i may
^{31}
```

³¹Using the derived type variable foo instead of its first basic type element foo%1 may ³² be impossible if the MPI library implements choice buffer arguments through overloading ³³ instead of using TYPE(*), DIMENSION(..), or through a non-standardized extension such ³⁴ as !\$PRAGMA IGNORE_TKR; see Section 19.1.6.

To use a derived type in an array requires a correct extent of the datatype handle 35 to take care of the alignment rules applied by the compiler. These alignment rules may 36 imply that there are gaps between the components of a derived type, and also between the 37 subsuguent elements of an array of a derived type. The extent of an interoperable derived 38 type (i.e., defined with BIND(C)) and a SEQUENCE derived type with the same content may 39 be different because C and Fortran may apply different alignment rules. As recommended 40 in the advice to users in Section 5.1.6, one should add an additional fifth structure element 41 with one numerical storage unit at the end of this structure to force in most cases that 42the array of structures is contiguous. Even with such an additional element, one should 43 keep this resizing due to the special alignment rules that can be used by the compiler for 44 structures, as also mentioned in this advice. 45

Using the extended semantics defined in TS 29113, it is also possible to use entities or derived types without either the BIND(C) or the SEQUENCE attribute as choice buffer arguments; some additional constraints must be observed, e.g., no ALLOCATABLE or POINTER

type components may exist. In this case, the **base** address in the example must be changed to become the address of **foo** instead of **foo%i**, because the Fortran compiler may rearrange type components or add padding. Sending the structure **foo** should then also be performed by providing it (and not **foo%i**) as actual argument for MPI_Send.

## 19.1.16 Optimization Problems, an Overview

MPI provides operations that may be hidden from the user code and run concurrently with it, accessing the same memory as user code. Examples include the data transfer for an MPI_IRECV. The optimizer of a compiler will assume that it can recognize periods when a copy of a variable can be kept in a register without reloading from or storing to memory. When the user code is working with a register copy of some variable while the hidden operation reads or writes the memory copy, problems occur. These problems are independent of the Fortran support method; i.e., they occur with the mpi_f08 module, the mpi module, and the mpif.h include file.

This section shows four problematic usage areas (the abbreviations in parentheses are used in the table below):

- Use of nonblocking routines or persistent requests (Nonbl.).
- Use of one-sided routines (1-sided).
- Use of MPI parallel file I/O split collective operations (Split).
- Use of MPI_BOTTOM together with absolute displacements in MPI datatypes, or relative displacements between two variables in such datatypes (*Bottom*).

The following compiler optimization strategies (valid for serial code) may cause problems in MPI applications:

- Code movement and register optimization problems; see Section 19.1.17.
- Temporary data movement and temporary memory modifications; see Section 19.1.18.
- Permanent data movement (e.g., through garbage collection); see Section 19.1.19.

Table 19.2 shows the only usage areas where these optimization problems may occur.

Optimization	may cause a problem in			
	following usage areas			
	Nonbl.	1-sided	Split	Bottom
Code movement	yes	yes	no	yes
and register optimization				
Temporary data movement	yes	yes	yes	no
Permanent data movement	yes	yes	yes	yes

# Table 19.2: Occurrence of Fortran optimization problems in several usage areas

The solutions in the following sections are based on compromises:

## Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

5 6

7

8

9

10

11

12

13

14

15

16

17 18

19

20 21

22 23

24

25 26

27 28

29 30

 31 

32 33

 $45 \\ 46$ 

1 • to minimize the burden for the application programmer, e.g., as shown in Sections  $\mathbf{2}$ "Solutions" through "The (Poorly Performing) Fortran VOLATILE Attribute" on 3 pages 819-824, 4 • to minimize the drawbacks on compiler based optimization, and 56 • to minimize the requirements defined in Section 19.1.7. 7 8 19.1.17 Problems with Code Movement and Register Optimization 9 10 Nonblocking Operations 11 If a variable is local to a Fortran subroutine (i.e., not in a module or a COMMON block), the 12compiler will assume that it cannot be modified by a called subroutine unless it is an actual 13 argument of the call. In the most common linkage convention, the subroutine is expected 14 to save and restore certain registers. Thus, the optimizer will assume that a register which 15held a valid copy of such a variable before the call will still hold a valid copy on return. 1617**Example 19.1** Fortran 90 register optimization—extreme. 18 19Source compiled as or compiled as 20REAL :: buf, b1 21REAL :: buf, b1 REAL :: buf, b1 call MPI_IRECV(buf,..req) call MPI_IRECV(buf,..req) call MPI_IRECV(buf,..req) 22 register = buf b1 = buf23call MPI_WAIT(req,..) call MPI_WAIT(req,..) call MPI_WAIT(req,..)  24 b1 = bufb1 = register 2526Example 19.1 shows extreme, but allowed, possibilities. MPI_WAIT on a concurrent 27thread modifies buf between the invocation of MPI_IRECV and the completion of MPI_WAIT. 28But the compiler cannot see any possibility that buf can be changed after MPI_IRECV has 29returned, and may schedule the load of **buf** earlier than typed in the source. The compiler 30 has no reason to avoid using a register to hold **buf** across the call to MPI_WAIT. It also may  31 reorder the instructions as illustrated in the rightmost column. 32 33 34**Example 19.2** Similar example with MPI_ISEND 35 Source compiled as with a possible MPI-internal 36 execution sequence 37 38REAL :: buf, copy REAL :: buf, copy REAL :: buf, copy 39 buf = val buf = val buf = val call MPI_ISEND(buf,..req) call MPI_ISEND(buf,..req) addr = &buf 40copy = buf copy = bufcopy= buf 41 buf = val_overwrite buf = val_overwrite 42call MPI_WAIT(req,..) call MPI_WAIT(req,..) call send(*addr) ! within 43 ! MPI_WAIT 44buf = val_overwrite 454647

⁴⁷ Due to valid compiler code movement optimizations in Example 19.2, the content of ⁴⁸ 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 which uses one variable as an anchor and brings along others by using MPI_GET_ADDRESS to determine their offsets from the anchor is another. The anchor variable would be the only one referenced in the call. Also attention must be paid if MPI operations are used that run in parallel with the user's application.

Example 19.3 shows what Fortran compilers are allowed to do.

In Example 19.3, the compiler does not invalidate the register because it cannot see that MPI_RECV changes the value of buf. The access to buf is hidden by the use of MPI_GET_ADDRESS and MPI_BOTTOM.

In Example 19.4, several successive assignments to the same variable buf can be combined in a way such that only the last assignment is executed. "Successive" means that no interfering load access to this variable occurs between the assignments. The compiler cannot detect that the call to MPI_SEND statement is interfering because the load access to buf is hidden by the usage of MPI_BOTTOM.

#### Solutions

The following sections show in detail how the problems with code movement and register optimization can be portably solved. Application writers can partially or fully avoid these compiler optimization problems by using one or more of the special Fortran declarations

#### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

```
1
 Example 19.3 Fortran 90 register optimization.
\mathbf{2}
3
 This source ...
 can be compiled as:
4
 call MPI_GET_ADDRESS(buf, bufaddr,
 call MPI_GET_ADDRESS(buf,...)
5
 ierror)
6
 call MPI_TYPE_CREATE_STRUCT(1,1,
 call MPI_TYPE_CREATE_STRUCT(...)
7
 bufaddr,
8
 MPI_REAL, type, ierror)
9
 call MPI_TYPE_COMMIT(type,ierror)
 call MPI_TYPE_COMMIT(...)
10
 register = buf
 val_old = buf
11
 val_old = register
12
 call MPI_RECV(MPI_BOTTOM,1,type,...)
 call MPI_RECV(MPI_BOTTOM,...)
13
 val_new = buf
 val_new = register
14
15
16
 Example 19.4 Similar example with MPI_SEND
17
18
 can be compiled as:
 This source ...
19
 ! buf contains val_old
 ! buf contains val_old
20
 buf = val_new
21
 call MPI_SEND(...)
 call MPI_SEND(MPI_BOTTOM,1,type,...)
22
 ! with buf as a displacement in type
 ! i.e. val_old is sent
23
24
 !
 buf=val_new is moved to here
25
26
 ! and detected as dead code
 and therefore removed
27
28
 buf = val_overwrite
 buf = val_overwrite
29
30
^{31}
 with the send and receive buffers used in nonblocking operations, or in operations in which
32
 MPI_BOTTOM is used, or if datatype handles that combine several variables are used:
33
34
 • Use of the Fortran ASYNCHRONOUS attribute.
35
36
 • Use of the helper routine MPI_F_SYNC_REG, or an equivalent user-written dummy
37
 routine.
38
 • Declare the buffer as a Fortran module variable or within a Fortran common block.
39
40
 • Use of the Fortran VOLATILE attribute.
41
42
 Each of these methods solves the problems of code movement and register optimization,
43
 but may incur various degrees of performance impact, and may not be usable in every
44
 application context. These methods may not be guaranteed by the Fortran standard, but
45
 they must be guaranteed by a MPI-3.0 (and later) compliant MPI library and associated
46
 compiler suite according to the requirements listed in Section 19.1.7. The performance
47
 impact of using MPI_F_SYNC_REG is expected to be low, that of using module variables
48
 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 827, 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 ⁴⁶ between the MPI_IXXX routines and MPI_Waitall. Case (a) works fine because the read ⁴⁷ accesses to b occur after the communication has completed. ⁴⁸

## Unofficial Draft for Comment Only

1

2

3

4 5

6

 $\overline{7}$ 

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

 24 

25 26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

In Case (b), the read accesses to b(1:100) in the loop i=2,99 are read accesses to a pending communication affector while input communication (i.e., the two MPI_Irecv calls) is pending. This is a contradiction to the rule that *for input communication, a pending communication affector shall not be referenced*. The problem can be solved by using separate variables for the halos and the inner array, or by splitting a common array into disjoint subarrays which are passed through different dummy arguments into a subroutine, as shown in Example 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.

¹⁵ 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	Example 19.2
can be solved with	can be solved with
<pre>call MPI_IRECV(buf,req)</pre>	buf = val
	<pre>call MPI_ISEND(buf,req)</pre>
	copy = buf
call MPI_WAIT(req,)	call MPI_WAIT(req,)
call MPI_F_SYNC_REG(buf)	call MPI_F_SYNC_REG(buf)
b1 = buf	<pre>buf = val_overwrite</pre>

The call to MPI_F_SYNC_REG(buf) prevents moving the last line before the MPI_WAIT call. Further calls to MPI_F_SYNC_REG(buf) are not needed because it is still correct if the additional read access copy=buf is moved below MPI_WAIT and before buf=val_overwrite.

• The problems illustrated by the Examples 19.3 and 19.4 can be solved with two additional MPI_F_SYNC_REG(buf) statements; one directly before MPI_RECV/MPI_SEND, and one directly after this communication operation.

Example 19.3 Example 19.4 42can be solved with can be solved with 43 call MPI_F_SYNC_REG(buf) call MPI_F_SYNC_REG(buf) 44 call MPI_RECV(MPI_BOTTOM,...) call MPI_SEND(MPI_BOTTOM,...) 45call MPI_F_SYNC_REG(buf) call MPI_F_SYNC_REG(buf) 46 47

9

10

11

12

13

14

21

22

34

35

36

37 38

39 40

41

The first call to MPI_F_SYNC_REG(buf) is needed to finish all load and store references to buf prior to MPI_RECV/MPI_SEND; the second call is needed to assure that any subsequent access to buf is not moved before MPI_RECV/MPI_SEND.

• In the example in Section 12.7.4, two asynchronous accesses must be protected: in Process 1, the access to bbbb must be protected similar to Example 19.1, i.e., a call to MPI_F_SYNC_REG(bbbb) is needed after the second MPI_WIN_FENCE to guarantee that further accesses to bbbb are not moved ahead of the call to MPI_WIN_FENCE. In Process 2, both calls to MPI_WIN_FENCE together act as a communication call with MPI_BOTTOM as the buffer. That is, before the first fence and after the second fence, a call to MPI_F_SYNC_REG(buff) is needed to guarantee that accesses to buff are not moved after or ahead of the calls to MPI_WIN_FENCE. Using MPI_GET instead of MPI_PUT, the same calls to MPI_F_SYNC_REG are necessary.

Source of Process 1	Source of Process 2
bbbb = 777	buff = 999
	call MPI_F_SYNC_REG(buff)
call MPI_WIN_FENCE	call MPI_WIN_FENCE
call MPI_PUT(bbbb	
into buff of process 2)	
call MPI_WIN_FENCE	call MPI_WIN_FENCE
call MPI_F_SYNC_REG(bbbb)	call MPI_F_SYNC_REG(buff)
	ccc = buff
	call MPI_F_SYNC_REG(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 ⁴⁶ such usage in existing application programs although new applications should prefer ⁴⁷ MPI_F_SYNC_REG or one of the other possibilities. In an existing application, calls to ⁴⁸

## Unofficial Draft for Comment Only

¹ such a user-written routine should be substituted by a call to MPI_F_SYNC_REG because
 ² the user-written routine may not be implemented in accordance with the rules specified in

- Section 19.1.7.
- 3 4 5

6

13

14

15

16 17

18

## 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.
- 19 20 21

## 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.

- 29
- 30 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.*)

39 40

37

38

⁴¹ 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 827. Example 19.6 also shows a possibility that could be problematic.

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 subsection *The Fortran ASYNCHRONOUS attribute* on page 821 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 ⁴⁶ restriction, a new or modified asynchronous feature in the Fortran language would ⁴⁷ be necessary: an asynchronous attribute that can be used on parts of an array and ⁴⁸

#### **Unofficial Draft for Comment Only**

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6

 $\overline{7}$ 

8 9

10

11

12

13

14 15

16

17 18

19 20

21

22

23

 24 

25 26

27

28 29

30 31

32

33

34

35

36

37

38

39

40

41

42

43

44

together with asynchronous operations outside the scope of Fortran. If such a feature becomes available in a future edition of the Fortran standard, then this restriction also may be weakened in a later version of the MPI standard. (End of rationale.)

In Example 19.9 (which is a solution for the problem shown in Example 19.5 and in Example 19.10 (which is a solution for the problem shown in Example 19.8), the ar-6 ray is split into inner and halo part and both disjoint parts are passed to a subroutine 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 10 area are strided arrays. Those can be used in nonblocking communication only with a TS 11 29113 based MPI library. 12

13 14

15

16

17

18

19 20

21

22

23

 24 25

26

27

28

1

2

3

4

5

7

8

9

## 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 MPL library together with the compiler used; see Section 19.1.7.

2930  31 

#### Comparison with C 19.1.20

32 In C, subroutines which modify variables that are not in the argument list will not cause 33 register optimization problems. This is because taking pointers to storage objects by using 34the & operator and later referencing the objects by indirection on the pointer is an integral 35 part of the language. A C compiler understands the implications, so that the problem should 36 not occur, in general. However, some compilers do offer optional aggressive optimization 37 levels which may not be safe. Problems due to temporary memory modifications can also 38 occur in C. As above, the best advice is to avoid the problem: use different variables for 39 buffers in nonblocking MPI operations and computation that is executed while a nonblocking 40operation is pending.

- 41
- 42
- 43
- 4445
- 46
- 47
- 48

Example 19.5 Protecting nonblocking communication with the ASYNCHRONOUS attribute.

```
4
USE mpi_f08
 5
REAL, ASYNCHRONOUS :: b(0:101) ! elements 0 and 101 are halo cells
 6
 ! elements 1 and 100 are newly computed
REAL :: bnew(0:101)
TYPE(MPI_Request) :: req(4)
INTEGER :: left, right, i
 9
CALL MPI_Cart_shift(...,left,right,...)
 10
CALL MPI_Irecv(b(0), ..., left, ..., req(1), ...)
 11
CALL MPI_Irecv(b(101), ..., right, ..., req(2), ...)
 12
CALL MPI_Isend(b(1), ..., left, ..., req(3), ...)
 13
CALL MPI_Isend(b(100), ..., right, ..., req(4), ...)
 14
 15
#ifdef WITHOUT_OVERLAPPING_COMMUNICATION_AND_COMPUTATION
 16
! Case (a)
 17
 CALL MPI_Waitall(4, req, ...)
 18
 DO i=1,100 ! compute all new local data
 19
 bnew(i) = function(b(i-1), b(i), b(i+1))
 20
 END DO
 21
#endif
 22
 23
#ifdef WITH_OVERLAPPING_COMMUNICATION_AND_COMPUTATION
 ^{24}
! Case (b)
 25
 DO i=2,99 ! compute only elements for which halo data is not needed
 26
 bnew(i) = function(b(i-1), b(i), b(i+1))
 27
 END DO
 28
 CALL MPI_Waital1(4, req, ...)
 29
 i=1 ! compute leftmost element
 30
 bnew(i) = function(b(i-1), b(i), b(i+1))
 31
 i=100 ! compute rightmost element
 32
 bnew(i) = function(b(i-1), b(i), b(i+1))
 33
#endif
 34
 35
 36
Example 19.6 Overlapping Communication and Computation.
 37
 38
USE mpi_f08
 39
REAL :: buf(100,100)
 40
CALL MPI_Irecv(buf(1,1:100),..., req,...)
 41
DO j=1,100
 42
 DO i=2,100
 43
 buf(i,j)=...
 44
 END DO
 45
END DO
 46
CALL MPI_Wait(req,...)
 47
 48
```

```
1
 Example 19.7 The compiler may substitute the nested loops through loop fusion.
\mathbf{2}
3
 REAL :: buf(100,100), buf_1dim(10000)
4
 EQUIVALENCE (buf(1,1), buf_1dim(1))
\mathbf{5}
 CALL MPI_Irecv(buf(1,1:100),..., req,...)
6
 tmp(1:100) = buf(1,1:100)
\overline{7}
 DO j=1,10000
8
 buf_1dim(h)=...
9
 END DO
10
 buf(1,1:100) = tmp(1:100)
11
 CALL MPI_Wait(req,...)
12
13
14
15
16
17
18
19
20
21
22
23
^{24}
25
26
27
 Example 19.8 Another optimization is based on the usage of a separate memory storage
28
 area, e.g., in a GPU.
29
30
 REAL :: buf(100,100), local_buf(100,100)
31
 CALL MPI_Irecv(buf(1,1:100),..., req,...)
32
 local_buf = buf
33
 DO j=1,100
34
 DO i=2,100
35
 local_buf(i,j)=...
36
 END DO
37
 END DO
38
 buf = local_buf ! may overwrite asynchronously received
39
 data in buf(1,1:100)
40
 CALL MPI_Wait(req,...)
41
42
43
44
45
46
47
48
```

**Example 19.9** Using separated variables for overlapping communication and computation to allow the protection of nonblocking communication with the ASYNCHRONOUS attribute.

```
USE mpi_f08
REAL :: b(0:101)
 ! elements 0 and 101 are halo cells
REAL :: bnew(0:101) ! elements 1 and 100 are newly computed
INTEGER :: i
CALL separated_sections(b(0), b(1:100), b(101), bnew(0:101))
i=1 ! compute leftmost element
 bnew(i) = function(b(i-1), b(i), b(i+1))
i=100 ! compute rightmost element
 bnew(i) = function(b(i-1), b(i), b(i+1))
END
SUBROUTINE separated_sections(b_lefthalo, b_inner, b_righthalo, bnew)
USE mpi_f08
REAL, ASYNCHRONOUS :: b_lefthalo(0:0), b_inner(1:100), b_righthalo(101:101)
REAL :: bnew(0:101) ! elements 1 and 100 are newly computed
TYPE(MPI_Request) :: req(4)
 20
INTEGER :: left, right, i
 21
CALL MPI_Cart_shift(...,left, right,...)
 22
CALL MPI_Irecv(b_lefthalo (0), ..., left, ..., req(1), ...)
 23
CALL MPI_Irecv(b_righthalo(101), ..., right, ..., req(2), ...)
! b_lefthalo and b_righthalo is written asynchronously.
! There is no other concurrent access to b_lefthalo and b_righthalo.
CALL MPI_Isend(b_inner(1),
 ..., left, ..., req(3), ...)
 27
CALL MPI_Isend(b_inner(100),
 ..., right, ..., req(4), ...)
 28
 29
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
 34
CALL MPI_Waitall(4, req,...)
 35
END SUBROUTINE
 37
```

1

2

4

5

6

7

9

10

11

12

13

1415

16

17

18

19

 24 

25

26

30

31

32

33

36

```
1
 Example 19.10 Protecting GPU optimizations with the ASYNCHRONOUS attribute.
\mathbf{2}
3
 USE mpi_f08
4
 REAL :: buf(100,100)
\mathbf{5}
 CALL separated_sections(buf(1:1,1:100), buf(2:100,1:100))
6
 END
\overline{7}
8
 SUBROUTINE separated_sections(buf_halo, buf_inner)
9
 REAL, ASYNCHRONOUS :: buf_halo(1:1,1:100)
10
 REAL :: buf_inner(2:100,1:100)
11
 REAL :: local_buf(2:100,100)
12
13
 CALL MPI_Irecv(buf_halo(1,1:100),..., req,...)
14
 local_buf = buf_inner
15
 DO j=1,100
16
 DO i=2,100
17
 local_buf(i,j)=...
18
 END DO
19
 END DO
20
 buf_inner = local_buf ! buf_halo is not touched!!!
21
^{22}
 CALL MPI_Wait(req,...)
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
```

19.2	Support for Large Count and Large Byte Displacement in MPI Lan-	1
	guage Bindings	2
	BadBo BulanBo	3
The fo	blowing types, which were used prior to MPI-4.0, have been deemed too small to hold	4
values	that applications wish to use:	5
• '	The C int type and the Fortran INTEGER type were used for <i>count</i> parameters.	6 7
• '	The C int type and the Fortran INTECEP type were used for some parameters that	8
	The C int type and the Fortran INTEGER type were used for some parameters that represent <i>byte displacement</i> in memory.	9 10
• ′	The C MPI_Aint type and the Fortran INTEGER(KIND=MPI_ADDRESS_KIND) type were	10
	used for some parameters that represent byte displacement in files (e.g., in constructors	12
(	of MPI datatypes that can be used with files).	13
In ord	ler to avoid breaking backwards compatibility, this version of MPI supports larger	14
	via separate additional MPI procedures in C (suffixed with "_c") and via interface	15
	orphism in Fortran when using USE mpi_f08. No polymorphic support for larger	16
	is provided in Fortran when using mpif.h and use mpi.	17
F	or the large count versions of three datatype constructors,	18
MPI_	TYPE_CREATE_HINDEXED, MPI_TYPE_CREATE_HINDEXED_BLOCK, and	19
MPI_	TYPE_CREATE_STRUCT, absolute addresses shall not be used to specify byte dis-	20
-	nents since the parameter is of type MPI_COUNT instead of type MPI_AINT (see Sec-	21
tion $2$		22
	addition, the functions MPI_TYPE_GET_ENVELOPE and	23
	GET_CONTENTS also support large count types via additional parameters in	24
-	te additional MPI procedures in C (suffixed with "_c") and interface polymorphism	25
	tran when using USE mpi_f08 (see Section 5.1.13).	26
	urther, the callbacks MPI_USER_FUNCTION and	27
	DATAREP_CONVERSION_FUNCTION also support large count types via separate ad-	28
	al callback prototypes in C (suffixed with "_c") and multiple abstract interfaces in	29 30
	n when using USE mpi_f08 (see Sections 6.9.5 and 14.5.3, respectively).	31
	C bindings, for each MPI procedure that had at least one <i>count</i> or <i>byte displacement</i>	32
	the that used the int and/or MPI_Aint types prior to MPI-4.0, an additional MPI	33
	lure is provided, with the same name but suffixed by "_c". The MPI procedure	34
	ut the "_c" token has the same name and parameter types as versions prior to MPI- 'he "_c" suffixed MPI procedure has MPI_Count for all <i>count</i> parameters, MPI_Aint for	35
	eters that represent byte displacement in memory, MPI_Offset for parameters that	36
-	ent byte displacement in files, and MPI_Count for parameters that may represent byte	37
-	<i>cement</i> in both memory and files.	38
	a Fortran, when using USE mpi_f08, for each MPI procedure that had at least one	39
	or byte displacement parameter that used the INTEGER or	40
	ER(KIND=MPI_ADDRESS_KIND) types prior to MPI-4.0, a polymorphic interface con-	41
	g two specific procedures is provided. One of the specific procedures has the same	42

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 44specific procedure has the same name followed by "_c", and then suffixed by the token 4546specified in Table 19.1 for USE mpi_f08. It also has INTEGER(KIND=MPI_COUNT_KIND) 47for all count parameters, INTEGER(KIND=MPI_ADDRESS_KIND) for parameters that repre-48sent byte displacement in memory, INTEGER(KIND=MPI_OFFSET_KIND) for parameters that

43

¹ represent *byte displacement* in files, and INTEGER(KIND=MPI_COUNT_KIND) for parame-² ters that may represent *byte displacement* in both memory and files (for more details ³ on specific Fortran procedure names and related calling conventions, refer to Table 19.1 ⁴ in Section 19.1.5). There is one exception: if the type signatures of the two specific ⁵ procedures are identical (e.g., if INTEGER(KIND=MPI_COUNT_KIND) is the same type as ⁶ INTEGER(KIND=MPI_ADDRESS_KIND)), then the implementation shall not provide the "_c" ⁷ specific procedure.

⁸ It is erroneous to directly invoke the "_c" specific procedures in the Fortran mpi_f08 ⁹ module with the exception of the following procedures: MPI_Op_create_c and ¹⁰ MPI_Register_datarep_c.

¹¹ In older Fortran bindings (mpif.h and use mpi), no new interfaces and no new specific ¹² procedures for larger types are provided beyond what existed in MPI-4.0; all MPI procedures ¹³ have the same types as in the versions prior to MPI-4.0.

14 15

16

## 19.3 Language Interoperability

# ¹⁷₁₈ 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.

28

35

38

39

40

There are several issues that need to be addressed in order to achieve interoperability.

²⁹₃₀ **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.

⁴¹₄₂ 19.3.2 Assumptions

⁴³ We assume that conventions exist for programs written in one language to call routines ⁴⁴ written in another language. These conventions specify how to link routines in different ⁴⁵ languages into one program, how to call functions in a different language, how to pass ⁴⁶ arguments between languages, and the correspondence between basic datatypes in different ⁴⁷ languages. In general, these conventions will be implementation dependent. Furthermore, ⁴⁸ not every basic datatype may have a matching type in other languages. For example,

C character strings may not be compatible with Fortran CHARACTER variables. However, we assume that a Fortran INTEGER, as well as a (sequence associated) Fortran array of INTEGERs, can be passed to a C program. We also assume that Fortran and C have address-sized integers. This does not mean that the default-size integers are the same size as default-sized pointers, but only that there is some way to hold (and pass) a C address in a Fortran integer. It is also assumed that INTEGER(KIND=MPI_OFFSET_KIND) can be passed from Fortran to C as MPI_Offset.

#### 19.3.3 Initialization

A call to MPI_INIT or MPI_INIT_THREAD, from any language, initializes MPI for execution in all languages.

Advice to users. Certain implementations use the (inout) argc, argv arguments of the C version of MPI_INIT in order to propagate values for argc and argv to all executing processes. Use of the Fortran version of MPI_INIT to initialize MPI may result in a loss of this ability. (*End of advice to users.*)

The function MPI_INITIALIZED returns the same answer in all languages. The function MPI_FINALIZE finalizes the MPI environments for all languages. The function MPI_FINALIZED returns the same answer in all languages.

The function MPI_ABORT kills processes, irrespective of the language used by the caller or by the processes killed.

The MPI environment is initialized in the same manner for all languages by MPI_INIT. E.g., MPI_COMM_WORLD carries the same information regardless of language: same processes, same environmental attributes, same error handlers.

Information can be added to info objects in one language and retrieved in another.

Advice to users. The use of several languages in one MPI program may require the use of special options at compile and/or link time. (*End of advice to users.*)

Advice to implementors. Implementations may selectively link language specific MPI libraries only to codes that need them, so as not to increase the size of binaries for codes that use only one language. The MPI initialization code need perform initialization for a language only if that language library is loaded. (*End of advice to implementors.*)

#### 19.3.4 Transfer of Handles

Handles are passed between Fortran and C by using an explicit C wrapper to convert Fortran handles to C handles. There is no direct access to C handles in Fortran.

The type definition MPI_Fint is provided in C for an integer of the size that matches a 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.

 $\mathbf{2}$ 

 24 

1	C binding
2	MPI_Comm_MPI_Comm_f2c(MPI_Fint comm)
3 4 5 6 7 8	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)
9 10 11 12 13	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)
14 15	MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)
16	MPI_Group MPI_Group_f2c(MPI_Fint group)
17 18	MPI_Fint MPI_Group_c2f(MPI_Group group)
19	MPI_Request MPI_Request_f2c(MPI_Fint request)
20 21	MPI_Fint MPI_Request_c2f(MPI_Request request)
22 23	MPI_File MPI_File_f2c(MPI_Fint file)
24	MPI_Fint MPI_File_c2f(MPI_File file)
25 26	MPI_Win MPI_Win_f2c(MPI_Fint win)
27	MPI_Fint MPI_Win_c2f(MPI_Win win)
28 29	MPI_Op MPI_Op_f2c(MPI_Fint op)
30	MPI_Fint MPI_Op_c2f(MPI_Op op)
31 32	MPI_Info MPI_Info_f2c(MPI_Fint info)
33	MPI_Fint MPI_Info_c2f(MPI_Info info)
34 35	MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)
36	MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)
37 38	MPI_Message MPI_Message_f2c(MPI_Fint message)
39	MPI_Fint MPI_Message_c2f(MPI_Message message)
40 41	MPI_Session MPI_Session_f2c(MPI_Fint session)
42 43	MPI_Fint MPI_Session_c2f(MPI_Session session)
43	
45	
46 47	
48	

**Example 19.11** The example below illustrates how the Fortran MPI function MPI_TYPE_COMMIT can be implemented by wrapping the C MPI function MPI_Type_commit with a C wrapper to do handle conversions. In this example a Fortran-C interface is assumed where a Fortran function is all upper case when referred to from C and arguments are passed by addresses.

```
! FORTRAN PROCEDURE
SUBROUTINE MPI_TYPE_COMMIT(DATATYPE, IERR)
INTEGER :: DATATYPE, IERR
CALL MPI_X_TYPE_COMMIT(DATATYPE, IERR)
RETURN
END
/* C wrapper */
void MPI_X_TYPE_COMMIT(MPI_Fint *f_handle, MPI_Fint *ierr)
{
 MPI_Datatype datatype;
 datatype = MPI_Type_f2c(*f_handle);
 *ierr = (MPI_Fint)MPI_Type_commit(&datatype);
 *f_handle = MPI_Type_c2f(datatype);
 return;
}
```

The same approach can be used for all other MPI functions. The call to MPI_XXX_f2c (resp. MPI_XXX_c2f) can be omitted when the handle is an OUT (resp. IN) argument, rather than INOUT.

*Rationale.* The design here provides a convenient solution for the prevalent case, where a C wrapper is used to allow Fortran code to call a C library, or C code to call a Fortran library. The use of C wrappers is much more likely than the use of Fortran wrappers, because it is much more likely that a variable of type INTEGER can be passed to C, than a C handle can be passed to Fortran.

Returning the converted value as a function value rather than through the argument list allows the generation of efficient inlined code when these functions are simple (e.g., the identity). The conversion function in the wrapper does not catch an invalid handle argument. Instead, an invalid handle is passed below to the library function, which, presumably, checks its input arguments. (*End of rationale.*)

## 19.3.5 Status

The following two procedures are provided in C to convert from a Fortran (with the mpi module or mpif.h) status (which is an array of integers) to a C status (which is a structure), and vice versa. The conversion occurs on all the information in status, including that which is hidden. That is, no status information is lost in the conversion.

int MPI_Status_f2c(const MPI_Fint *f_status, MPI_Status *c_status)

 $\mathbf{2}$ 

3

4

5 6

7

9

10

11

12 13

14 15

16

17

18 19

20

21

22

23

24 25

26

27

28 29

30

31

32

33

34

35

36

37

38

39 40 41

42

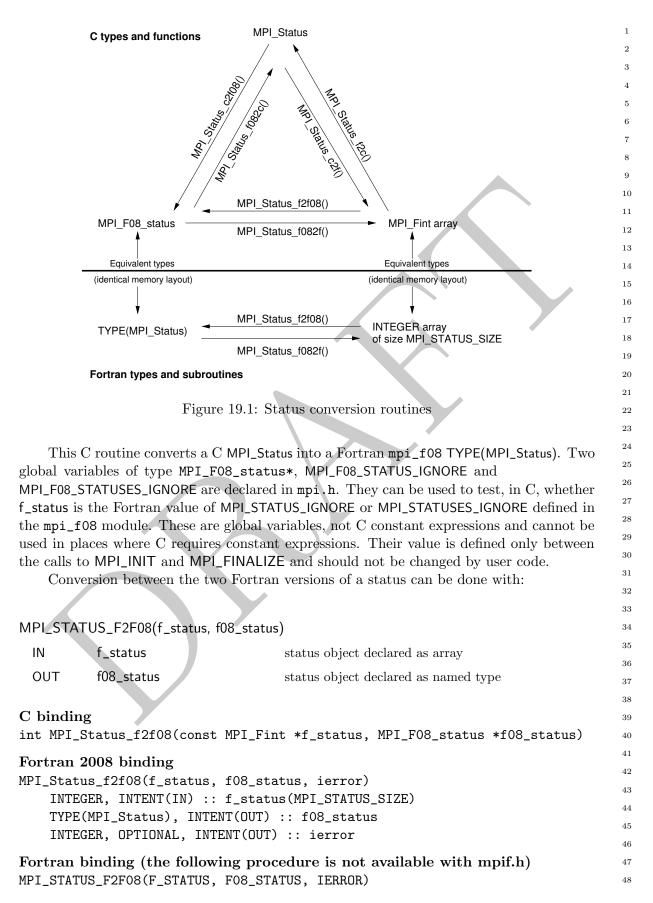
43

44

45

46 47

1 If f_status is a valid Fortran status, but not the Fortran value of MPI_STATUS_IGNORE  $\mathbf{2}$ or MPI_STATUSES_IGNORE, then MPI_Status_f2c returns in c_status a valid C status with 3 the same content. If f_status is the Fortran value of MPI_STATUS_IGNORE or 4 MPI_STATUSES_IGNORE, or if f_status is not a valid Fortran status, then the call is erroneous. 5In C, such an f_status array can be defined with MPI_Fint f_status[ 6 MPI_F_STATUS_SIZE]. Within this array, one can use in C the indexes MPI_F_SOURCE,  $\overline{7}$ MPI_F_TAG, and MPI_F_ERROR, to access the same elements as in Fortran with MPI_SOURCE, 8 MPI_TAG and MPI_ERROR. The C indexes are 1 less than the corresponding indexes in 9 Fortran due to the different default array start indexes in both languages. 10 The C status has the same source, tag and error code values as the Fortran status, 11and returns the same answers when queried for count, elements, and cancellation. The 12conversion function may be called with a Fortran status argument that has an undefined 13error field, in which case the value of the error field in the C status argument is undefined. 14Two global variables of type MPI_Fint*, MPI_F_STATUS_IGNORE and 15MPI_F_STATUSES_IGNORE are declared in mpi.h. They can be used to test, in C, whether 16f_status is the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE defined in 17 the mpi module or mpif.h. These are global variables, not C constant expressions and 18 cannot be used in places where C requires constant expressions. Their value is defined only 19between the calls to MPI_INIT and MPI_FINALIZE and should not be changed by user code. 20To do the conversion in the other direction, we have the following: 21int MPI_Status_c2f(const MPI_Status *c_status, MPI_Fint *f_status) 22 23This call converts a C status into a Fortran status, and has a behavior similar to 24MPI_Status_f2c. That is, the value of c_status must not be either MPI_STATUS_IGNORE or 25MPI_STATUSES_IGNORE. 26Advice to users. There exists no separate conversion function for arrays of statuses, 27since one can simply loop through the array, converting each status with the routines 28in Figure 19.1. (End of advice to users.) 29 30 *Rationale.* The handling of MPI_STATUS_IGNORE is required in order to layer libraries 31with only a C wrapper: if the Fortran call has passed MPI_STATUS_IGNORE, then the 32 C wrapper must handle this correctly. Note that this constant need not have the 33 same value in Fortran and C. If MPI_Status_f2c were to handle MPI_STATUS_IGNORE, 34 then the type of its result would have to be MPI_Status**, which was considered an 35inferior solution. (End of rationale.) 36 37 Using the mpi_f08 Fortran module, a status is declared as TYPE(MPI_Status). The C 38 type MPI_F08_status can be used to pass a Fortran TYPE(MPI_Status) argument into a C 39 routine. Figure 19.1 illustrates all status conversion routines. Some are only available in 40 C, some in both C and the Fortran mpi and mpi_f08 interfaces (but not in the mpif.h 41 interface). 4243 int MPI_Status_f082c(const MPI_F08_status *f08_status, 44MPI_Status *c_status) 45This C routine converts a Fortran mpi_f08 TYPE(MPI_Status) into a C MPI_Status. 4647int MPI_Status_c2f08(const MPI_Status *c_status, 48 MPI_F08_status *f08_status)



```
1
 INTEGER :: F_STATUS(MPI_STATUS_SIZE), IERROR
\mathbf{2}
 TYPE(MPI_Status) :: F08_STATUS
3
 This routine converts a Fortran INTEGER, DIMENSION(MPI_STATUS_SIZE) status array
4
 into a Fortran mpi_f08 TYPE(MPI_Status).
5
6
\overline{7}
 MPI_STATUS_F082F(f08_status, f_status)
8
 IN
 f08_status
 status object declared as named type
9
 OUT
10
 f_status
 status object declared as array
11
12
 C binding
13
 int MPI_Status_f082f(const MPI_F08_status *f08_status, MPI_Fint *f_status)
14
 Fortran 2008 binding
15
 MPI_Status_f082f(f08_status, f_status, ierror)
16
 TYPE(MPI_Status), INTENT(IN) :: f08_status
17
 INTEGER, INTENT(OUT) :: f_status(MPI_STATUS_SIZE)
18
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20
 Fortran binding (the following procedure is not available with mpif.h)
21
 MPI_STATUS_F082F(F08_STATUS, F_STATUS, IERROR)
22
 TYPE(MPI_Status) :: F08_STATUS
23
 INTEGER :: F_STATUS(MPI_STATUS_SIZE), IERROR
24
 This routine converts a Fortran mpi_f08 TYPE(MPI_Status) into a Fortran INTEGER,
25
 DIMENSION(MPI_STATUS_SIZE) status array.
26
27
 19.3.6
 MPI Opaque Objects
28
29
 Unless said otherwise, opaque objects are "the same" in all languages: they carry the same
30
 information, and have the same meaning in both languages. The mechanism described
^{31}
 in the previous section can be used to pass references to MPI objects from language to
32
 language. An object created in one language can be accessed, modified or freed in another
33
 language.
34
 We examine below in more detail issues that arise for each type of MPI object.
35
36
 Datatypes
37
38
 Datatypes encode the same information in all languages. E.g., a datatype accessor like
39
 MPI_TYPE_GET_EXTENT will return the same information in all languages. If a datatype
40
 defined in one language is used for a communication call in another language, then the
41
 message sent will be identical to the message that would be sent from the first language:
42
 the same communication buffer is accessed, and the same representation conversion is per-
43
 formed, if needed. All predefined datatypes can be used in datatype constructors in any
44
 language. If a datatype is committed, it can be used for communication in any language.
45
 The function MPI_GET_ADDRESS returns the same value in all languages. Note that
46
 we do not require that the constant MPI_BOTTOM have the same value in all languages (see
47
 Section 19.3.9).
48
```

```
Example 19.12
 2
! FORTRAN CODE
REAL :: R(5)
INTEGER :: TYPE, IERR, AOBLEN(1), AOTYPE(1)
INTEGER (KIND=MPI_ADDRESS_KIND) :: AODISP(1)
 6
! create an absolute datatype for array R
AOBLEN(1) = 5
CALL MPI_GET_ADDRESS(R, AODISP(1), IERR)
 10
AOTYPE(1) = MPI_REAL
 11
CALL MPI_TYPE_CREATE_STRUCT(1, AOBLEN, AODISP, AOTYPE, TYPE, IERR)
 12
CALL C_ROUTINE(TYPE)
 13
 14
/* C code */
 15
 16
void C_ROUTINE(MPI_Fint *ftype)
 17
{
 18
 int count = 5;
 19
 int lens[2] = {1,1};
 20
 MPI_Aint displs[2];
 21
 MPI_Datatype types[2], newtype;
 22
 23
 /* create an absolute datatype for buffer that consists
 */
 24
 /* of count, followed by R(5)
 */
 25
 26
 MPI_Get_address(&count, &displs[0]);
 27
 displs[1] = 0;
 28
 types[0] = MPI_INT;
 29
 types[1] = MPI_Type_f2c(*ftype);
 30
 MPI_Type_create_struct(2, lens, displs, types, &newtype);
 31
 MPI_Type_commit(&newtype);
 32
 33
 MPI_Send(MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
 34
 /* the message sent contains an int count of 5, followed
 */
 35
 /* by the 5 REAL entries of the Fortran array R.
 */
 36
}
 37
```

38 Advice to implementors. The following implementation can be used: MPI addresses, 39 as returned by MPI_GET_ADDRESS, will have the same value in all languages. One obvious choice is that MPI addresses be identical to regular addresses. The address 40 41 is stored in the datatype, when datatypes with absolute addresses are constructed. 42When a send or receive operation is performed, then addresses stored in a datatype are interpreted as displacements that are all augmented by a base address. This base 4344address 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 4546a call with a regular buffer argument: in both cases the base address is **buf**. On the 47other hand, if MPI_BOTTOM is not zero, then the implementation has to be slightly 48 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.)

10 Callback Functions 11

12MPI calls may associate callback functions with MPI objects: error handlers are associated 13 with communicators, files, windows, and sessions; attribute copy and delete functions are 14associated with attribute keys; reduce operations are associated with operation objects, etc. 15In a multilanguage environment, a function passed in an MPI call in one language may be 16invoked by an MPI call in another language. MPI implementations must make sure that 17such 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 19tag is set when the callback function is passed in by the library function (which is 20presumably 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 22 advice to implementors.) 23

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 365. (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.)

38 **Reduce Operations** 39

40All predefined named and unnamed datatypes as listed in Section 6.9.2 can be used in the 41 listed predefined operations independent of the programming language from which the MPI 42routine is called.

- Advice to users. Reduce operations receive as one of their arguments the datatype 4445of the operands. Thus, one can define "polymorphic" reduce operations that work for C and Fortran datatypes. (End of advice to users.) 46
- 47 48

43

1

2

3

4

5

6

7

8

9

18

21

24

25

26

27

28

29

30

 31 32

33 34

35

36

#### 19.3.7 Attributes

Attribute keys can be allocated in one language and freed in another. Similarly, attribute values can be set in one language and accessed in another. To achieve this, attribute keys will be allocated in an integer range that is valid all languages. The same holds true for system-defined attribute values (such as MPI_TAG_UB, MPI_WTIME_IS_GLOBAL, etc.).

Attribute keys declared in one language are associated with copy and delete functions in that language (the functions provided by the MPI_XXX_CREATE_KEYVAL call). When a communicator is duplicated, for each attribute, the corresponding copy function is called, using the right calling convention for the language of that function; and similarly, for the delete callback function.

Advice to implementors. This requires that attributes be tagged either as "C" or "Fortran" and that the language tag be checked in order to use the right calling convention for the callback function. (*End of advice to implementors.*)

The attribute manipulation functions described in Section 7.7 defines attributes arguments to be of type void* in C, and of type INTEGER, in Fortran. On some systems, INTEGERs will have 32 bits, while C pointers will have 64 bits. This is a problem if communicator attributes are used to move information from a Fortran caller to a C callee, or vice-versa.

MPI behaves as if it stores, internally, address sized attributes. If Fortran INTEGERs are smaller, then the (deprecated) Fortran function MPI_ATTR_GET will return the least significant part of the attribute word; the (deprecated) Fortran function MPI_ATTR_PUT will set the least significant part of the attribute word, which will be sign extended to the entire word. (These two functions may be invoked explicitly by user code, or implicitly, by attribute copying callback functions.)

As for addresses, new functions are provided that manipulate Fortran address sized attributes, and have the same functionality as the old functions in C. These functions are described in Section 7.7. Users are encouraged to use these new functions.

MPI supports two types of attributes: address-valued (pointer) attributes, and integervalued attributes. C attribute functions put and get address-valued attributes. Fortran attribute functions put and get integer-valued attributes. When an integer-valued attribute is accessed from C, then MPI_XXX_get_attr will return the address of (a pointer to) the integer-valued attribute, which is a pointer to MPI_Aint if the attribute was stored with Fortran MPI_XXX_SET_ATTR, and a pointer to int if it was stored with the deprecated Fortran MPI_ATTR_PUT. When an address-valued attribute is accessed from Fortran, then MPI_XXX_GET_ATTR will convert the address into an integer and return the result of this conversion. This conversion is lossless if new style attribute functions are used, and an integer of kind MPI_ADDRESS_KIND is returned. The conversion may cause truncation if deprecated attribute functions are used. In C, the deprecated routines MPI_Attr_put and MPI_Attr_get behave identical to MPI_Comm_set_attr and MPI_Comm_get_attr.

#### Example 19.13

A. Setting an attribute value in C

 $\overline{7}$ 

 24 

 31 

```
1
 int set_val = 3;
\mathbf{2}
 struct foo set_struct;
3
4
 /* Set a value that is a pointer to an int */
5
6
 MPI_Comm_set_attr(MPI_COMM_WORLD, keyval1, &set_val);
\overline{7}
 /* Set a value that is a pointer to a struct */
8
 MPI_Comm_set_attr(MPI_COMM_WORLD, keyval2, &set_struct);
9
 /* Set an integer value */
10
 MPI_Comm_set_attr(MPI_COMM_WORLD, keyval3, (void *) 17);
11
 B. Reading the attribute value in C
12
13
 int flag, *get_val;
14
 struct foo *get_struct;
15
16
 /* Upon successful return, get_val == &set_val
17
 (and therefore *get_val == 3) */
18
 MPI_Comm_get_attr(MPI_COMM_WORLD, keyval1, &get_val, &flag);
19
 /* Upon successful return, get_struct == &set_struct */
20
 MPI_Comm_get_attr(MPI_COMM_WORLD, keyval2, &get_struct, &flag);
21
 /* Upon successful return, get_val == (void*) 17 */
22
 /*
 i.e., (MPI_Aint) get_val == 17 */
23
 MPI_Comm_get_attr(MPI_COMM_WORLD, keyval3, &get_val, &flag);
24
25
 C. Reading the attribute value with (deprecated) Fortran MPI-1 calls
26
27
 LOGICAL FLAG
28
 INTEGER IERR, GET_VAL, GET_STRUCT
29
30
 ! Upon successful return, GET_VAL == &set_val, possibly truncated
^{31}
 CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL1, GET_VAL, FLAG, IERR)
32
 ! Upon successful return, GET_STRUCT == &set_struct, possibly truncated
33
 CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL2, GET_STRUCT, FLAG, IERR)
34
 ! Upon successful return, GET_VAL == 17
35
 CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL3, GET_VAL, FLAG, IERR)
36
37
 D. Reading the attribute value with Fortran MPI-2 calls
38
39
 LOGICAL FLAG
40
 INTEGER IERR
41
 INTEGER (KIND=MPI_ADDRESS_KIND) GET_VAL, GET_STRUCT
42
43
 ! Upon successful return, GET_VAL == &set_val
44
 CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL1, GET_VAL, FLAG, IERR)
45
 ! Upon successful return, GET_STRUCT == &set_struct
46
 CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL2, GET_STRUCT, FLAG, IERR)
47
 ! Upon successful return, GET_VAL == 17
48
 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 2 INTEGER IERR, VAL 3 VAL = 74 CALL MPI_ATTR_PUT(MPI_COMM_WORLD, KEYVAL, VAL, IERR) 56 B. Reading the attribute value in C 7 int flag; 9 int *value; 10 11 /* Upon successful return, value points to internal MPI storage and 12*value == (int) 7 */ 13 MPI_Comm_get_attr(MPI_COMM_WORLD, keyval, &value, &flag); 1415C. Reading the attribute value with (deprecated) Fortran MPI-1 calls 1617 LOGICAL FLAG 18 INTEGER IERR, VALUE 19 20! Upon successful return, VALUE == 7 21CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR) 22 23D. Reading the attribute value with Fortran MPI-2 calls  24 25LOGICAL FLAG 26INTEGER IERR 27INTEGER (KIND=MPI_ADDRESS_KIND) VALUE 2829! Upon successful return, VALUE == 7 (sign extended) 30 CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR) 3132 **Example 19.15** A. Setting an attribute value via a Fortran MPI-2 call 33 34 INTEGER IERR 35INTEGER(KIND=MPI_ADDRESS_KIND) VALUE1 36 INTEGER(KIND=MPI_ADDRESS_KIND) VALUE2 37 VALUE1 = 4238 VALUE2 = INT(2, KIND=MPI_ADDRESS_KIND) ** 40 39 40 CALL MPI_COMM_SET_ATTR(MPI_COMM_WORLD, KEYVAL1, VALUE1, IERR) 41 CALL MPI_COMM_SET_ATTR(MPI_COMM_WORLD, KEYVAL2, VALUE2, IERR) 4243 B. Reading the attribute value in C 4445464748

```
1
 int flag;
\mathbf{2}
 MPI_Aint *value1, *value2;
3
4
 /* Upon successful return, value1 points to internal MPI storage and
5
 *value1 == 42 */
6
 MPI_Comm_get_attr(MPI_COMM_WORLD, keyval1, &value1, &flag);
7
 /* Upon successful return, value2 points to internal MPI storage and
8
 *value2 == 2^40 */
9
 MPI_Comm_get_attr(MPI_COMM_WORLD, keyval2, &value2, &flag);
10
 C. Reading the attribute value with (deprecated) Fortran MPI-1 calls
11
12
 LOGICAL FLAG
13
 INTEGER IERR, VALUE1, VALUE2
14
15
 ! Upon successful return, VALUE1 == 42
16
 CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL1, VALUE1, FLAG, IERR)
17
 ! Upon successful return, VALUE2 == 2<sup>40</sup>, or 0 if truncation
18
 ! needed (i.e., the least significant part of the attribute word)
19
 CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL2, VALUE2, FLAG, IERR)
20
 D. Reading the attribute value with Fortran MPI-2 calls
21
22
 LOGICAL FLAG
23
 INTEGER IERR
24
 INTEGER (KIND=MPI_ADDRESS_KIND) VALUE1, VALUE2
25
26
 ! Upon successful return, VALUE1 == 42
27
 CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL1, VALUE1, FLAG, IERR)
28
 ! Upon successful return, VALUE2 == 2^40
29
 CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL2, VALUE2, FLAG, IERR)
30
31
 The predefined MPI attributes can be integer valued or address-valued. Predefined
32
 integer valued attributes, such as MPI_TAG_UB, behave as if they were put by a call to
33
 the deprecated Fortran routine MPI_ATTR_PUT, i.e., in Fortran,
34
 MPI_COMM_GET_ATTR(MPI_COMM_WORLD, MPI_TAG_UB, val, flag, ierr) will return
35
 in val the upper bound for tag value; in C, MPI_Comm_get_attr(MPI_COMM_WORLD,
36
 MPI_TAG_UB, &p, &flag) will return in p a pointer to an int containing the upper bound
37
 for tag value.
38
 Address-valued predefined attributes, such as MPI_WIN_BASE behave as if they were
39
 put by a C call, i.e., in Fortran, MPI_WIN_GET_ATTR(win, MPI_WIN_BASE, val, flag,
40
 ierror) will return in val the base address of the window, converted to an integer. In C,
41
 MPI_Win_get_attr(win, MPI_WIN_BASE, &p, &flag) will return in p a pointer to the window
42
 base, cast to (void *).
43
 The design is consistent with the behavior specified for predefined at-
44
 Rationale.
45
 tributes, and ensures that no information is lost when attributes are passed from
46
 language to language. Because the language interoperability for predefined attributes
47
 was defined based on MPI_ATTR_PUT, this definition is kept for compatibility reasons
48
 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 which must be allocated to hold the largest possible such string, rather than the maximum number of printable characters the string could contain.

Advice to users. This definition means that it is safe in C to allocate a buffer to receive a string using a declaration like

char name [MPI_MAX_OBJECT_NAME];

(End of advice to users.)

Also constant "addresses," i.e., special values for reference arguments that are not handles, such as MPI_BOTTOM or MPI_STATUS_IGNORE may have different values in different languages.

The current MPI standard specifies that MPI_BOTTOM can be used in Rationale. 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.)

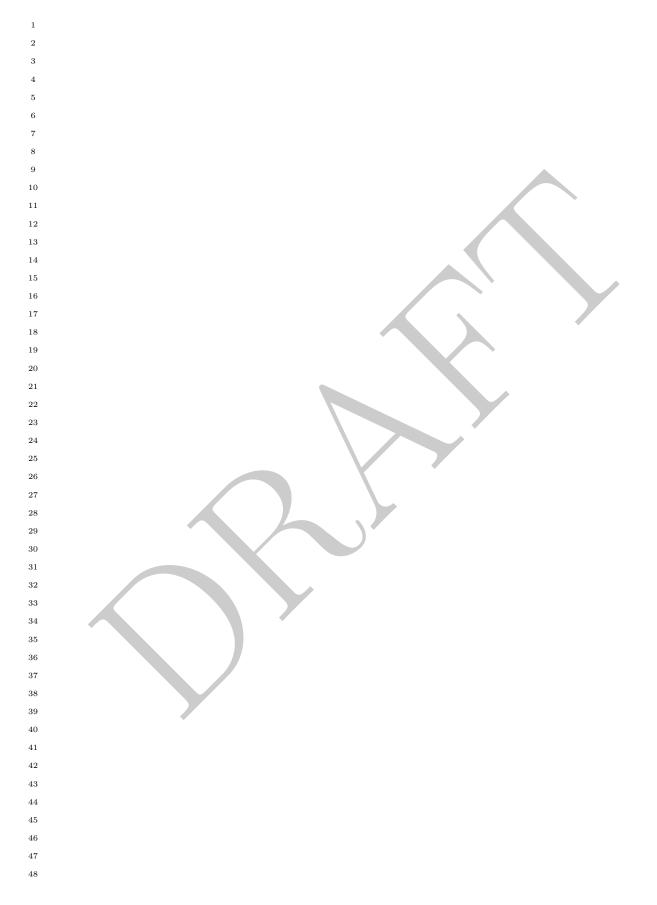
### Unofficial Draft for Comment Only

 24 

```
1
 19.3.10 Interlanguage Communication
\mathbf{2}
 The type matching rules for communication in MPI are not changed: the datatype specifi-
3
 cation for each item sent should match, in type signature, the datatype specification used to
4
 receive this item (unless one of the types is MPI_PACKED). Also, the type of a message item
5
 should match the type declaration for the corresponding communication buffer location,
6
 unless the type is MPI_BYTE or MPI_PACKED. Interlanguage communication is allowed if it
7
 complies with these rules.
8
9
 Example 19.16 In the example below, a Fortran array is sent from Fortran and received
10
 in C.
11
 ! FORTRAN CODE
12
 SUBROUTINE MYEXAMPLE()
13
14
 USE mpi_f08
 REAL :: R(5)
15
16
 INTEGER :: IERR, MYRANK, AOBLEN(1)
17
 TYPE(MPI_Datatype) :: TYPE, AOTYPE(1)
 INTEGER (KIND=MPI_ADDRESS_KIND) :: AODISP(1)
18
19
 ! create an absolute datatype for array R
20
21
 AOBLEN(1) = 5
 CALL MPI_GET_ADDRESS(R, AODISP(1), IERR)
22
 AOTYPE(1) = MPI_REAL
23
 CALL MPI_TYPE_CREATE_STRUCT(1, AOBLEN, AODISP, AOTYPE, TYPE, IERR)
^{24}
 CALL MPI_TYPE_COMMIT(TYPE, IERR)
25
26
 CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYRANK, IERR)
27
 IF (MYRANK.EQ.O) THEN
28
 CALL MPI_SEND(MPI_BOTTOM, 1, TYPE, 1, 0, MPI_COMM_WORLD, IERR)
29
 ELSE
30
 CALL C_ROUTINE(TYPE%MPI_VAL)
^{31}
 END IF
32
 END SUBROUTINE
33
34
35
 /* C code */
36
37
 void C_ROUTINE(MPI_Fint *fhandle)
38
 {
39
 MPI_Datatype type;
40
 MPI_Status status;
41
42
 type = MPI_Type_f2c(*fhandle);
43
44
 MPI_Recv(MPI_BOTTOM, 1, type, 0, 0, MPI_COMM_WORLD, &status);
45
 }
46
47
 MPI implementors may weaken these type matching rules, and allow messages to be sent
```

⁴⁷ MPI implementors may weaken these type matching rules, and allow messages to be sent ⁴⁸ with Fortran types and received with C types, and vice versa, when those types match. I.e.,

if the Fortran type INTEGER is identical to the C type int, then an MPI implementation may allow data to be sent with datatype MPI_INTEGER and be received with datatype MPI_INT. However, such code is not portable.



# Annex A

# Language Bindings Summary

In this section we summarize the specific bindings for C and Fortran. First we present the constants, type definitions, info values and keys. Then we present the routine prototypes separately for each binding. Listings are alphabetical within chapter.

## A.1 Defined Values and Handles

### A.1.1 Defined Constants

The C and Fortran names are listed below. Constants with the type const int may also be implemented as literal integer constants substituted by the preprocessor.

	24
Error classes	25
C type: const int (or unnamed enum)	26
Fortran type: INTEGER	27
MPI_SUCCESS	28
MPI_ERR_BUFFER	29
MPI_ERR_COUNT	30
MPI_ERR_TYPE	31
MPI_ERR_TAG	32
MPI_ERR_COMM	33
MPI_ERR_RANK	34
MPI_ERR_REQUEST	35
MPI_ERR_ROOT	36
MPI_ERR_GROUP	37
MPI_ERR_OP	38
MPI_ERR_TOPOLOGY	39
MPI_ERR_DIMS	40
MPI_ERR_ARG	41
MPI_ERR_UNKNOWN	42
MPI_ERR_TRUNCATE	43
MPI_ERR_OTHER	44
MPI_ERR_INTERN	45
MPI_ERR_PENDING	46
(Continued on next page)	47
	48

1	Error classes (continued)
2	C type: const int (or unnamed enum)
3	Fortran type: INTEGER
4	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	
10	
11	
	MPI_ERR_DUP_DATAREP
13	MPI_ERR_FILE_EXISTS
14	MPI_ERR_FILE_IN_USE
15	MPI_ERR_FILE
16	MPI_ERR_INFO_KEY
17	MPI_ERR_INFO_NOKEY
18	MPI_ERR_INFO_VALUE
19	MPI_ERR_INFO
20	MPI_ERR_IO
21	MPI_ERR_KEYVAL
22	MPI_ERR_LOCKTYPE
23	MPI_ERR_NAME
24	MPI_ERR_NO_MEM
25	MPI_ERR_NOT_SAME
26	MPI_ERR_NO_SPACE
27	MPI_ERR_NO_SUCH_FILE
28	MPI_ERR_PORT
29	MPI_ERR_PROC_ABORTED
30	MPI_ERR_QUOTA
31	MPI_ERR_READ_ONLY
32	MPI_ERR_RMA_ATTACH
33	MPI_ERR_RMA_CONFLICT
34	MPI_ERR_RMA_RANGE
35	MPI_ERR_RMA_KANGE
36	
37	
	MPI_ERR_RMA_FLAVOR
38	MPI_ERR_SERVICE
39	MPI_ERR_SIZE
40	MPI_ERR_SPAWN
41	MPI_ERR_UNSUPPORTED_DATAREP
42	MPI_ERR_UNSUPPORTED_OPERATION
43	MPI_ERR_WIN
44	(Continued on next page)
45	
46	
47	
48	

	Error classes (continued)	1
	type: const int (or unnamed enum)	2
	rtran type: INTEGER	3
	PI_T_ERR_CANNOT_INIT	4
	PI_T_ERR_NOT_ACCESSIBLE	5
	PI_T_ERR_NOT_INITIALIZED	6
	PI_T_ERR_NOT_SUPPORTED	7
	PI_T_ERR_MEMORY	8
	PI_T_ERR_INVALID	9
	PI_T_ERR_INVALID_INDEX	10
	PI_T_ERR_INVALID_ITEM	11
	PI_T_ERR_INVALID_SESSION	12
	PI_T_ERR_INVALID_HANDLE	13
	PI_T_ERR_INVALID_NAME	14
	PI_T_ERR_OUT_OF_HANDLES	15
M	PI_T_ERR_OUT_OF_SESSIONS	16
M	PI_T_ERR_CVAR_SET_NOT_NOW	17
M	PI_T_ERR_CVAR_SET_NEVER	18
M	PI_T_ERR_PVAR_NO_WRITE	19
M	PI_T_ERR_PVAR_NO_STARTSTOP	20
M	PI_T_ERR_PVAR_NO_ATOMIC	21
M	PI_ERR_LASTCODE	22
	Buffer Address Constants	23
C type: void * cons		24
01	ined memory location) ¹	25 26
MPI_BOTTOM		20 27
MPI_IN_PLACE		21
	ran these constants are not usable for initialization	20
	ignment. See Section 2.5.4.	30
I I III I I I	5	31
	Assorted Constants	32
C ·	type: const int (or unnamed enum)	33
	rtran type: INTEGER	34
	PI_PROC_NULL	35
	PI_ANY_SOURCE	36
	PI_ANY_TAG	37
	PI_UNDEFINED	38
	PI_BSEND_OVERHEAD	39
	PI_KEYVAL_INVALID	40
	PI_LOCK_EXCLUSIVE	41
	PI_LOCK_SHARED	42
M	PI_ROOT	43
		44
т	No Process Message Handle	45
	: MPI_Message	46
• -	n type: INTEGER or TYPE(MPI_Message)	47
	MESSAGE_NO_PROC	48

1	Fortran Support Method Specific Constants
2	
	Fortran type: LOGICAL
3	MPI_SUBARRAYS_SUPPORTED (Fortran only)
4	MPI_ASYNC_PROTECTS_NONBLOCKING (Fortran only)
5	
6	Status array size and reserved index values (Fortran only)
7	Fortran type: INTEGER
8	MPI_STATUS_SIZE
9	MPI_SOURCE
10	MPI_TAG
11	MPI_ERROR
12	
13	
14	Fortran status array size and reserved index values (C only)
15	C type: int
16	MPI_F_STATUS_SIZE
17	MPI_F_SOURCE
18	MPI_F_TAG
19	MPI_F_ERROR
20	
21	Variable Address Size (Fortran only)
22	Fortran type: INTEGER
23	MPI_ADDRESS_KIND
24	MPI_COUNT_KIND
25	MPI_INTEGER_KIND
26	MPI_OFFSET_KIND
27	
28	Error-handling specifiers
29	C type: MPI_Errhandler
30	Fortran type: INTEGER or TYPE(MPI_Errhandler)
31	MPI_ERRORS_ARE_FATAL
32	MPI_ERRORS_RETURN
33	
34	Maximum Sizes for Strings
35	C type: const int (or unnamed enum)
36	Fortran type: INTEGER
37	MPI_MAX_DATAREP_STRING
38	MPI_MAX_ERROR_STRING
39	MPI_MAX_INFO_KEY
40	MPI_MAX_INFO_VAL
41	MPI_MAX_LIBRARY_VERSION_STRING
42	MPI_MAX_OBJECT_NAME
43	MPI_MAX_PORT_NAME
44	MPI_MAX_PROCESSOR_NAME
45	MPI_MAX_FROM_GROUP_TAG
46	MPI_MAX_PSET_NAME_LEN
47	
48	

 $\checkmark$ 

Named Predefined Datatypes	C types	1
C type: MPI_Datatype		2
Fortran type: INTEGER		3
or TYPE(MPI_Datatype)		4
MPI_CHAR	char	5
	(treated as printable character)	6
MPI_SHORT	signed short int	7
MPI_INT	signed int	8
MPI_LONG	signed long	9
MPI_LONG_LONG_INT	signed long long	10
$MPI_LONG_LONG$ (as a synonym)	signed long long	11
MPI_SIGNED_CHAR	signed char	12
	(treated as integral value)	13
MPI_UNSIGNED_CHAR	unsigned char	14
	(treated as integral value)	15
MPI_UNSIGNED_SHORT	unsigned short	16
MPI_UNSIGNED	unsigned int	17
MPI_UNSIGNED_LONG	unsigned long	18
MPI_UNSIGNED_LONG_LONG	unsigned long long	19
MPI_FLOAT	float	20
MPI_DOUBLE	double	21
MPI_LONG_DOUBLE	long double	22
MPI_WCHAR	wchar_t	23
	(defined in <stddef.h>)</stddef.h>	24
	(treated as printable character)	25
MPI_C_BOOL	_Bool	26
MPI_INT8_T	int8_t	27
MPI_INT16_T	int16_t	28
MPI_INT32_T	int32_t	29
MPI_INT64_T	int64_t	30
MPI_UINT8_T	uint8_t	31
MPI_UINT16_T	uint16_t	32
MPI_UINT32_T	uint32_t	33
MPI_UINT64_T	uint64_t	34
MPI_AINT	MPI_Aint	35
MPI_COUNT	MPI_Count	36
MPI_OFFSET	MPI_Offset	37
MPI_C_COMPLEX	float _Complex	38
MPI_C_FLOAT_COMPLEX	float _Complex	39
MPI_C_DOUBLE_COMPLEX	double _Complex	40
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex	41
MPI_BYTE	(any C type)	42
MPI_PACKED	(any C type)	43
		44

Unofficial Draft for Comment Only

1	Named Predefined Datatypes	Fortran types
2	C type: MPI_Datatype	<i>v</i> <b>i</b>
3	Fortran type: INTEGER	
4	or TYPE(MPI_Datatype)	
5	MPI_INTEGER	INTEGER
6	 MPI_REAL	REAL
7		DOUBLE PRECISION
8	MPI_COMPLEX	COMPLEX
9	MPI_LOGICAL	LOGICAL
10		CHARACTER(1)
11	 MPI_AINT	INTEGER (KIND=MPI_ADDRESS_KIND)
12	 MPI_COUNT	INTEGER (KIND=MPI_COUNT_KIND)
13	MPI_OFFSET	INTEGER (KIND=MPI_OFFSET_KIND)
14	MPI_BYTE	(any Fortran type)
15	MPI_PACKED	(any Fortran type)
16		
17	Named Predefined Datatype	$\mathbf{s}^1 \mid \mathbf{C} + + \mathbf{types}$
18	$\mathrm{C}\ \mathrm{type}:\ MPI_Datatype$	
19	Fortran type: INTEGER	
20	or TYPE(MPI_Datatype)	
21	MPI_CXX_BOOL	bool
22	MPI_CXX_FLOAT_COMPLEX	<pre>std::complex<float></float></pre>
23	MPI_CXX_DOUBLE_COMPLEX	<pre>std::complex<double></double></pre>
24	MPI_CXX_LONG_DOUBLE_COMPL	
25	¹ If an accompanying $C++$ comp	0.
26	MPI datatypes in this table are	not defined.
27		
28	Optional datatypes (F	ortran) Fortran types
29	C type: MPI_Datatype	
30	Fortran type: INTEGER	
31 32	or TYPE(MPI_Datatype)	
32	MPI_DOUBLE_COMPLEX	DOUBLE COMPLEX
33 34	MPI_INTEGER1	INTEGER*1
	MPI_INTEGER2	INTEGER*2
35	MPI_INTEGER4	INTEGER*4
36 27	MPI_INTEGER8	INTEGER*8
37	MPI_INTEGER16	INTEGER*16
38 20	MPI_REAL2	REAL*2
39 40	MPI_REAL4	REAL*4
40 41	MPI_REAL8	REAL*8
41 42	MPI_REAL16	REAL*16
	MPI_COMPLEX4	COMPLEX*4
43 44		COMPLEX*8
	MPI_COMPLEX16	COMPLEX*16
45 46	MPI_COMPLEX32	COMPLEX*32
40 47		
47		
40		

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	12
Fortran type: INTEGER or TYPE(MPI_Datatype)	13
MPI_2REAL	14
MPI_2DOUBLE_PRECISION	15
MPI_2INTEGER	16
Reserved communicators	17
C type: MPI_Comm	18
Fortran type: INTEGER or TYPE(MPI_Comm)	19
MPI_COMM_WORLD	20
MPI_COMM_SELF	21
Communicator split type constants	22 23
C type: const int (or unnamed enum)	24
Fortran type: INTEGER	25
MPI_COMM_TYPE_SHARED	26
MPI_COMM_TYPE_HW_UNGUIDED	27
MPI_COMM_TYPE_HW_GUIDED	28
	29
Results of communicator and group comparisons	30
C type: const int (or unnamed enum)	31
Fortran type: INTEGER	32
MPI_IDENT	33
MPI_CONGRUENT	34
MPI_SIMILAR	35
MPI_UNEQUAL	36
Environmental inquiry info key	37
C type: MPI_Info	38
Fortran type: INTEGER or TYPE(MPI_Info)	39
MPI_INFO_ENV	40
	41
Environmental inquiry keys	42
C type: const int (or unnamed enum)	43
Fortran type: INTEGER	44
MPI_TAG_UB	45
MPI_IO	46 47
MPI_HOST	
MPI_WTIME_IS_GLOBAL	48

2C type: MPI_Op3Fortran type: INTEGER or TYPE(MPI_Op)4MPI_MAX5MPI_MAX6MPI_SUM7MPI_PROD8MPI_MAXLOC9MPI_MINLOC10MPI_BOR11MPI_BOR12MPI_LAND13MPI_LAND14MPI_LOR15MPI_REPLACE	
4MPI_MAX5MPI_MIN6MPI_SUM7MPI_PROD8MPI_MAXLOC9MPI_MINLOC10MPI_BAND11MPI_BOR12MPI_BXOR13MPI_LAND14MPI_LOR15MPI_LXOR	
4MPI_MAX5MPI_MIN6MPI_SUM7MPI_PROD8MPI_MAXLOC9MPI_MINLOC10MPI_BAND11MPI_BOR12MPI_BXOR13MPI_LAND14MPI_LOR15MPI_LXOR	
6MPI_SUM7MPI_PROD8MPI_MAXLOC9MPI_MINLOC10MPI_BAND11MPI_BOR12MPI_BXOR13MPI_LAND14MPI_LOR15MPI_LXOR	
6MPI_SUM7MPI_PROD8MPI_MAXLOC9MPI_MINLOC10MPI_BAND11MPI_BOR12MPI_BXOR13MPI_LAND14MPI_LOR15MPI_LXOR	
7MPI_PROD8MPI_MAXLOC9MPI_MINLOC10MPI_BAND11MPI_BOR12MPI_BXOR13MPI_LAND14MPI_LOR15MPI_LXOR	
8MPI_MAXLOC9MPI_MINLOC10MPI_BAND11MPI_BOR12MPI_BXOR13MPI_LAND14MPI_LOR15MPI_LXOR	
9MPI_MINLOC10MPI_BAND11MPI_BOR12MPI_BXOR13MPI_LAND14MPI_LOR15MPI_LXOR	
10MPI_BAND11MPI_BOR12MPI_BXOR13MPI_LAND14MPI_LOR15MPI_LXOR	
11MPI_BOR12MPI_BXOR13MPI_LAND14MPI_LOR15MPI_LXOR	
12MPI_BXOR13MPI_LAND14MPI_LOR15MPI_LXOR	
13     MPI_LAND       14     MPI_LOR       15     MPI_LXOR	
14     MPI_LOR       15     MPI_LXOR	
15 MPI_LXOR	
¹⁶ MPI_REPLACE	
¹⁷ MPI_NO_OP	
¹⁹ Null Handles	
²⁰ C/Fortran name	
²¹ C type / Fortran type	
²² MPI_GROUP_NULL	
²³ MPI_Group / INTEGER or TYPE(MPI_Group)	
²⁴ MPI_COMM_NULL	
²⁵ MPI_Comm / INTEGER or TYPE(MPI_Comm)	
²⁶ MPI_DATATYPE_NULL	
²⁷ MPI_Datatype / INTEGER or TYPE(MPI_Datatype)	
²⁸ MPI_REQUEST_NULL	
²⁹ MPI_Request / INTEGER or TYPE(MPI_Request)	
³⁰ MPI_OP_NULL	
³¹ MPI_Op / INTEGER or TYPE(MPI_Op)	
³² MPI_ERRHANDLER_NULL	
³³ MPI_Errhandler / INTEGER or TYPE(MPI_Errhandler)	
³⁴ MPI_FILE_NULL	
³⁵ MPI_File / INTEGER or TYPE(MPI_File)	
³⁶ MPI_INFO_NULL	
³⁷ MPI_Info / INTEGER or TYPE(MPI_Info)	
³⁸ MPI_SESSION_NULL	
³⁹ MPI_Session / INTEGER	
MFI_ME33AGE_NOLE	
43 MPI_Message / INTEGER or TYPE(MPI_Message)	
45 Empty group	
46 C type: MPI_Group	
47 Fortran type: INTEGER or TYPE(MPI_Group)	
48 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_COP	, , , , , , , , , , , , , , , , , , , ,
MPI_Comm_copy_attr_f	unction
/ COMM_COPY_ATTR_FUNC	CTION / PROCEDURE(MPI_Comm_copy_attr_function) ¹ )
MPI_COMM_DUP_FN	
MPI_Comm_copy_attr_f	unction
/ COMM_COPY_ATTR_FUNC	CTION / PROCEDURE(MPI_Comm_copy_attr_function) 1)
MPI_COMM_NULL_DEL	ETE_FN
MPI_Comm_delete_attr	_function
/ COMM_DELETE_ATTR_FU	· · · · · · · · · · · · · · · · · · ·
MPI_WIN_NULL_COPY_	FN
MPI_Win_copy_attr_fu	
/ WIN_COPY_ATTR_FUNC	TION / PROCEDURE(MPI_Win_copy_attr_function) 1 )
MPI_WIN_DUP_FN	
MPI_Win_copy_attr_fu	
/ WIN_COPY_ATTR_FUNC	
MPI_WIN_NULL_DELET	
MPI_Win_delete_attr_	
/ WIN_DELETE_ATTR_FU	· · · · · · · · · · · · · · · · · · ·
MPI_TYPE_NULL_COPY	
MPI_Type_copy_attr_f	
/ TYPE_COPY_ATTR_FUNC	CTION / PROCEDURE(MPI_Type_copy_attr_function) ¹ )
MPI_TYPE_DUP_FN	un at i an
MPI_Type_copy_attr_f	
/ TYPE_COPY_ATTR_FUNG MPI_TYPE_NULL_DELE	
MPI_Type_delete_attr	_
/ TYPE_DELETE_ATTR_FU	
MPI_CONVERSION_FN_	
MPI_Datarep_conversi	
/ DATAREP_CONVERSION	
/	mentors (on page 365) and advice to users (on page 365)
-	tran functions MPI_COMM_NULL_COPY_FN, in
Section 7.7.2.	

1	Deprecated predefined functions
2	C/Fortran name
3	C type / Fortran type with mpi module
4	MPI_NULL_COPY_FN
5	MPI_Copy_function / COPY_FUNCTION
6	MPI_DUP_FN
7	MPI_Copy_function / COPY_FUNCTION
8	MPI_NULL_DELETE_FN
9	MPI_Delete_function / DELETE_FUNCTION
10	
11	Predefined Attribute Keys
12	C type: const int (or unnamed enum)
13	Fortran type: INTEGER
14	MPI_APPNUM
15	MPI_LASTUSEDCODE
16	MPI_UNIVERSE_SIZE
17	MPI_WIN_BASE
18	MPI_WIN_DISP_UNIT
19	MPI_WIN_SIZE
20	MPI_WIN_CREATE_FLAVOR
21	MPI_WIN_MODEL
22	
23	MPI Window Create Flavors
24	C type: const int (or unnamed enum)
25	Fortran type: INTEGER
26	MPI_WIN_FLAVOR_CREATE
27	MPI_WIN_FLAVOR_ALLOCATE
28	MPI_WIN_FLAVOR_DYNAMIC
29	MPI_WIN_FLAVOR_SHARED
30	
31	MPI Window Models
32	C type: const int (or unnamed enum)
33	Fortran type: INTEGER
34	MPI_WIN_SEPARATE
35	MPI_WIN_UNIFIED
36	
37 38	
38	
40	
40	
42	
43	
43	
45	
46	
40	
48	

## Mode Constants C type: const int (or unnamed enum) Fortran type: INTEGER MPI_MODE_APPEND MPI_MODE_CREATE MPI_MODE_DELETE_ON_CLOSE MPI_MODE_EXCL MPI_MODE_NOCHECK MPI_MODE_NOPRECEDE MPI_MODE_NOPUT MPI_MODE_NOSTORE MPI_MODE_NOSUCCEED MPI_MODE_RDONLY MPI_MODE_RDWR MPI_MODE_SEQUENTIAL MPI_MODE_UNIQUE_OPEN MPI_MODE_WRONLY **Datatype Decoding Constants** C type: const int (or unnamed enum) Fortran type: INTEGER MPI_COMBINER_CONTIGUOUS MPI_COMBINER_DARRAY MPI_COMBINER_DUP MPI_COMBINER_F90_COMPLEX MPI_COMBINER_F90_INTEGER MPI_COMBINER_F90_REAL MPI_COMBINER_HINDEXED MPI_COMBINER_HVECTOR MPI_COMBINER_INDEXED_BLOCK MPI_COMBINER_HINDEXED_BLOCK MPI_COMBINER_INDEXED MPI_COMBINER_NAMED MPI_COMBINER_RESIZED MPI_COMBINER_STRUCT MPI_COMBINER_SUBARRAY MPI_COMBINER_VECTOR **Threads Constants** C type: const int (or unnamed enum) Fortran type: INTEGER MPI_THREAD_FUNNELED MPI_THREAD_MULTIPLE MPI_THREAD_SERIALIZED MPI_THREAD_SINGLE

1

 $\mathbf{2}$ 

3

10 11

12

13

14 15

16

17 18

19

20

21

22 23

24 25

26

27

28

29 30

31

32

33

34

35

36

37 38

39

40 41

42

43 44

45

1	File Operation Constants, Part 1
2	C type: const MPI_Offset (or unnamed enum)
3	Fortran type: INTEGER (KIND=MPI_OFFSET_KIND)
4	MPI_DISPLACEMENT_CURRENT
5	
6	File Operation Constants, Part 2
7	C type: const int (or unnamed enum)
8	Fortran type: INTEGER
9	MPI_DISTRIBUTE_BLOCK
10	MPI_DISTRIBUTE_CYCLIC
11	MPI_DISTRIBUTE_DFLT_DARG
12	MPI_DISTRIBUTE_NONE
13	MPI_ORDER_C
14	MPI_ORDER_FORTRAN
15	MPI_SEEK_CUR
16	MPI_SEEK_END
17	MPI_SEEK_SET
18	
19	F90 Datatype Matching Constants
20	C type: const int (or unnamed enum)
21	Fortran type: INTEGER
22	MPI_TYPECLASS_COMPLEX
23	MPI_TYPECLASS_INTEGER
24	MPI_TYPECLASS_REAL
25	
26	Constants Specifying Empty or Ignored Input
27	C/Fortran name
28	C type / Fortran type ¹
29	MPI_ARGVS_NULL
30	char*** / 2-dim. array of CHARACTER*(*)
31	MPI_ARGV_NULL
32 33	<pre>char** / array of CHARACTER*(*)</pre>
34	MPI_ERRCODES_IGNORE
35	int* / INTEGER array
36	MPI_STATUSES_IGNORE
37	MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)
38	or TYPE(MPI_Status), DIMENSION(*)
39	MPI_STATUS_IGNORE
40	MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)
41	or TYPE(MPI_Status)
42	MPI_UNWEIGHTED
43	int* / INTEGER array
44	MPI_WEIGHTS_EMPTY
45	$\frac{\text{int}* / \text{INTEGER array}}{1}$ Note that in Fortran these constants are not usable for initialization
46	expressions or assignment. See Section 2.5.4.
	expressions or assignment the decision 7.5.4
47	expressions of assignment. See Section 2.0.1.
47 48	

C type: MPI_Fint*	ing Ignored Input (no Fortran) equivalent to Fortran
MPI_F_STATUSES_IGNORE	MPI_STATUSES_IGNORE in mpi / mpif.h
MPI_F_STATUS_IGNORE	MPI_STATUS_IGNORE in mpi / mpif.h
C type: MPI_F08_status*	equivalent to Fortran
/PI_F08_STATUSES_IGNORE	MPI_STATUSES_IGNORE in mpi_f08
IPI_F08_STATUS_IGNORE	MPI_STATUS_IGNORE in mpi_f08
	stants and Fortran Parameters
Null handles used in t	he MPI tool information interface
MPI_T_ENUM_NULL	
MPI_T_enum	
MPI_T_CVAR_HANDLE_NU	
$MPI_T_cvar_handle$	
MPI_T_PVAR_HANDLE_NU	LL
$MPI_T_pvar_handle$	
MPI_T_PVAR_SESSION_NU	
MPI_T_pvar_session	
Vorbogity Loyola in th	e MPI tool information interface
C type: const int (or unn	
MPI_T_VERBOSITY_USER	· · · · · · · · · · · · · · · · · · ·
MPI_T_VERBOSITY_USER	
MPI_T_VERBOSITY_USER	
MPI_T_VERBOSITY_TUNE	
MPI_T_VERBOSITY_TUNE	
MPI_T_VERBOSITY_TUNE	ER_ALL
MPI_T_VERBOSITY_MPID	EV_BASIC
MPI_T_VERBOSITY_MPID	EV_DETAIL
MPI_T_VERBOSITY_MPID	EV_ALL

	862	ANNEX A. LANGUAGE BINDINGS S	UMMARY
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			
16		Constants describing the scope of a control variable	
17		in the MPI tool information interface	
18		C type: const int (or unnamed enum)	
19		MPI_T_SCOPE_CONSTANT	
20		MPI_T_SCOPE_READONLY	
21		MPI_T_SCOPE_LOCAL	
22 23		MPI_T_SCOPE_GROUP	
23 24		MPI_T_SCOPE_GROUP_EQ	
24 25		MPI_T_SCOPE_ALL	
26		MPI_T_SCOPE_ALL_EQ	
27		Additional constants used	
28		by the MPI tool information interface	
29		C type: MPI_T_pvar_handle	
30		MPI_T_PVAR_ALL_HANDLES	
31		MFI_I_FVAR_ALL_HANDLES	
32		Performance variables classes used by the	
33		MPI tool information interface	
34		C type: const int (or unnamed enum)	
35		MPI_T_PVAR_CLASS_STATE	
36		MPI_T_PVAR_CLASS_LEVEL	
37		MPI_T_PVAR_CLASS_SIZE	
38		MPI_T_PVAR_CLASS_PERCENTAGE	
39		MPI_T_PVAR_CLASS_HIGHWATERMARK	
40		MPI_T_PVAR_CLASS_LOWWATERMARK	
41		MPI_T_PVAR_CLASS_COUNTER	
42		MPI_T_PVAR_CLASS_AGGREGATE	
43		MPI_T_PVAR_CLASS_TIMER	
44		MPI_T_PVAR_CLASS_GENERIC	
45			
46			
47			
48			

Source event ordering guarantees in the	1
<b>MPI</b> tool information interface	2
C type: MPI_T_source_order	3
MPI_T_SOURCE_ORDERED	4
MPI_T_SOURCE_UNORDERED	5
	6
Callback safety requirement levels used in the	7
MPI tool information interface	8
C type: MPI_T_cb_safety	9
MPI_T_CB_REQUIRE_NONE	10
MPI_T_CB_REQUIRE_MPI_RESTRICTED	11
MPI_T_CB_REQUIRE_THREAD_SAFE	12
MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE	13
	14
A.1.2 Types	15
The following are defined C type definitions included in the flamminh	16
The following are defined C type definitions, included in the file mpi.h.	17
/* C opaque types */	18
MPI_Aint	19
MPI_Count	20
MPI_Fint	21
MPI_Offset	22
MPI_Status	23
MPI_F08_status	24
	25
<pre>/* C handles to assorted structures */</pre>	26
MPI_Comm	27
MPI_Datatype	28
MPI_Errhandler	29
MPI_File	30
MPI_Group	31
MPI_Info	32
MPI_Message	33
MPI_Op	34
MPI_Request	35
MPI_Session	36
MPI_Win	37
	38
<pre>/* Types for the MPI_T interface */</pre>	39
MPI_T_enum	40
MPI_T_cvar_handle	41
MPI_T_pvar_handle	42
MPI_T_pvar_session	43
MPI_T_event_instance	44
MPI_T_event_registration	45
MPI_T_source_order	46
MPI_T_cb_safety	47
	48

1 2 3 The following are defined Fortran type definitions, included in the mpi_f08 and mpi 4 modules. 5! Fortran opaque types in the mpi_f08 and mpi modules 6 TYPE(MPI_Status) 7 8 Fortran handles in the mpi_f08 and mpi modules 9 TYPE(MPI_Comm) 10 TYPE(MPI_Datatype) 11 TYPE(MPI_Errhandler) 12TYPE(MPI_File) 13 TYPE(MPI_Group) 14 TYPE(MPI_Info) 15TYPE(MPI_Message) 16 TYPE(MPI_Op) 17TYPE(MPI_Request) 18 TYPE(MPI_Session) 19 TYPE(MPI_Win) 2021A.1.3 Prototype Definitions 22 23C Bindings 24The following are defined C typedefs for user-defined functions, also included in the file 25mpi.h. 2627/* prototypes for user-defined functions */ 28typedef void MPI_User_function(void *invec, void *inoutvec, int *len, 29 MPI_Datatype *datatype); 30 31typedef void MPI_User_function_c(void *invec, void *inoutvec, 32 MPI_Count *len, MPI_Datatype *datatype); 33 34 typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm, int comm_keyval, void *extra_state, void *attribute_val_in, 35void *attribute_val_out, int *flag); 36 37 typedef int MPI_Comm_delete_attr_function(MPI_Comm comm, int comm_keyval, 38 void *attribute_val, void *extra_state); 39 40typedef int MPI_Win_copy_attr_function(MPI_Win oldwin, int win_keyval, 41 void *extra_state, void *attribute_val_in, 42void *attribute_val_out, int *flag); 43 typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval, 44 void *attribute_val, void *extra_state); 4546typedef int MPI_Type_copy_attr_function(MPI_Datatype oldtype, 47 int type_keyval, void *extra_state, void *attribute_val_in, 48 void *attribute_val_out, int *flag);

typedef	<pre>int MPI_Type_delete_attr_function(MPI_Datatype datatype,</pre>	1 2
typedef	<pre>void MPI_Comm_errhandler_function(MPI_Comm *comm, int *error_code, );</pre>	3 4 5
typedef	<pre>void MPI_Win_errhandler_function(MPI_Win *win, int *error_code, );</pre>	6 7
typedef	<pre>void MPI_File_errhandler_function(MPI_File *file, int *error_code, );</pre>	8 9 10
typedef	<pre>void MPI_Session_errhandler_function(MPI_Session *session,</pre>	11 12 13
typedef	<pre>int MPI_Grequest_query_function(void *extra_state,</pre>	13 14 15
typedef	<pre>int MPI_Grequest_free_function(void *extra_state);</pre>	16 17
typedef	<pre>int MPI_Grequest_cancel_function(void *extra_state, int complete);</pre>	18
typedef	<pre>int MPI_Datarep_extent_function(MPI_Datatype datatype,</pre>	19 20 21
typedef	<pre>int MPI_Datarep_conversion_function(void *userbuf, MPI_Datatype datatype, int count, void *filebuf, MPI_Offset position, void *extra_state);</pre>	22 23 24
typedef	<pre>int MPI_Datarep_conversion_function_c(void *userbuf, MPI_Datatype datatype, MPI_Count count, void *filebuf, MPI_Offset position, void *extra_state);</pre>	25 26 27 28 29
typedef	<pre>void MPI_T_event_cb_function( MPI_T_event_instance event_instance, MPI_T_event_registration event_registration, MPI_T_cb_safety cb_safety, void *user_data);</pre>	23 30 31 32 33 34 35 36
typedef	<pre>void MPI_T_event_free_cb_function(     MPI_T_event_registration event_registration,     MPI_T_cb_safety cb_safety,     void *user_data);</pre>	30 37 38 39 40 41
typedef	<pre>void MPI_T_event_dropped_cb_function(int count, MPI_T_event_registration event_registration, MPI_T_cb_safety cb_safety, void *user_data);</pre>	42 43 44 45 46
		47 48

```
1
 Fortran 2008 Bindings with the mpi_f08 Module
\mathbf{2}
 The callback prototypes when using the Fortran mpi_f08 module are shown below:
3
 The user-function argument to MPI_Op_create should be declared according to:
4
 ABSTRACT INTERFACE
5
 SUBROUTINE MPI_User_function(invec, inoutvec, len, datatype)
6
 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
7
 TYPE(C_PTR), VALUE :: invec, inoutvec
8
 INTEGER :: len
9
 TYPE(MPI_Datatype) :: datatype
10
11
 ABSTRACT INTERFACE
12
 SUBROUTINE MPI_User_function_c(invec, inoutvec, len, datatype)
13
 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
14
 TYPE(C_PTR), VALUE :: invec, inoutvec
15
 INTEGER(KIND=MPI_COUNT_KIND) :: len
16
 TYPE(MPI_Datatype) :: datatype
17
 The copy and delete function arguments to MPI_Comm_create_keyval should be de-
18
 clared according to:
19
 ABSTRACT INTERFACE
20
 SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,
21
 attribute_val_in, attribute_val_out, flag, ierror)
22
 TYPE(MPI_Comm) :: oldcomm
23
 INTEGER :: comm_keyval, ierror
24
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
25
 attribute_val_out
26
 LOGICAL :: flag
27
28
 ABSTRACT INTERFACE
29
 SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval,
30
 attribute_val, extra_state, ierror)
31
 TYPE(MPI_Comm) :: comm
32
 INTEGER :: comm_keyval, ierror
33
 INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
34
 The copy and delete function arguments to MPI_Win_create_keyval should be declared
35
 according to:
36
 ABSTRACT INTERFACE
37
 SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state,
38
 attribute_val_in, attribute_val_out, flag, ierror)
39
 TYPE(MPI_Win) :: oldwin
40
 INTEGER :: win_keyval, ierror
41
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
42
 attribute_val_out
43
 LOGICAL :: flag
44
45
 ABSTRACT INTERFACE
46
 SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,
47
 extra_state, ierror)
48
```

```
1
 TYPE(MPI_Win) :: win
 2
 INTEGER :: win_keyval, ierror
 INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
 4
 The copy and delete function arguments to MPI_Type_create_keyval should be declared
 5
according to:
 6
ABSTRACT INTERFACE
 SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state,
 8
 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,
 12
 attribute_val_out
 13
 LOGICAL :: flag
 14
 15
ABSTRACT INTERFACE
 16
 SUBROUTINE MPI_Type_delete_attr_function(datatype, type_keyval,
 17
 attribute_val, extra_state, ierror)
 18
 TYPE(MPI_Datatype) :: datatype
 19
 INTEGER :: type_keyval, ierror
 INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
 20
 21
 The handler-function argument to MPI_Comm_create_errhandler should be declared
 22
like this:
 23
ABSTRACT INTERFACE
 24
 SUBROUTINE MPI_Comm_errhandler_function(comm, error_code)
 25
 TYPE(MPI_Comm) :: comm
 26
 INTEGER :: error_code
 27
 28
 The handler-function argument to MPI_Win_create_errhandler should be declared like
 29
this:
 30
ABSTRACT INTERFACE
 31
 SUBROUTINE MPI_Win_errhandler_function(win, error_code)
 32
 TYPE(MPI_Win) :: win
 33
 INTEGER :: error_code
 34
 The handler-function argument to MPI_File_create_errhandler should be declared like
 35
this:
 36
ABSTRACT INTERFACE
 37
 SUBROUTINE MPI_File_errhandler_function(file, error_code)
 38
 TYPE(MPI_File) :: file
 39
 INTEGER :: error_code
 40
 41
ABSTRACT INTERFACE
 42
 SUBROUTINE MPI_File_errhandler_function(file, error_code)
 43
 TYPE(MPI_File) :: file
 44
 INTEGER :: error_code
 45
 The handler-function argument to MPI_Session_create_errhandler should be declared
 46
like this:
 47
ABSTRACT INTERFACE
 48
```

```
1
 SUBROUTINE MPI_Session_errhandler_function(session, error_code)
2
 TYPE(MPI_Session) :: session
3
 INTEGER :: error_code
4
 ABSTRACT INTERFACE
5
 SUBROUTINE MPI_Session_errhandler_function(session, error_code)
6
 TYPE(MPI Session) :: session
7
 INTEGER :: error_code
8
9
 The query, free, and cancel function arguments to MPI_Grequest_start should be de-
10
 clared according to:
11
 ABSTRACT INTERFACE
12
 SUBROUTINE MPI_Grequest_query_function(extra_state, status, ierror)
13
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
14
 TYPE(MPI_Status) :: status
15
 INTEGER :: ierror
16
 ABSTRACT INTERFACE
17
 SUBROUTINE MPI_Grequest_free_function(extra_state, ierror)
18
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
19
 INTEGER :: ierror
20
21
 ABSTRACT INTERFACE
22
 SUBROUTINE MPI_Grequest_cancel_function(extra_state, complete, ierror)
23
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
24
 LOGICAL :: complete
25
 INTEGER :: ierror
26
 The extent and conversion function arguments to MPI_Register_datarep should be de-
27
 clared according to:
28
 ABSTRACT INTERFACE
29
 SUBROUTINE MPI_Datarep_extent_function(datatype, extent, extra_state,
30
 ierror)
31
 TYPE(MPI_Datatype) :: datatype
32
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extent, extra_state
33
 INTEGER :: ierror
34
35
 ABSTRACT INTERFACE
36
 SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count,
37
 filebuf, position, extra_state, ierror)
38
 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
39
 TYPE(C_PTR), VALUE :: userbuf, filebuf
40
 TYPE(MPI_Datatype) :: datatype
41
 INTEGER :: count, ierror
42
 INTEGER(KIND=MPI_OFFSET_KIND) :: position
43
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
44
 ABSTRACT INTERFACE
45
 SUBROUTINE MPI_Datarep_conversion_function_c(userbuf, datatype, count,
46
 filebuf, position, extra_state, ierror)
47
 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
48
```

```
1
 TYPE(C_PTR), VALUE :: userbuf, filebuf
 2
 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
Fortran Bindings with mpif.h or the mpi Module
 10
With the Fortran mpi module or mpif.h, here are examples of how each of the user-defined
 11
subroutines should be declared.
 12
 The user-function argument to MPI_OP_CREATE should be declared like this:
 13
 SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, DATATYPE)
 14
 <type> INVEC(LEN), INOUTVEC(LEN)
 15
 INTEGER LEN, DATATYPE
 16
 The copy and delete function arguments to MPL_COMM_CREATE_KEYVAL should be
 17
declared like these:
 18
SUBROUTINE COMM_COPY_ATTR_FUNCTION(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
 19
 ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
 20
 INTEGER OLDCOMM, COMM_KEYVAL, IERROR
 21
 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
 22
 ATTRIBUTE_VAL_OUT
 23
 LOGICAL FLAG
 24
 25
SUBROUTINE COMM_DELETE_ATTR_FUNCTION(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
 26
 EXTRA_STATE, IERROR)
 27
 INTEGER COMM, COMM_KEYVAL, IERROR
 28
 INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
 29
 The copy and delete function arguments to MPI_WIN_CREATE_KEYVAL should be
 30
declared like these:
 31
SUBROUTINE WIN_COPY_ATTR_FUNCTION(OLDWIN, WIN_KEYVAL, EXTRA_STATE,
 32
 ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
 33
 INTEGER OLDWIN, WIN_KEYVAL, IERROR
 34
 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
 35
 ATTRIBUTE_VAL_OUT
 36
 LOGICAL FLAG
 37
 38
SUBROUTINE WIN_DELETE_ATTR_FUNCTION(WIN, WIN_KEYVAL, ATTRIBUTE_VAL,
 39
 EXTRA_STATE, IERROR)
 40
 INTEGER WIN, WIN_KEYVAL, IERROR
 41
 INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
 42
 The delete function argument to MPI_SESSION_CREATE_KEYVAL should be declared
 43
like this:
 44
 45
SUBROUTINE SESSION_DELETE_ATTR_FUNCTION(SESSION, SESSION_KEYVAL,
 46
 ATTRIBUTE_VAL, EXTRA_STATE, IERROR)
 47
 INTEGER SESSION, SESSION_KEYVAL, IERROR
 48
```

1	INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
2	
$\frac{3}{4}$	The copy and delete function arguments to MPI_TYPE_CREATE_KEYVAL should be declared like these:
5	SUBROUTINE TYPE_COPY_ATTR_FUNCTION(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE,
6	ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
7	INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
8	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
9	ATTRIBUTE_VAL_OUT
10	LOGICAL FLAG
11	SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,
12	EXTRA_STATE, IERROR)
13	INTEGER DATATYPE, TYPE_KEYVAL, IERROR
14	INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
15	
16 17	The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be de- clared like this:
18	SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)
19	INTEGER COMM, ERROR_CODE
20	
21	The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be de-
22	clared like this:
23	SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE) INTEGER WIN, ERROR_CODE
24	INTEGER WIN, ERROR_CODE
25	The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de-
26	clared like this:
27 28	SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)
28 29	INTEGER FILE, ERROR_CODE
30	The handler-function argument to $MPI_SESSION_CREATE_ERRHANDLER$ should be
31	declared like this:
32	SUBROUTINE SESSION_ERRHANDLER_FUNCTION(SESSION, ERROR_CODE)
33	INTEGER SESSION, ERROR_CODE
34	The query, free, and cancel function arguments to MPI_GREQUEST_START should be
35	declared like these:
36	SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)
37	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
38 20	INTEGER STATUS(MPI_STATUS_SIZE), IERROR
39 40	SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)
40	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
42	INTEGER IERROR
43	SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)
44	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
45	LOGICAL COMPLETE
46	INTEGER IERROR
47	
48	

The extent and conversion function arguments to MPI_REGISTER_DATAREP should	1 2
be declared like these: SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)	3
INTEGER DATATYPE, IERROR	4
INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE	5
SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF, POSITION, EXTRA_STATE, IERROR)	6 7 8
<type> USERBUF(*), FILEBUF(*)</type>	9
INTEGER DATATYPE, COUNT, IERROR	10
INTEGER(KIND=MPI_OFFSET_KIND) POSITION INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	11
INTEGER(RIND-MIT_RDDRESS_RIND) ERITR_STRIE	12
A.1.4 Deprecated Prototype Definitions	13 14
	15
The following are defined C typedefs for deprecated user-defined functions, also included in the file mpi.h.	16 17
	18
<pre>/* prototypes for user-defined functions */</pre>	19
typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval,	20 21
void *extra_state, void *attribute_val_in,	22
<pre>void *attribute_val_out, int *flag);</pre>	23
<pre>typedef int MPI_Delete_function(MPI_Comm comm, int keyval,</pre>	24
<pre>void *attribute_val, void *extra_state);</pre>	25 26
The following are deprecated Fortran user-defined callback subroutine prototypes. The deprecated copy and delete function arguments to MPI_KEYVAL_CREATE should be de-	27
clared like these:	28 29
SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	30
ATTRIBUTE_VAL_OUT, FLAG, IERR)	31
INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, IERR	32
LOGICAL FLAG	33
	34 35
SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR) INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR	36
INIBOLA COM, ADIVAL, ATTAIDOTL_VAL, EXTAL_DIATE, IMAC	37
	38
A.1.5 Info Keys	39
The following info keys are reserved. They are strings.	40 41
"access_style"	42
"accumulate_ops"	43
"accumulate_ordering" "alloc_shared_noncontig"	44
"appnum"	45
"arch"	$46 \\ 47$
"cb_block_size"	48

¹ "cb	_buffer_	_size"
------------------	----------	--------

- $\mathbf{2}$ "cb_nodes"
- 3 "chunked_item"
- 4 "chunked_size"
- $\mathbf{5}$ "chunked"
- 6 "collective_buffering"
- $\overline{7}$ "file"
- 8 "file_perm"
- 9 "filename"
- 10"host"
- 11 "io_node_list"
- 12"ip_address"
- 13"ip_port"
- 14"mpi_assert_allow_overtaking"
- 15"mpi_assert_exact_length"
- 16"mpi_assert_no_any_source"
- 17"mpi_assert_no_any_tag"
- 18"mpi_assert_strict_start_ordering"
- 19"mpi_hw_resource_type"
- 20"mpi_initial_errhandler"
- 21"mpi_optimization_goal"
- 22"mpi_reuse_count"
- 23"mpi_minimum_memory_alignment"
- 24"nb_proc"
- 25"no_locks"
- 26"num_io_nodes"
- 27"path"
- 28 "same_disp_unit"
- 29"same_size"
- 30 "soft"
- 31"striping_factor"
- 32"striping_unit" "wdir"
- 33

3435

36

37

## Info Values A.1.6

- The following info values are reserved. They are strings. 38
- "false" 39
- 40"mpi_errors_abort"
- 41 "mpi_errors_are_fatal"
- 42"mpi_errors_return"
- 43 "mpi_shared_memory"
- "random" 44
- 45"rar"
- "raw" 46
- 47"read_mostly"
- 48"read_once"

"reverse_sequential" "same_op"	1 2
"same_op_no_op"	3
"sequential"	4
"true"	5
"war"	6 7
"waw"	8
"write_mostly" "write_once"	9
write_once	10
A.2 Summary of the Semantics of all Operation-Related MPI Procedures	11 12
A summary of the semantics of all operation-related $MPI$ procedures can be found in [51].	13 14
	15
	16
	17
	18
	19
	20
	21
	22 23
	23 24
	25
	26
	27
	28
	29
	30
	31
	32
	33 34
	35
	36
	37
	38
	39
	40
	41
	42
	43
	44 45
	45 46
	40
	48

1	A.3 C Bindings
2 3	A.3.1 Point-to-Point Communication C Bindings
4 5 6	<pre>int MPI_Bsend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>
7 8	<pre>int MPI_Bsend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,</pre>
9 10 11	<pre>int MPI_Bsend_init(const void *buf, int count, MPI_Datatype datatype,</pre>
12 13 14	<pre>int MPI_Bsend_init_c(const void *buf, MPI_Count count,</pre>
15 16	<pre>int MPI_Buffer_attach(void *buffer, int size)</pre>
17	<pre>int MPI_Buffer_attach_c(void *buffer, MPI_Count size)</pre>
18 19	<pre>int MPI_Buffer_detach(void *buffer_addr, int *size)</pre>
20	<pre>int MPI_Buffer_detach_c(void *buffer_addr, MPI_Count *size)</pre>
21 22	int MPI_Cancel(MPI_Request *request)
23 24 25	<pre>int MPI_Get_count(const MPI_Status *status, MPI_Datatype datatype,</pre>
26 27	<pre>int MPI_Get_count_c(const MPI_Status *status, MPI_Datatype datatype,</pre>
28 29 30	<pre>int MPI_Ibsend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>
31 32	<pre>int MPI_Ibsend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,</pre>
33 34 35	<pre>int MPI_Improbe(int source, int tag, MPI_Comm comm, int *flag, MPI_Message *message, MPI_Status *status)</pre>
36 37	<pre>int MPI_Imrecv(void *buf, int count, MPI_Datatype datatype, MPI_Message *message, MPI_Request *request)</pre>
38 39 40	<pre>int MPI_Imrecv_c(void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Message *message, MPI_Request *request)</pre>
41 42	int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)
43 44 45	<pre>int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source,</pre>
46 47 48	<pre>int MPI_Irecv_c(void *buf, MPI_Count count, MPI_Datatype datatype,</pre>

int	<pre>MPI_Irsend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>	1 2
int	<pre>MPI_Irsend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,</pre>	3 4 5
int	<pre>MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>	6 7 8
int	<pre>MPI_Isend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,</pre>	9 10
int	<pre>MPI_Isendrecv(const void *sendbuf, int sendcount,</pre>	11 12 13 14 15
int	<pre>MPI_Isendrecv_c(const void *sendbuf, MPI_Count sendcount,</pre>	16 17 18 19
int	<pre>MPI_Isendrecv_replace(void *buf, int count, MPI_Datatype datatype,</pre>	20 21 22 23
int	<pre>MPI_Isendrecv_replace_c(void *buf, MPI_Count count,</pre>	24 25 26
int	<pre>MPI_Issend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>	27 28 29
int	<pre>MPI_Issend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,</pre>	30 31
int	<pre>MPI_Mprobe(int source, int tag, MPI_Comm comm, MPI_Message *message, MPI_Status *status)</pre>	32 33 34
int	<pre>MPI_Mrecv(void *buf, int count, MPI_Datatype datatype, MPI_Message *message, MPI_Status *status)</pre>	35 36
int	<pre>MPI_Mrecv_c(void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Message *message, MPI_Status *status)</pre>	37 38 39
int	MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)	40
int	<pre>MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,</pre>	41 42 43
int	<pre>MPI_Recv_c(void *buf, MPI_Count count, MPI_Datatype datatype,</pre>	44 45
int	<pre>MPI_Recv_init(void *buf, int count, MPI_Datatype datatype, int source,</pre>	46 47 48

1int MPI_Recv_init_c(void *buf, MPI_Count count, MPI_Datatype datatype,  $\mathbf{2}$ int source, int tag, MPI_Comm comm, MPI_Request *request) 3 int MPI_Request_free(MPI_Request *request) 4  $\mathbf{5}$ int MPI_Request_get_status(MPI_Request request, int *flag, 6 MPI_Status *status) 7 int MPI_Rsend(const void *buf, int count, MPI_Datatype datatype, int dest, 8 int tag, MPI_Comm comm) 9 10int MPI_Rsend_c(const void *buf, MPI_Count count, MPI_Datatype datatype, 11 int dest, int tag, MPI_Comm comm) 12int MPI_Rsend_init(const void *buf, int count, MPI_Datatype datatype, 13 int dest, int tag, MPI_Comm comm, MPI_Request *request) 1415int MPI_Rsend_init_c(const void *buf, MPI_Count count, 16MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, 17MPI_Request *request) 18 int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, 19int tag, MPI_Comm comm) 2021int MPI_Send_c(const void *buf, MPI_Count count, MPI_Datatype datatype, 22int dest, int tag, MPI_Comm comm) 23int MPI_Send_init(const void *buf, int count, MPI_Datatype datatype,  24 int dest, int tag, MPI_Comm comm, MPI_Request *request) 2526int MPI_Send_init_c(const void *buf, MPI_Count count, 27MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, 28 MPI_Request *request) 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 int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype, 39 40int dest, int sendtag, int source, int recvtag, MPI_Comm comm, 41 MPI_Status *status) 42int MPI_Sendrecv_replace_c(void *buf, MPI_Count count, 43 MPI_Datatype datatype, int dest, int sendtag, int source, 44 int recvtag, MPI_Comm comm, MPI_Status *status) 4546int MPI_Ssend(const void *buf, int count, MPI_Datatype datatype, int dest, 47int tag, MPI_Comm comm) 48

int	<pre>MPI_Ssend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,</pre>	1 2
int	<pre>MPI_Ssend_init(const void *buf, int count, MPI_Datatype datatype,</pre>	3 4 5
int	<pre>MPI_Ssend_init_c(const void *buf, MPI_Count count,</pre>	6 7 8
int	MPI_Start(MPI_Request *request)	9 10
int	<pre>MPI_Startall(int count, MPI_Request array_of_requests[])</pre>	11
int	MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)	12 13
int	MPI_Test_cancelled(const MPI_Status *status, int *flag)	14
int	<pre>MPI_Testall(int count, MPI_Request array_of_requests[], int *flag, MPI_Status array_of_statuses[])</pre>	15 16 17
int	<pre>MPI_Testany(int count, MPI_Request array_of_requests[], int *index,</pre>	18 19 20
int	<pre>MPI_Testsome(int incount, MPI_Request array_of_requests[],</pre>	20 21 22 23
int	MPI_Wait(MPI_Request *request, MPI_Status *status)	24 25
int	<pre>MPI_Waitall(int count, MPI_Request array_of_requests[], MPI_Status array_of_statuses[])</pre>	26 26 27
int	<pre>MPI_Waitany(int count, MPI_Request array_of_requests[], int *index, MPI_Status *status)</pre>	28 29 30
int	<pre>MPI_Waitsome(int incount, MPI_Request array_of_requests[],</pre>	31 32 33 34
A.3.	2 Partitioned Communication C Bindings	35 36
int	MPI_Parrived(MPI_Request *request, int partition, int *flag)	37
int	MPI_Pready(int partition, MPI_Request *request)	38 39
int	<pre>MPI_Pready_list(int length, const int array_of_partitions[], MPI_Request *request)</pre>	40 41
int	<pre>MPI_Pready_range(int partition_low, int partition_high, MPI_Request *request)</pre>	42 43 44
int	<pre>MPI_Precv_init(void *buf, int partitions, MPI_Count count,</pre>	45 46 47 48

1 2 3 4	int	<pre>MPI_Psend_init(const void *buf, int partitions, MPI_Count count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Info info, MPI_Request *request)</pre>
5 6	A.3.	.3 Datatypes C Bindings
7 8	int	<pre>MPI_Get_address(const void *location, MPI_Aint *address)</pre>
9 10	int	<pre>MPI_Get_elements(const MPI_Status *status, MPI_Datatype datatype,</pre>
11 12 13	int	<pre>MPI_Get_elements_c(const MPI_Status *status, MPI_Datatype datatype,</pre>
14 15	int	<pre>MPI_Get_elements_x(const MPI_Status *status, MPI_Datatype datatype,</pre>
16 17 18	int	<pre>MPI_Pack(const void *inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outsize, int *position, MPI_Comm comm)</pre>
19 20 21	int	<pre>MPI_Pack_c(const void *inbuf, MPI_Count incount, MPI_Datatype datatype, void *outbuf, MPI_Count outsize, MPI_Count *position, MPI_Comm comm)</pre>
22 23 24 25	int	<pre>MPI_Pack_external(const char datarep[], const void *inbuf, int incount,</pre>
26 27 28	int	<pre>MPI_Pack_external_c(const char datarep[], const void *inbuf, MPI_Count incount, MPI_Datatype datatype, void *outbuf, MPI_Count outsize, MPI_Count *position)</pre>
29 30 31	int	<pre>MPI_Pack_external_size(const char datarep[], int incount, MPI_Datatype datatype, MPI_Aint *size)</pre>
32 33	int	<pre>MPI_Pack_external_size_c(const char datarep[], MPI_Count incount, MPI_Datatype datatype, MPI_Count *size)</pre>
34 35 36	int	<pre>MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,</pre>
37 38 39	int	<pre>MPI_Pack_size_c(MPI_Count incount, MPI_Datatype datatype,</pre>
40	int	MPI_Type_commit(MPI_Datatype *datatype)
41 42 43	int	MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
43 44 45	int	MPI_Type_contiguous_c(MPI_Count count, MPI_Datatype oldtype, MPI_Datatype *newtype)
46 47 48	int	<pre>MPI_Type_create_darray(int size, int rank, int ndims,</pre>

	<pre>const int array_of_dargs[], const int array_of_psizes[],</pre>	1
	int order, MPI_Datatype oldtype, MPI_Datatype *newtype)	2
int MPI_Type	_create_darray_c(int size, int rank, int ndims,	3
- 51	const MPI_Count array_of_gsizes[],	4
	const int array_of_distribs[], const int array_of_dargs[],	5
	const int array_of_psizes[], int order, MPI_Datatype oldtype,	6 7
	MPI_Datatype *newtype)	8
		9
int MPI_Type	_create_hindexed(int count, const int array_of_blocklengths[],	10
	<pre>const MPI_Aint array_of_displacements[], MPI_Datatype oldtype, MDI_Datatyme transformed</pre>	11
	MPI_Datatype *newtype)	12
int MPI_Type	_create_hindexed_block(int count, int blocklength,	13
	<pre>const MPI_Aint array_of_displacements[], MPI_Datatype oldtype,</pre>	14
	MPI_Datatype *newtype)	15
int MPT Tune	_create_hindexed_block_c(MPI_Count count,	16
int mi_iype	MPI_Count blocklength,	17
	const MPI_Count array_of_displacements[],	18
	MPI_Datatype oldtype, MPI_Datatype *newtype)	19
		20
int MPI_Type	_create_hindexed_c(MPI_Count count,	21
	<pre>const MPI_Count array_of_blocklengths[],</pre>	22
	<pre>const MPI_Count array_of_displacements[],</pre>	23
	MPI_Datatype oldtype, MPI_Datatype *newtype)	24
int MPI_Type	_create_hvector(int count, int blocklength, MPI_Aint stride,	25
	MPI_Datatype oldtype, MPI_Datatype *newtype)	26
int MDT Turno	_create_hvector_c(MPI_Count count, MPI_Count blocklength,	27
inc hri_iype	MPI_Count stride, MPI_Datatype oldtype, MPI_Datatype *newtype)	28
	In i_count stride, in i_batatype ordtype, in i_batatype wnewtype)	29 30
int MPI_Type	_create_indexed_block(int count, int blocklength,	31
	<pre>const int array_of_displacements[], MPI_Datatype oldtype,</pre>	32
	MPI_Datatype *newtype)	33
int MPI_Type	_create_indexed_block_c(MPI_Count count, MPI_Count blocklength,	34
- 51	const MPI_Count array_of_displacements[],	35
	MPI_Datatype oldtype, MPI_Datatype *newtype)	36
· · NDT T		37
int MPI_Type	_create_resized(MPI_Datatype oldtype, MPI_Aint lb,	38
	MPI_Aint extent, MPI_Datatype *newtype)	39
int MPI_Type	_create_resized_c(MPI_Datatype oldtype, MPI_Count lb,	40
	MPI_Count extent, MPI_Datatype *newtype)	41
int MPT Tune	_create_struct(int count, const int array_of_blocklengths[],	42
ing uni_iybe	<pre>const MPI_Aint array_of_displacements[],</pre>	43
	const MPI_Datatype array_of_types[], MPI_Datatype *newtype)	44
		45
int MPI_Type	_create_struct_c(MPI_Count count,	46
	<pre>const MPI_Count array_of_blocklengths[],</pre>	47
	<pre>const MPI_Count array_of_displacements[],</pre>	48

```
1
 const MPI_Datatype array_of_types[], MPI_Datatype *newtype)
\mathbf{2}
 int MPI_Type_create_subarray(int ndims, const int array_of_sizes[],
3
 const int array_of_subsizes[], const int array_of_starts[],
4
 int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
5
6
 int MPI_Type_create_subarray_c(int ndims, const MPI_Count array_of_sizes[],
7
 const MPI_Count array_of_subsizes[],
8
 const MPI_Count array_of_starts[], int order,
9
 MPI_Datatype oldtype, MPI_Datatype *newtype)
10
 int MPI_Type_dup(MPI_Datatype oldtype, MPI_Datatype *newtype)
11
12
 int MPI_Type_free(MPI_Datatype *datatype)
13
 int MPI_Type_get_contents(MPI_Datatype datatype, int max_integers,
14
 int max_addresses, int max_datatypes, int array_of_integers[],
15
 MPI_Aint array_of_addresses[],
16
 MPI_Datatype array_of_datatypes[])
17
18
 int MPI_Type_get_contents_c(MPI_Datatype datatype, MPI_Count max_integers,
19
 MPI_Count max_addresses, MPI_Count max_large_counts,
20
 MPI_Count max_datatypes, int array_of_integers[],
21
 MPI_Aint array_of_addresses[],
22
 MPI_Count array_of_large_counts[],
23
 MPI_Datatype array_of_datatypes[])
24
 int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers,
25
 int *num_addresses, int *num_datatypes, int *combiner)
26
27
 int MPI_Type_get_envelope_c(MPI_Datatype datatype, MPI_Count *num_integers,
28
 MPI_Count *num_addresses, MPI_Count *num_large_counts,
29
 MPI_Count *num_datatypes, int *combiner)
30
 int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb,
^{31}
 MPI_Aint *extent)
32
33
 int MPI_Type_get_extent_c(MPI_Datatype datatype, MPI_Count *lb,
34
 MPI_Count *extent)
35
 int MPI_Type_get_extent_x(MPI_Datatype datatype, MPI_Count *1b,
36
 MPI_Count *extent)
37
38
 int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb,
39
 MPI_Aint *true_extent)
40
41
 int MPI_Type_get_true_extent_c(MPI_Datatype datatype, MPI_Count *true_lb,
42
 MPI_Count *true_extent)
43
 int MPI_Type_get_true_extent_x(MPI_Datatype datatype, MPI_Count *true_lb,
44
 MPI_Count *true_extent)
45
46
 int MPI_Type_indexed(int count, const int array_of_blocklengths[],
47
 const int array_of_displacements[], MPI_Datatype oldtype,
48
 MPI_Datatype *newtype)
```

1 int MPI_Type_indexed_c(MPI_Count count,  $\mathbf{2}$ const MPI_Count array_of_blocklengths[], 3 const MPI_Count array_of_displacements[], MPI_Datatype oldtype, MPI_Datatype *newtype) 4 5 int MPI_Type_size(MPI_Datatype datatype, int *size) 6 7 int MPI_Type_size_c(MPI_Datatype datatype, MPI_Count *size) int MPI_Type_size_x(MPI_Datatype datatype, MPI_Count *size) 9 10 int MPI_Type_vector(int count, int blocklength, int stride, 11 MPI_Datatype oldtype, MPI_Datatype *newtype) 12int MPI_Type_vector_c(MPI_Count count, MPI_Count blocklength, 13 MPI_Count stride, MPI_Datatype oldtype, MPI_Datatype *newtype) 1415int MPI_Unpack(const void *inbuf, int insize, int *position, void *outbuf, 16int outcount, MPI_Datatype datatype, MPI_Comm comm) 17int MPI_Unpack_c(const void *inbuf, MPI_Count insize, MPI_Count *position, 18 void *outbuf, MPI_Count outcount, MPI_Datatype datatype, 19 MPI_Comm comm) 2021int MPI_Unpack_external(const char datarep[], const void *inbuf, 22 MPI_Aint insize, MPI_Aint *position, void *outbuf, 23int outcount, MPI_Datatype datatype) 24int MPI_Unpack_external_c(const char datarep[], const void *inbuf, 25MPI_Count insize, MPI_Count *position, void *outbuf, 26MPI_Count outcount, MPI_Datatype datatype) 2728 29 A.3.4 Collective Communication C Bindings 30 int MPI_Allgather(const void *sendbuf, int sendcount, 31MPI_Datatype sendtype, void *recvbuf, int recvcount, 32 33 MPI_Datatype recvtype, MPI_Comm comm) 34 int MPI_Allgather_c(const void *sendbuf, MPI_Count sendcount, 35 MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount, 36 MPI_Datatype recvtype, MPI_Comm comm) 37 38 int MPI_Allgather_init(const void *sendbuf, int sendcount, 39 MPI_Datatype sendtype, void *recvbuf, int recvcount, 40 MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, 41 MPI_Request *request) 42int MPI_Allgather_init_c(const void *sendbuf, MPI_Count sendcount, 43 MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount, 44 MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, 45MPI_Request *request) 4647int MPI_Allgatherv(const void *sendbuf, int sendcount, 48

1MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],  $\mathbf{2}$ const int displs[], MPI_Datatype recvtype, MPI_Comm comm) 3 int MPI_Allgatherv_c(const void *sendbuf, MPI_Count sendcount, 4 MPI_Datatype sendtype, void *recvbuf, 5const MPI_Count recvcounts[], const MPI_Aint displs[], 6 MPI_Datatype recvtype, MPI_Comm comm) 7 8 int MPI_Allgatherv_init(const void *sendbuf, int sendcount, 9 MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], 10 const int displs[], MPI_Datatype recvtype, MPI_Comm comm, 11 MPI_Info info, MPI_Request *request) 12int MPI_Allgatherv_init_c(const void *sendbuf, MPI_Count sendcount, 13 MPI_Datatype sendtype, void *recvbuf, 14 const MPI_Count recvcounts[], const MPI_Aint displs[], 15MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, 16MPI_Request *request) 17 18int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count, 19MPI_Datatype datatype, MPI_Op op, MPI_Comm comm) 20int MPI_Allreduce_c(const void *sendbuf, void *recvbuf, MPI_Count count, 21MPI_Datatype datatype, MPI_Op op, MPI_Comm comm) 22 23int MPI_Allreduce_init(const void *sendbuf, void *recvbuf, int count, 24MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, 25MPI_Info info, MPI_Request *request) 26int MPI_Allreduce_init_c(const void *sendbuf, void *recvbuf, 27MPI_Count count, MPI_Datatype datatype, MPI_Op op, 28MPI_Comm comm, MPI_Info info, MPI_Request *request) 29 30 int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype,  31 void *recvbuf, int recvcount, MPI_Datatype recvtype, 32 MPI_Comm comm) 33 int MPI_Alltoall_c(const void *sendbuf, MPI_Count sendcount, 34 MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount, 35 MPI_Datatype recvtype, MPI_Comm comm) 36 37 int MPI_Alltoall_init(const void *sendbuf, int sendcount, 38 MPI_Datatype sendtype, void *recvbuf, int recvcount, 39 MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, 40 MPI_Request *request) 41 int MPI_Alltoall_init_c(const void *sendbuf, MPI_Count sendcount, 42MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount, 43 MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, 44 MPI_Request *request) 4546int MPI_Alltoallv(const void *sendbuf, const int sendcounts[], 47const int sdispls[], MPI_Datatype sendtype, void *recvbuf, 48

ANNEX A. LANGUAGE BINDINGS SUMMARY

1 const int recvcounts[], const int rdispls[], 2 MPI_Datatype recvtype, MPI_Comm comm) int MPI_Alltoallv_c(const void *sendbuf, const MPI_Count sendcounts[], 4 const MPI_Aint sdispls[], MPI_Datatype sendtype, 5 void *recvbuf, const MPI_Count recvcounts[], 6 const MPI_Aint rdispls[], MPI_Datatype recvtype, 7 MPI_Comm comm) 8 9 int MPI_Alltoallv_init(const void *sendbuf, const int sendcounts[], 10const int sdispls[], MPI_Datatype sendtype, void *recvbuf, 11 const int recvcounts[], const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, 1213 MPI_Request *request) 14int MPI_Alltoallv_init_c(const void *sendbuf, const MPI_Count sendcounts[], 15const MPI_Aint sdispls[], MPI_Datatype sendtype, 16void *recvbuf, const MPI_Count recvcounts[], 17const MPI_Aint rdispls[], MPI_Datatype recvtype, 18 MPI_Comm comm, MPI_Info info, MPI_Request *request) 19 20int MPI_Alltoallw(const void *sendbuf, const int sendcounts[], 21const int sdispls[], const MPI_Datatype sendtypes[], 22 void *recvbuf, const int recvcounts[], const int rdispls[], 23const MPI_Datatype recvtypes[], MPI_Comm comm) 24int MPI_Alltoallw_c(const void *sendbuf, const MPI_Count sendcounts[], 25const MPI_Aint sdispls[], const MPI_Datatype sendtypes[], 26void *recvbuf, const MPI_Count recvcounts[], 27const MPI_Aint rdispls[], const MPI_Datatype recvtypes[], 28 MPI_Comm comm) 29 30 int MPI_Alltoallw_init(const void *sendbuf, const int sendcounts[], 31const int sdispls[], const MPI_Datatype sendtypes[], 32 void *recvbuf, const int recvcounts[], const int rdispls[], 33 const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info, 34 MPI_Request *request) 35 int MPI_Alltoallw_init_c(const void *sendbuf, const MPI_Count sendcounts[], 36 const MPI_Aint sdispls[], const MPI_Datatype sendtypes[], 37 void *recvbuf, const MPI_Count recvcounts[], 38 const MPI_Aint rdispls[], const MPI_Datatype recvtypes[], 39 MPI_Comm comm, MPI_Info info, MPI_Request *request) 40 41 int MPI_Barrier(MPI_Comm comm) 42int MPI_Barrier_init(MPI_Comm comm, MPI_Info info, MPI_Request *request) 43 44int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, 45MPI_Comm comm) 46int MPI_Bcast_c(void *buffer, MPI_Count count, MPI_Datatype datatype, 47int root, MPI_Comm comm) 48

```
1
 int MPI_Bcast_init(void *buffer, int count, MPI_Datatype datatype,
\mathbf{2}
 int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)
3
 int MPI_Bcast_init_c(void *buffer, MPI_Count count, MPI_Datatype datatype,
4
 int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)
5
6
 int MPI_Exscan(const void *sendbuf, void *recvbuf, int count,
7
 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
8
 int MPI_Exscan_c(const void *sendbuf, void *recvbuf, MPI_Count_count,
9
 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
10
11
 int MPI_Exscan_init(const void *sendbuf, void *recvbuf, int count,
12
 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
13
 MPI_Info info, MPI_Request *request)
14
 int MPI_Exscan_init_c(const void *sendbuf, void *recvbuf, MPI_Count count,
15
 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
16
 MPI_Info info, MPI_Request *request)
17
18
 int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
19
 void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
20
 MPI_Comm comm)
21
 int MPI_Gather_c(const void *sendbuf, MPI_Count sendcount,
22
 MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
23
 MPI_Datatype recvtype, int root, MPI_Comm comm)
24
25
 int MPI_Gather_init(const void *sendbuf, int sendcount,
26
 MPI_Datatype sendtype, void *recvbuf, int recvcount,
27
 MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,
28
 MPI_Request *request)
29
 int MPI_Gather_init_c(const void *sendbuf, MPI_Count sendcount,
30
 MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
31
 MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,
32
 MPI_Request *request)
33
34
 int MPI_Gatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
35
 void *recvbuf, const int recvcounts[], const int displs[],
36
 MPI_Datatype recvtype, int root, MPI_Comm comm)
37
 int MPI_Gatherv_c(const void *sendbuf, MPI_Count sendcount,
38
 MPI_Datatype sendtype, void *recvbuf,
39
 const MPI_Count recvcounts[], const MPI_Aint displs[],
40
 MPI_Datatype recvtype, int root, MPI_Comm comm)
41
42
 int MPI_Gatherv_init(const void *sendbuf, int sendcount,
43
 MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
44
 const int displs[], MPI_Datatype recvtype, int root,
45
 MPI_Comm comm, MPI_Info info, MPI_Request *request)
46
47
 int MPI_Gatherv_init_c(const void *sendbuf, MPI_Count sendcount,
48
 MPI_Datatype sendtype, void *recvbuf,
```

	<pre>const MPI_Count recvcounts[], const MPI_Aint displs[],</pre>	1
	MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,	2
	MPI_Request *request)	3
int MPI_Ialls	gather(const void *sendbuf, int sendcount,	4
	MPI_Datatype sendtype, void *recvbuf, int recvcount,	5 6
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)	7
int MDT Toll	rother closest usid transbuf MDI Count condepunt	8
IIIC MPI_IAIIE	<pre>gather_c(const void *sendbuf, MPI_Count sendcount, MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,</pre>	9
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)	10
		11
int MPI_Iallg	gatherv(const void *sendbuf, int sendcount,	12
	<pre>MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],</pre>	13
	<pre>const int displs[], MPI_Datatype recvtype, MPI_Comm comm, MDI_Dermost transmisst)</pre>	14
	MPI_Request *request)	15
int MPI_Iallg	gatherv_c(const void *sendbuf, MPI_Count sendcount,	16
	MPI_Datatype sendtype, void *recvbuf,	17
	<pre>const MPI_Count recvcounts[], const MPI_Aint displs[],</pre>	18 19
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)	20
int MPI_Iallı	ceduce(const void *sendbuf, void *recvbuf, int count,	20
	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,	22
	MPI_Request *request)	23
int MPT Tallı	ceduce_c(const void *sendbuf, void *recvbuf, MPI_Count count,	24
int in i_iaiii	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,	25
	MPI_Request *request)	26
· · NDT T 11.		27
int MPI_lallt	coall(const void *sendbuf, int sendcount,	28
	MPI_Datatype sendtype, void *recvbuf, int recvcount,	29
	<pre>MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)</pre>	30
int MPI_Iallt	coall_c(const void *sendbuf, MPI_Count sendcount,	31 32
	MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,	33
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)	34
int MPI_Iallt	coallv(const void *sendbuf, const int sendcounts[],	35
	<pre>const int sdispls[], MPI_Datatype sendtype, void *recvbuf,</pre>	36
	<pre>const int recvcounts[], const int rdispls[],</pre>	37
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)	38
int MPI Iallt	coallv_c(const void *sendbuf, const MPI_Count sendcounts[],	39
	<pre>const MPI_Aint sdispls[], MPI_Datatype sendtype,</pre>	40
	void *recvbuf, const MPI_Count recvcounts[],	41
	<pre>const MPI_Aint rdispls[], MPI_Datatype recvtype,</pre>	42
	MPI_Comm comm, MPI_Request *request)	43 44
int MPT Tall:	coallw(const void *sendbuf, const int sendcounts[],	44 45
THO IN T_TATTO	<pre>const int sdispls[], const MPI_Datatype sendtypes[],</pre>	40
	<pre>void *recvbuf, const int recvcounts[], const int rdispls[],</pre>	47
	const MPI_Datatype recvtypes[], MPI_Comm comm,	48

```
1
 MPI_Request *request)
\mathbf{2}
 int MPI_Ialltoallw_c(const void *sendbuf, const MPI_Count sendcounts[],
3
 const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],
4
 void *recvbuf, const MPI_Count recvcounts[],
5
 const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],
6
 MPI_Comm comm, MPI_Request *request)
7
8
 int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)
9
 int MPI_Ibcast(void *buffer, int count, MPI_Datatype datatype, int root,
10
 MPI_Comm comm, MPI_Request *request)
11
12
 int MPI_Ibcast_c(void *buffer, MPI_Count count, MPI_Datatype datatype,
13
 int root, MPI_Comm comm, MPI_Request *request)
14
 int MPI_Iexscan(const void *sendbuf, void *recvbuf, int count,
15
 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
16
 MPI_Request *request)
17
18
 int MPI_Iexscan_c(const void *sendbuf, void *recvbuf, MPI_Count count,
19
 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
20
 MPI_Request *request)
21
 int MPI_Igather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
22
 void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
23
 MPI_Comm comm, MPI_Request *request)
24
25
 int MPI_Igather_c(const void *sendbuf, MPI_Count sendcount,
26
 MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
27
 MPI_Datatype recvtype, int root, MPI_Comm comm,
28
 MPI_Request *request)
29
 int MPI_Igatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
30
 void *recvbuf, const int recvcounts[], const int displs[],
31
 MPI_Datatype recvtype, int root, MPI_Comm comm,
32
 MPI_Request *request)
33
34
 int MPI_Igatherv_c(const void *sendbuf, MPI_Count sendcount,
35
 MPI_Datatype sendtype, void *recvbuf,
36
 const MPI_Count recvcounts[], const MPI_Aint displs[],
37
 MPI_Datatype recvtype, int root, MPI_Comm comm,
38
 MPI_Request *request)
39
 int MPI_Ireduce(const void *sendbuf, void *recvbuf, int count,
40
 MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
41
42
 MPI_Request *request)
43
 int MPI_Ireduce_c(const void *sendbuf, void *recvbuf, MPI_Count count,
44
 MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
45
 MPI_Request *request)
46
47
 int MPI_Ireduce_scatter(const void *sendbuf, void *recvbuf,
48
 const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
```

	MPI_Comm comm, MPI_Request *request)	1
int MPI_Ired	uce_scatter_block(const void *sendbuf, void *recvbuf,	2 3
	int recvcount, MPI_Datatype datatype, MPI_Op op,	4
	MPI_Comm comm, MPI_Request *request)	5
int MPI_Ired	uce_scatter_block_c(const void *sendbuf, void *recvbuf,	6
	MPI_Count recvcount, MPI_Datatype datatype, MPI_Op op,	7 8
	MPI_Comm comm, MPI_Request *request)	9
int MPI_Ired	uce_scatter_c(const void *sendbuf, void *recvbuf,	10
	<pre>const MPI_Count recvcounts[], MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Request *request)</pre>	11
		12 13
int MPI_Isca	<pre>n(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,</pre>	14
	MPI_Request *request)	15
int MDT Taco	n_c(const void *sendbuf, void *recvbuf, MPI_Count count,	16
IIIC MFI_ISCA	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,	17 18
	MPI_Request *request)	19
int MPI Isca	tter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,	20
	void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,	21
	MPI_Comm comm, MPI_Request *request)	22 23
int MPI_Isca	tter_c(const void *sendbuf, MPI_Count sendcount,	23 24
	MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,	25
	MPI_Datatype recvtype, int root, MPI_Comm comm,	26
	MPI_Request *request)	27
int MPI_Isca	tterv(const void *sendbuf, const int sendcounts[],	28 29
	const int displs[], MPI_Datatype sendtype, void *recvbuf,	30
	<pre>int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)</pre>	31
int NDT Take		32
int MPI_Isca	<pre>tterv_c(const void *sendbuf, const MPI_Count sendcounts[], const MPI_Aint displs[], MPI_Datatype sendtype, void *recvbuf,</pre>	33 34
	MPI_Count recvcount, MPI_Datatype recvtype, int root,	35
	MPI_Comm comm, MPI_Request *request)	36
int MPI_Op_c	commutative(MPI_Op op, int *commute)	37
int MPI_Op_c	reate(MPI_User_function *user_fn, int commute, MPI_Op *op)	38 39
int MPI_Op_c	reate_c(MPI_User_function_c *user_fn, int commute, MPI_Op *op)	40
int MPI_Op_f	ree(MPI_Op *op)	41 42
int MPI Redu	ce(const void *sendbuf, void *recvbuf, int count,	43
	MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)	44 45
int MPI_Redu	ce_c(const void *sendbuf, void *recvbuf, MPI_Count count,	46
	MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)	47
		48

1 2 3	int	<pre>MPI_Reduce_init(const void *sendbuf, void *recvbuf, int count,</pre>
4 5 6 7	int	<pre>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)</pre>
8 9 10	int	<pre>MPI_Reduce_local(const void *inbuf, void *inoutbuf, int count,</pre>
11 12	int	<pre>MPI_Reduce_local_c(const void *inbuf, void *inoutbuf, MPI_Count count, MPI_Datatype datatype, MPI_Op op)</pre>
13 14 15 16	int	<pre>MPI_Reduce_scatter(const void *sendbuf, void *recvbuf,</pre>
17 18 19	int	<pre>MPI_Reduce_scatter_block(const void *sendbuf, void *recvbuf,</pre>
20 21 22 23	int	<pre>MPI_Reduce_scatter_block_c(const void *sendbuf, void *recvbuf,</pre>
24 25 26	int	<pre>MPI_Reduce_scatter_block_init(const void *sendbuf, void *recvbuf,</pre>
27 28 29 30	int	<pre>MPI_Reduce_scatter_block_init_c(const void *sendbuf, void *recvbuf,</pre>
31 32 33	int	<pre>MPI_Reduce_scatter_c(const void *sendbuf, void *recvbuf,</pre>
34 35 36 37	int	<pre>MPI_Reduce_scatter_init(const void *sendbuf, void *recvbuf,</pre>
38 39 40	int	<pre>MPI_Reduce_scatter_init_c(const void *sendbuf, void *recvbuf,</pre>
41 42 43	int	<pre>MPI_Scan(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)</pre>
44 45	int	<pre>MPI_Scan_c(const void *sendbuf, void *recvbuf, MPI_Count count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)</pre>
46 47 48	int	<pre>MPI_Scan_init(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,</pre>

MPI_Info info, MPI_Request *request)	1
<pre>int MPI_Scan_init_c(const void *sendbuf, void *recvbuf, MPI_Count count,</pre>	2
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,	3
MPI_Info info, MPI_Request *request)	4 5
	6
<pre>int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,</pre>	7
<pre>void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MDI_Comm_comm)</pre>	8
MPI_Comm comm)	9
<pre>int MPI_Scatter_c(const void *sendbuf, MPI_Count sendcount,</pre>	10
<pre>MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,</pre>	11
MPI_Datatype recvtype, int root, MPI_Comm comm)	12
<pre>int MPI_Scatter_init(const void *sendbuf, int sendcount,</pre>	13
MPI_Datatype sendtype, void *recvbuf, int recvcount,	14
MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,	15
MPI_Request *request)	16
	17
<pre>int MPI_Scatter_init_c(const void *sendbuf, MPI_Count sendcount, MDI_Datatement and the send of MDI_Count as a send MDI_Datatement and the send of MDI_Count as a send of the send of</pre>	18
MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,	19
<pre>MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)</pre>	20
MrI_Mequest *Iequest)	21
<pre>int MPI_Scatterv(const void *sendbuf, const int sendcounts[],</pre>	22
<pre>const int displs[], MPI_Datatype sendtype, void *recvbuf,</pre>	23
<pre>int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)</pre>	24 25
<pre>int MPI_Scatterv_c(const void *sendbuf, const MPI_Count sendcounts[],</pre>	25 26
const MPI_Aint displs[], MPI_Datatype sendtype, void *recvbuf,	20
MPI_Count recvcount, MPI_Datatype recvtype, int root,	28
MPI_Comm comm)	29
	30
<pre>int MPI_Scatterv_init(const void *sendbuf, const int sendcounts[],</pre>	31
<pre>const int displs[], MPI_Datatype sendtype, void *recvbuf, int recvourt MPI Datatype recuting int rest MPI Comm comm</pre>	32
<pre>int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)</pre>	33
WI_INO INO, WI_Nequest *request)	34
<pre>int MPI_Scatterv_init_c(const void *sendbuf, const MPI_Count sendcounts[],</pre>	35
<pre>const MPI_Aint displs[], MPI_Datatype sendtype, void *recvbuf,</pre>	36
MPI_Count recvcount, MPI_Datatype recvtype, int root,	37
MPI_Comm comm, MPI_Info info, MPI_Request *request)	38
	39
A.3.5 Groups, Contexts, Communicators, and Caching C Bindings	40
	41
int MPI_COMM_DUP_FN(MPI_Comm oldcomm, int comm_keyval, void *extra_state,	42
<pre>void *attribute_val_in, void *attribute_val_out, int *flag)</pre>	43
int MPI_COMM_NULL_COPY_FN(MPI_Comm oldcomm, int comm_keyval,	44
void *extra_state, void *attribute_val_in,	45
void *attribute_val_out, int *flag)	46
	47 48
	40

1 2	int	<pre>MPI_COMM_NULL_DELETE_FN(MPI_Comm comm, int comm_keyval,</pre>
3 4	int	<pre>MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)</pre>
5	int	MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
6 7 8	int	MPI_Comm_create_from_group(MPI_Group group, const char *stringtag, MPI_Info info, MPI_Errhandler errhandler, MPI_Comm *newcomm)
9 10 11	int	<pre>MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag,</pre>
12 13 14	int	<pre>MPI_Comm_create_keyval(MPI_Comm_copy_attr_function *comm_copy_attr_fn,</pre>
15 16	int	MPI_Comm_delete_attr(MPI_Comm comm, int comm_keyval)
17	int	MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
18 19	int	MPI_Comm_dup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm)
20	int	MPI_Comm_free(MPI_Comm *comm)
21 22	int	MPI_Comm_free_keyval(int *comm_keyval)
23 24	int	<pre>MPI_Comm_get_attr(MPI_Comm comm, int comm_keyval, void *attribute_val,</pre>
25 26	int	<pre>MPI_Comm_get_info(MPI_Comm comm, MPI_Info *info_used)</pre>
27	int	MPI_Comm_get_name(MPI_Comm comm, char *comm_name, int *resultlen)
28 29	int	MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
30	int	MPI_Comm_idup(MPI_Comm comm, MPI_Comm *newcomm, MPI_Request *request)
31 32 33	int	<pre>MPI_Comm_idup_with_info(MPI_Comm comm, MPI_Info info,</pre>
34	int	MPI_Comm_rank(MPI_Comm comm, int *rank)
35 36	int	MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
37	int	MPI_Comm_remote_size(MPI_Comm comm, int *size)
38 39	int	<pre>MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void *attribute_val)</pre>
40 41	int	MPI_Comm_set_info(MPI_Comm comm, MPI_Info info)
42	int	<pre>MPI_Comm_set_name(MPI_Comm comm, const char *comm_name)</pre>
43 44	int	<pre>MPI_Comm_size(MPI_Comm comm, int *size)</pre>
45	int	<pre>MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)</pre>
46 47 48	int	<pre>MPI_Comm_split_type(MPI_Comm comm, int split_type, int key, MPI_Info info, MPI_Comm *newcomm)</pre>

<pre>int MPI_Comm_test_inter(MPI_Comm comm, int *flag)</pre>	
	1
int MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result)	2 3
int MPI_Group_difference(MPI_Group group1, MPI_Group group2,	4
MPI_Group *newgroup)	5
<pre>int MPI_Group_excl(MPI_Group group, int n, const int ranks[],</pre>	6
MPI_Group *newgroup)	7 8
<pre>int MPI_Group_free(MPI_Group *group)</pre>	9
<pre>int MPI_Group_from_session_pset(MPI_Session session, const char *pset_name,</pre>	10 11
MPI_Group *newgroup)	11
<pre>int MPI_Group_incl(MPI_Group group, int n, const int ranks[],</pre>	13
MPI_Group *newgroup)	14 15
<pre>int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,</pre>	16
MPI_Group *newgroup)	17
<pre>int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],</pre>	18 19
MPI_Group *newgroup)	20
<pre>int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],</pre>	21
MPI_Group *newgroup)	22
<pre>int MPI_Group_rank(MPI_Group group, int *rank)</pre>	23 24
<pre>int MPI_Group_size(MPI_Group group, int *size)</pre>	25
<pre>int MPI_Group_translate_ranks(MPI_Group group1, int n, const int ranks1[],</pre>	26
MPI_Group group2, int ranks2[])	27 28
<pre>int MPI_Group_union(MPI_Group group1, MPI_Group group2,</pre>	29
MPI_Group *newgroup)	30
<pre>int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,</pre>	31 32
MPI_Comm peer_comm, int remote_leader, int tag,	33
MPI_Comm *newintercomm)	34
<pre>int MPI_Intercomm_create_from_groups(MPI_Group local_group,</pre>	35
<pre>int local_leader, MPI_Group remote_group, int remote_leader, const char *stringtag, MPI_Info info,</pre>	36 37
MPI_Errhandler errhandler, MPI_Comm *newintercomm)	38
int MPI_Intercomm_merge(MPI_Comm intercomm, int high,	39
	40
MPI_Comm *newintracomm)	41
MPI_Comm *newintracomm)	$\frac{41}{42}$
MPI_Comm *newintracomm) int MPI_TYPE_DUP_FN(MPI_Datatype oldtype, int type_keyval,	42 43 44
MPI_Comm *newintracomm) int MPI_TYPE_DUP_FN(MPI_Datatype oldtype, int type_keyval, void *extra_state, void *attribute_val_in,	42 43 44 45
MPI_Comm *newintracomm) int MPI_TYPE_DUP_FN(MPI_Datatype oldtype, int type_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag)	42 43 44

1	nt MPI_TYPE_NULL_DELETE_FN(MPI_Datatype datatype, int type_keyval, void *attribute_val, void *extra_state)	
3 4 5 6	<pre>nt MPI_Type_create_keyval(MPI_Type_copy_attr_function *type_copy_attr_fn,</pre>	
7	nt MPI_Type_delete_attr(MPI_Datatype datatype, int type_keyval)	
8 9	nt MPI_Type_free_keyval(int *type_keyval)	
10 11	nt MPI_Type_get_attr(MPI_Datatype datatype, int type_keyval, void *attribute_val, int *flag)	
12 13 14	<pre>nt MPI_Type_get_name(MPI_Datatype datatype, char *type_name,</pre>	
15 16	nt MPI_Type_set_attr(MPI_Datatype datatype, int type_keyval, void *attribute_val)	
17 18	nt MPI_Type_set_name(MPI_Datatype datatype, const char *type_name)	
19 20 21	nt MPI_WIN_DUP_FN(MPI_Win oldwin, int win_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag)	
22 23	nt MPI_WIN_NULL_COPY_FN(MPI_Win oldwin, int win_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag)	,
24 25 26	nt MPI_WIN_NULL_DELETE_FN(MPI_Win win, int win_keyval, void *attribute_val, void *extra_state)	
27 28 29	nt 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)	
30 31	nt MPI_Win_delete_attr(MPI_Win win, int win_keyval)	
32	nt MPI_Win_free_keyval(int *win_keyval)	
33 34 35	nt MPI_Win_get_attr(MPI_Win win, int win_keyval, void *attribute_val, int *flag)	
36	nt MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)	
37 38	nt MPI_Win_set_attr(MPI_Win win, int win_keyval, void *attribute_val)	
39 40	nt MPI_Win_set_name(MPI_Win win, const char *win_name)	
41 42	A.3.6 Process Topologies C Bindings	
43	nt MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])	
44 45	<pre>nt MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[],</pre>	
46	<pre>const int periods[], int reorder, MPI_Comm *comm_cart)</pre>	
47 48		

int	<pre>MPI_Cart_get(MPI_Comm comm, int maxdims, int dims[], int periods[],</pre>	1 2
int	<pre>MPI_Cart_map(MPI_Comm comm, int ndims, const int dims[],</pre>	3 4 5
int	<pre>MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank)</pre>	6
int	<pre>MPI_Cart_shift(MPI_Comm comm, int direction, int disp,</pre>	7 8 9
int	MPI_Cart_sub(MPI_Comm comm, const int remain_dims[], MPI_Comm *newcomm)	10
int	MPI_Cartdim_get(MPI_Comm comm, int *ndims)	11 12
int	MPI_Dims_create(int nnodes, int ndims, int dims[])	13
int	<pre>MPI_Dist_graph_create(MPI_Comm comm_old, int n, const int sources[],</pre>	14 15 16 17 18
int	<pre>MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree,</pre>	19 20 21 22 23
int	<pre>MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[],</pre>	24 25 26
int	<pre>MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree,</pre>	27 28 29
int	<pre>MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int index[],</pre>	30 31
int	<pre>MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int index[],</pre>	32 33 34
int	<pre>MPI_Graph_map(MPI_Comm comm, int nnodes, const int index[], const int edges[], int *newrank)</pre>	35 36
int	<pre>MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,</pre>	37 38 39
int	MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)	40
int	MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)	41 42
int	<pre>MPI_Ineighbor_allgather(const void *sendbuf, int sendcount,</pre>	43 44 45
int	<pre>MPI_Ineighbor_allgather_c(const void *sendbuf, MPI_Count sendcount, MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,</pre>	46 47 48

1MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)  $\mathbf{2}$ int MPI_Ineighbor_allgatherv(const void *sendbuf, int sendcount, 3 MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], 4 const int displs[], MPI_Datatype recvtype, MPI_Comm comm, 5MPI_Request *request) 6  $\overline{7}$ int MPI_Ineighbor_allgatherv_c(const void *sendbuf, MPI_Count sendcount, 8 MPI_Datatype sendtype, void *recvbuf, 9 const MPI_Count recvcounts[], const MPI_Aint displs[], 10 MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) 11 int MPI_Ineighbor_alltoall(const void *sendbuf, int sendcount, 12MPI_Datatype sendtype, void *recvbuf, int recvcount, 13 MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) 1415int MPI_Ineighbor_alltoall_c(const void *sendbuf, MPI_Count sendcount, 16MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount, 17MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) 18 int MPI_Ineighbor_alltoallv(const void *sendbuf, const int sendcounts[], 19 const int sdispls[], MPI_Datatype sendtype, void *recvbuf, 20const int recvcounts[], const int rdispls[], 21MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) 22 23int MPI_Ineighbor_alltoallv_c(const void *sendbuf, 24const MPI_Count sendcounts[], const MPI_Aint sdispls[], 25MPI_Datatype sendtype, void *recvbuf, 26const MPI_Count recvcounts[], const MPI_Aint rdispls[], 27MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) 28int MPI_Ineighbor_alltoallw(const void *sendbuf, const int sendcounts[], 29 const MPI_Aint sdispls[], const MPI_Datatype sendtypes[], 30 void *recvbuf, const int recvcounts[], 31const MPI_Aint rdispls[], const MPI_Datatype recvtypes[], 32 MPI_Comm comm, MPI_Request *request) 33 34int MPI_Ineighbor_alltoallw_c(const void *sendbuf, 35 const MPI_Count sendcounts[], const MPI_Aint sdispls[], 36 const MPI_Datatype sendtypes[], void *recvbuf, 37 const MPI_Count recvcounts[], const MPI_Aint rdispls[], 38 const MPI_Datatype recvtypes[], MPI_Comm comm, 39 MPI_Request *request) 40int MPI_Neighbor_allgather(const void *sendbuf, int sendcount, 41 MPI_Datatype sendtype, void *recvbuf, int recvcount, 42MPI_Datatype recvtype, MPI_Comm comm) 43 44int MPI_Neighbor_allgather_c(const void *sendbuf, MPI_Count sendcount, 45 MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount, 46MPI_Datatype recvtype, MPI_Comm comm) 47int MPI_Neighbor_allgather_init(const void *sendbuf, int sendcount, 48

	MPI_Datatype sendtype, void *recvbuf, int recvcount,	1
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,	2
	MPI_Request *request)	3
		4
int MPI_Neigl	hbor_allgather_init_c(const void *sendbuf, MPI_Count sendcount,	5
	<pre>MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,</pre>	6
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,	7
	MPI_Request *request)	8
int MPT Neig	hbor_allgatherv(const void *sendbuf, int sendcount,	9
THE HEITWEIGH	MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],	10
	const int displs[], MPI_Datatype recvtype, MPI_Comm comm)	11
	const int dispis[], Mri_batatype recytype, Mri_comm comm)	12
int MPI_Neigl	hbor_allgatherv_c(const void *sendbuf, MPI_Count sendcount,	13
	MPI_Datatype sendtype, void *recvbuf,	14
	<pre>const MPI_Count recvcounts[], const MPI_Aint displs[],</pre>	15
	MPI_Datatype recvtype, MPI_Comm comm)	16
		17
int MPI_Neigl	hbor_allgatherv_init(const void *sendbuf, int sendcount,	18
	<pre>MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],</pre>	19
	<pre>const int displs[], MPI_Datatype recvtype, MPI_Comm comm,</pre>	20
	MPI_Info info, MPI_Request *request)	
int MPT Neig	hbor_allgatherv_init_c(const void *sendbuf,	21
110 11 1_0016	MPI_Count sendcount, MPI_Datatype sendtype, void *recvbuf,	22
	const MPI_Count recvcounts[], const MPI_Aint displs[],	23
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,	24
		25
	MPI_Request *request)	26
int MPI_Neigl	hbor_alltoall(const void *sendbuf, int sendcount,	27
	MPI_Datatype sendtype, void *recvbuf, int recvcount,	28
	MPI_Datatype recvtype, MPI_Comm comm)	29
		30
int MPI_Neigl	hbor_alltoall_c(const void *sendbuf, MPI_Count sendcount,	31
	MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,	32
	MPI_Datatype recvtype, MPI_Comm comm)	33
int MPT Neig	hbor_alltoall_init(const void *sendbuf, int sendcount,	34
1110 111 1_1101.81	MPI_Datatype sendtype, void *recvbuf, int recvcount,	35
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,	36
	MPI_Request *request)	37
	MII_nequest *request)	38
int MPI_Neigl	hbor_alltoall_init_c(const void *sendbuf, MPI_Count sendcount,	39
	MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,	40
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,	41
	MPI_Request *request)	42
		43
int MPI_Neigi	hbor_alltoallv(const void *sendbuf, const int sendcounts[],	44
	<pre>const int sdispls[], MPI_Datatype sendtype, void *recvbuf,</pre>	45
	<pre>const int recvcounts[], const int rdispls[],</pre>	46
	MPI_Datatype recvtype, MPI_Comm comm)	47
int MPI Neig	hbor_alltoallv_c(const void *sendbuf,	48
	,	

```
1
 const MPI_Count sendcounts[], const MPI_Aint sdispls[],
2
 MPI_Datatype sendtype, void *recvbuf,
3
 const MPI_Count recvcounts[], const MPI_Aint rdispls[],
4
 MPI_Datatype recvtype, MPI_Comm comm)
5
 int MPI_Neighbor_alltoallv_init(const void *sendbuf,
6
 const int sendcounts[], const int sdispls[],
7
 MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
8
 const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm,
9
 MPI_Info info, MPI_Request *request)
10
11
 int MPI_Neighbor_alltoallv_init_c(const void *sendbuf,
12
 const MPI_Count sendcounts[], const MPI_Aint sdispls[],
13
 MPI_Datatype sendtype, void *recvbuf,
14
 const MPI_Count recvcounts[], const MPI_Aint rdispls[],
15
 MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
16
 MPI_Request *request)
17
 int MPI_Neighbor_alltoallw(const void *sendbuf, const int sendcounts[],
18
 const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],
19
 void *recvbuf, const int recvcounts[],
20
 const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],
21
 MPI_Comm comm)
22
23
 int MPI_Neighbor_alltoallw_c(const void *sendbuf,
24
 const MPI_Count sendcounts[], const MPI_Aint sdispls[],
25
 const MPI_Datatype sendtypes[], void *recvbuf,
26
 const MPI_Count recvcounts[], const MPI_Aint rdispls[],
27
 const MPI_Datatype recvtypes[], MPI_Comm comm)
28
 int MPI_Neighbor_alltoallw_init(const void *sendbuf,
29
 const int sendcounts[], const MPI_Aint sdispls[],
30
 const MPI_Datatype sendtypes[], void *recvbuf,
31
 const int recvcounts[], const MPI_Aint rdispls[],
32
 const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info,
33
 MPI_Request *request)
34
35
 int MPI_Neighbor_alltoallw_init_c(const void *sendbuf,
36
 const MPI_Count sendcounts[], const MPI_Aint sdispls[],
37
 const MPI_Datatype sendtypes[], void *recvbuf,
38
 const MPI_Count recvcounts[], const MPI_Aint rdispls[],
39
 const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info,
40
 MPI_Request *request)
41
 int MPI_Topo_test(MPI_Comm comm, int *status)
42
43
44
 A.3.7 MPI Environmental Management C Bindings
45
46
 int MPI_Add_error_class(int *errorclass)
47
 int MPI_Add_error_code(int errorclass, int *errorcode)
48
```

ANNEX A. LANGUAGE BINDINGS SUMMARY

896

```
1
int MPI_Add_error_string(int errorcode, const char *string)
 \mathbf{2}
int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)
 3
 \mathbf{4}
int MPI_Comm_call_errhandler(MPI_Comm comm, int errorcode)
 5
int MPI_Comm_create_errhandler(
 6
 MPI_Comm_errhandler_function *comm_errhandler_fn,
 7
 MPI_Errhandler *errhandler)
 9
int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)
 10
int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)
 11
 12
int MPI_Errhandler_free(MPI_Errhandler *errhandler)
 13
int MPI_Error_class(int errorcode, int *errorclass)
 14
 15
int MPI_Error_string(int errorcode, char *string, int *resultlen)
 16
int MPI_File_call_errhandler(MPI_File fh, int errorcode)
 17
 18
int MPI_File_create_errhandler(
 19
 MPI_File_errhandler_function *file_errhandler_fn,
 20
 MPI_Errhandler *errhandler)
 21
int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)
 22
 23
int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)
 ^{24}
int MPI_Free_mem(void *base)
 25
 26
int MPI_Get_library_version(char *version, int *resultlen)
 27
int MPI_Get_processor_name(char *name, int *resultlen)
 28
 29
int MPI_Get_version(int *version, int *subversion)
 30
 31
int MPI_Session_call_errhandler(MPI_Session session, int errorcode)
 32
int MPI_Session_create_errhandler(
 33
 MPI_Session_errhandler_function *session_errhandler_fn,
 34
 MPI_Errhandler *errhandler)
 35
 36
int MPI_Session_get_errhandler(MPI_Session session,
 37
 MPI_Errhandler *errhandler)
 38
int MPI_Session_set_errhandler(MPI_Session session,
 39
 MPI_Errhandler errhandler)
 40
 41
int MPI_Win_call_errhandler(MPI_Win win, int errorcode)
 42
int MPI_Win_create_errhandler(
 43
 MPI_Win_errhandler_function *win_errhandler_fn,
 44
 MPI_Errhandler *errhandler)
 45
 46
int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)
 47
int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)
 48
```

1	A.3.8 The Info Object C Bindings
2 3	<pre>int MPI_Info_create(MPI_Info *info)</pre>
4	<pre>int MPI_Info_create_env(int argc, char argv[], MPI_Info *info)</pre>
5 6	<pre>int MPI_Info_delete(MPI_Info info, const char *key)</pre>
7	int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo)
8 9	<pre>int MPI_Info_free(MPI_Info *info)</pre>
10 11	<pre>int MPI_Info_get(MPI_Info info, const char *key, int valuelen, char *value,</pre>
12 13	<pre>int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)</pre>
14 15	int MPI_Info_get_nthkey(MPI_Info info, int n, char *key)
16 17	<pre>int MPI_Info_get_string(MPI_Info info, const char *key, int *buflen,</pre>
18 19 20	<pre>int MPI_Info_get_valuelen(MPI_Info info, const char *key, int *valuelen,</pre>
21 22	<pre>int MPI_Info_set(MPI_Info info, const char *key, const char *value)</pre>
23 24	A.3.9 Process Creation and Management C Bindings
25	int MPI_Abort(MPI_Comm comm, int errorcode)
26 27	<pre>int MPI_Close_port(const char *port_name)</pre>
28 29	<pre>int MPI_Comm_accept(const char *port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *newcomm)</pre>
30 31 32	<pre>int MPI_Comm_connect(const char *port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *newcomm)</pre>
33	<pre>int MPI_Comm_disconnect(MPI_Comm *comm)</pre>
35	<pre>int MPI_Comm_get_parent(MPI_Comm *parent)</pre>
36 37	int MPI_Comm_join(int fd, MPI_Comm *intercomm)
38	int MPI_Comm_spawn(const char *command, char *argv[], int maxprocs,
39 40	<pre>MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm, int array_of_errcodes[])</pre>
41 42 43 44 45	<pre>int MPI_Comm_spawn_multiple(int count, char *array_of_commands[],</pre>
46	<pre>int MPI_Finalize(void)</pre>
47 48	<pre>int MPI_Finalized(int *flag)</pre>

<pre>int MPI_Init(int *argc, char ***argv)</pre>	1
int MPI_Init_thread(int *argc, char ***argv, int required, int *provided)	2 3
int MPI_Initialized(int *flag)	4
int MPI_Is_thread_main(int *flag)	5
^o	6 7
<pre>int MPI_Lookup_name(const char *service_name, MPI_Info info,</pre>	8
<pre>int MPI_Open_port(MPI_Info info, char *port_name)</pre>	9 10
<pre>int MPI_Publish_name(const char *service_name, MPI_Info info,</pre>	11 12
<pre>int MPI_Query_thread(int *provided)</pre>	13 14
<pre>int MPI_Session_finalize(MPI_Session *session)</pre>	15 16
<pre>int MPI_Session_get_info(MPI_Session session, MPI_Info *info_used)</pre>	17
int MPI_Session_get_nth_pset(MPI_Session session, MPI_Info info, int n,	18
<pre>int *pset_len, char *pset_name)</pre>	19 20
int MPI_Session_get_num_psets(MPI_Session session, MPI_Info info,	21
<pre>int *npset_names)</pre>	22
<pre>int MPI_Session_get_pset_info(MPI_Session session, const char *pset_name,</pre>	23 24
	25
<pre>int MPI_Session_init(MPI_Info info, MPI_Errhandler errhandler, MPI_Session *session)</pre>	26 27
int MPI_Unpublish_name(const char *service_name, MPI_Info info,	28
const char *port_name)	29 30
	31
A.3.10 One-Sided Communications C Bindings	32
int MPI_Accumulate(const void *origin_addr, int origin_count,	33 34
MPI_Datatype origin_datatype, int target_rank,	35
MPI_Aint target_disp, int target_count,	36
MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)	37
<pre>int MPI_Accumulate_c(const void *origin_addr, MPI_Count origin_count, MPI_Datatype origin_datatype, int target_rank,</pre>	38 39
MPI_Aint target_disp, MPI_Count target_count,	40
MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)	41 42
int MPI_Compare_and_swap(const void *origin_addr, const void *compare_addr,	
void *result_addr, MPI_Datatype datatype, int target_rank,	44
MPI_Aint target_disp, MPI_Win win)	45 46
<pre>int MPI_Fetch_and_op(const void *origin_addr, void *result_addr, MDI_Datatume_datatume_int_termst_monk_MDI_Aint_termst_dian</pre>	47
MPI_Datatype datatype, int target_rank, MPI_Aint target_disp,	48

1		MPI_Op op, MPI_Win win)
2	int M	<pre>/PI_Get(void *origin_addr, int origin_count,</pre>
$\frac{3}{4}$		MPI_Datatype origin_datatype, int target_rank,
5		MPI_Aint target_disp, int target_count,
6		MPI_Datatype target_datatype, MPI_Win win)
7	int M	<pre>/PI_Get_accumulate(const void *origin_addr, int origin_count,</pre>
8		MPI_Datatype origin_datatype, void *result_addr,
9		<pre>int result_count, MPI_Datatype result_datatype,</pre>
10 11		int target_rank, MPI_Aint target_disp, int target_count,
12		MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
13	int M	<pre>MPI_Get_accumulate_c(const void *origin_addr, MPI_Count origin_count,</pre>
14		MPI_Datatype origin_datatype, void *result_addr,
15		<pre>MPI_Count result_count, MPI_Datatype result_datatype, int target_rank, MPI_Aint target_disp, MPI_Count target_count,</pre>
16 17		MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
17	ir+ 14	
19	int M	<pre>IPI_Get_c(void *origin_addr, MPI_Count origin_count, MPI_Datatype origin_datatype, int target_rank,</pre>
20		MPI_Aint target_disp, MPI_Count target_count,
21		MPI_Datatype target_datatype, MPI_Win win)
22	int M	<pre>/PI_Put(const void *origin_addr, int origin_count,</pre>
23 24	1110 1	MPI_Datatype origin_datatype, int target_rank,
25		MPI_Aint target_disp, int target_count,
26		MPI_Datatype target_datatype, MPI_Win win)
27	int M	<pre>MPI_Put_c(const void *origin_addr, MPI_Count origin_count,</pre>
28		MPI_Datatype origin_datatype, int target_rank,
29 30		MPI_Aint target_disp, MPI_Count target_count,
31		MPI_Datatype target_datatype, MPI_Win win)
32	int M	<pre>/PI_Raccumulate(const void *origin_addr, int origin_count,</pre>
33		MPI_Datatype origin_datatype, int target_rank,
34		MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
35 36		MPI_Request *request)
37	·	
38	INT M	<pre>MPI_Raccumulate_c(const void *origin_addr, MPI_Count origin_count, MPI_Datatype origin_datatype, int target_rank,</pre>
39		MPI_Aint target_disp, MPI_Count target_count,
40		MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
41 42		MPI_Request *request)
42	int M	<pre>/PI_Rget(void *origin_addr, int origin_count,</pre>
44		MPI_Datatype origin_datatype, int target_rank,
45		MPI_Aint target_disp, int target_count,
46		MPI_Datatype target_datatype, MPI_Win win,
47 48		MPI_Request *request)
40		

int	<pre>MPI_Rget_accumulate(const void *origin_addr, int origin_count,</pre>	1
	MPI_Datatype origin_datatype, void *result_addr,	2
	<pre>int result_count, MPI_Datatype result_datatype,</pre>	3
	<pre>int target_rank, MPI_Aint target_disp, int target_count,</pre>	4
	MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,	5
	MPI_Request *request)	6
·	NDT Dest second and sold having alle NDT Count origin south	7
int	MPI_Rget_accumulate_c(const void *origin_addr, MPI_Count origin_count,	8
	MPI_Datatype origin_datatype, void *result_addr,	9
	MPI_Count result_count, MPI_Datatype result_datatype,	10
	int target_rank, MPI_Aint target_disp, MPI_Count target_count,	11
	MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,	12
	MPI_Request *request)	13
int	MPI_Rget_c(void *origin_addr, MPI_Count origin_count,	14
	MPI_Datatype origin_datatype, int target_rank,	15
	MPI_Aint target_disp, MPI_Count target_count,	16
	MPI_Datatype target_datatype, MPI_Win win,	17
	MPI_Request *request)	18
		19
int	<pre>MPI_Rput(const void *origin_addr, int origin_count,</pre>	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
	MPI_Request *request)	24
int	MPI_Rput_c(const void *origin_addr, MPI_Count origin_count,	25
1110	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
	MPI_Request *request)	29
		30
int	<pre>MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info,</pre>	31
	MPI_Comm comm, void *baseptr, MPI_Win *win)	32
int	MPI_Win_allocate_c(MPI_Aint size, MPI_Aint disp_unit, MPI_Info info,	33
THE	MPI_Comm comm, void *baseptr, MPI_Win *win)	34
	In 1_comm comm, void #basepti, in 1_win #win/	35
int	MPI_Win_allocate_shared(MPI_Aint size, int disp_unit, MPI_Info info,	36
	MPI_Comm comm, void *baseptr, MPI_Win *win)	37
	MDI Win allegate shared a(MDI Aint size MDI Aint dian unit	38
TUC	MPI_Win_allocate_shared_c(MPI_Aint size, MPI_Aint disp_unit,	39
	MPI_Info info, MPI_Comm comm, void *baseptr, MPI_Win *win)	40
int	MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size)	41
		42
int	MPI_Win_complete(MPI_Win win)	43
int	MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info,	44
	MPI_Comm comm, MPI_Win *win)	45
• .		46
int	MPI_Win_create_c(void *base, MPI_Aint size, MPI_Aint disp_unit,	47
	MPI_Info info, MPI_Comm comm, MPI_Win *win)	48

1	int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win)
2 3	<pre>int MPI_Win_detach(MPI_Win win, const void *base)</pre>
4	<pre>int MPI_Win_fence(int assert, MPI_Win win)</pre>
5 6	int MPI_Win_flush(int rank, MPI_Win win)
7	int MPI_Win_flush_all(MPI_Win win)
8 9	int MPI_Win_flush_local(int rank, MPI_Win win)
10 11	int MPI_Win_flush_local_all(MPI_Win win)
12	int MPI_Win_free(MPI_Win *win)
13 14	int MPI_Win_get_group(MPI_Win win, MPI_Group *group)
15	<pre>int MPI_Win_get_info(MPI_Win win, MPI_Info *info_used)</pre>
16 17	<pre>int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)</pre>
18	int MPI_Win_lock_all(int assert, MPI_Win win)
19 20	<pre>int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)</pre>
21	int MPI_Win_set_info(MPI_Win win, MPI_Info info)
22 23	<pre>int MPI_Win_shared_query(MPI_Win win, int rank, MPI_Aint *size,</pre>
24	<pre>int *disp_unit, void *baseptr)</pre>
25 26	<pre>int MPI_Win_shared_query_c(MPI_Win win, int rank, MPI_Aint *size, MPI_Aint *disp_unit, void *baseptr)</pre>
27 28	int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
29 30	int MPI_Win_sync(MPI_Win win)
31	<pre>int MPI_Win_test(MPI_Win win, int *flag)</pre>
32 33	int MPI_Win_unlock(int rank, MPI_Win win)
34	int MPI_Win_unlock_all(MPI_Win win)
35 36	int MPI_Win_wait(MPI_Win win)
37	
38 39	A.3.11 External Interfaces C Bindings
40	<pre>int MPI_Grequest_complete(MPI_Request request)</pre>
41 42	<pre>int MPI_Grequest_start(MPI_Grequest_query_function *query_fn,</pre>
43	MPI_Grequest_cancel_function *cancel_fn, void *extra_state,
44 45	MPI_Request *request)
46	<pre>int MPI_Status_set_cancelled(MPI_Status *status, int flag)</pre>
47 48	<pre>int MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype,</pre>

<pre>int MPI_Status_set_elements_x(MPI_Status *status, MPI_Datatype datatype, MPI_Count count)</pre>	1 2 3
A.3.12 I/O C Bindings	4 5
<pre>int MPI_CONVERSION_FN_NULL(void *userbuf, MPI_Datatype datatype, int count, void *filebuf, MPI_Offset position, void *extra_state)</pre>	6 7 8
<pre>int MPI_CONVERSION_FN_NULL_c(void *userbuf, MPI_Datatype datatype, MPI_Count count, void *filebuf, MPI_Offset position, void *extra_state)</pre>	9 10 11
<pre>int MPI_File_close(MPI_File *fh)</pre>	12 13
int MPI_File_delete(const char *filename, MPI_Info info)	14
<pre>int MPI_File_get_amode(MPI_File fh, int *amode)</pre>	15 16
<pre>int MPI_File_get_atomicity(MPI_File fh, int *flag)</pre>	17
<pre>int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset,</pre>	18 19 20
<pre>int MPI_File_get_group(MPI_File fh, MPI_Group *group)</pre>	21
<pre>int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)</pre>	22 23
int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)	24
<pre>int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)</pre>	25 26
int MPI_File_get_size(MPI_File fh, MPI_Offset *size)	27
<pre>int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype,</pre>	28 29 30
<pre>int MPI_File_get_type_extent_c(MPI_File fh, MPI_Datatype datatype, MPI_Count *extent)</pre>	31 32
<pre>int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype, MPI_Datatype *filetype, char *datarep)</pre>	33 34 35
<pre>int MPI_File_iread(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)</pre>	36 37
int MPI_File_iread_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)	38 39 40
int MPI_File_iread_all_c(MPI_File fh, void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Request *request)	41 42 43
int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)	43 44 45
<pre>int MPI_File_iread_at_all(MPI_File fh, MPI_Offset offset, void *buf,</pre>	46 47 48

1 2	int	<pre>MPI_File_iread_at_all_c(MPI_File fh, MPI_Offset offset, void *buf,</pre>
3 4 5	int	<pre>MPI_File_iread_at_c(MPI_File fh, MPI_Offset offset, void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Request *request)</pre>
6 7 8	int	<pre>MPI_File_iread_c(MPI_File fh, void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Request *request)</pre>
9 10	int	<pre>MPI_File_iread_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)</pre>
11 12 13	int	<pre>MPI_File_iread_shared_c(MPI_File fh, void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Request *request)</pre>
14 15	int	<pre>MPI_File_iwrite(MPI_File fh, const void *buf, int count,</pre>
16 17 18	int	<pre>MPI_File_iwrite_all(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Request *request)</pre>
19 20	int	<pre>MPI_File_iwrite_all_c(MPI_File fh, const void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Request *request)</pre>
21 22 23	int	<pre>MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, const void *buf,</pre>
24 25	int	<pre>MPI_File_iwrite_at_all(MPI_File fh, MPI_Offset offset, const void *buf,</pre>
26 27 28 29	int	<pre>MPI_File_iwrite_at_all_c(MPI_File fh, MPI_Offset offset,</pre>
30 31	int	<pre>MPI_File_iwrite_at_c(MPI_File fh, MPI_Offset offset, const void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Request *request)</pre>
32 33 34	int	<pre>MPI_File_iwrite_c(MPI_File fh, const void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Request *request)</pre>
35 36	int	<pre>MPI_File_iwrite_shared(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Request *request)</pre>
37 38 39	int	<pre>MPI_File_iwrite_shared_c(MPI_File fh, const void *buf, MPI_Count count,</pre>
40 41	int	<pre>MPI_File_open(MPI_Comm comm, const char *filename, int amode, MPI_Info info, MPI_File *fh)</pre>
42 43	int	<pre>MPI_File_preallocate(MPI_File fh, MPI_Offset size)</pre>
44 45	int	<pre>MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype,</pre>
46 47 48	int	MPI_File_read_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

int	MPI_File_read_all_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype)	1 $2$
int	<pre>MPI_File_read_all_begin_c(MPI_File fh, void *buf, MPI_Count count, MPI_Datatype datatype)</pre>	3 4 5
int	<pre>MPI_File_read_all_c(MPI_File fh, void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Status *status)</pre>	6 7
int	MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status)	8 9
int	<pre>MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>	10 11
int	<pre>MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,</pre>	12 13 14
int	<pre>MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,</pre>	15 16 17
int	<pre>MPI_File_read_at_all_begin_c(MPI_File fh, MPI_Offset offset, void *buf, MPI_Count count, MPI_Datatype datatype)</pre>	17 18 19
int	<pre>MPI_File_read_at_all_c(MPI_File fh, MPI_Offset offset, void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Status *status)</pre>	20 21 22
int	<pre>MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)</pre>	23
int	<pre>MPI_File_read_at_c(MPI_File fh, MPI_Offset offset, void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Status *status)</pre>	24 25 26
int	<pre>MPI_File_read_c(MPI_File fh, void *buf, MPI_Count count,</pre>	27 28
int	<pre>MPI_File_read_ordered(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>	29 30 31
int	<pre>MPI_File_read_ordered_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype)</pre>	32 33
int	<pre>MPI_File_read_ordered_begin_c(MPI_File fh, void *buf, MPI_Count count, MPI_Datatype datatype)</pre>	34 35 36
int	<pre>MPI_File_read_ordered_c(MPI_File fh, void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Status *status)</pre>	37 38
int	<pre>MPI_File_read_ordered_end(MPI_File fh, void *buf, MPI_Status *status)</pre>	39 40
int	MPI_File_read_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)	41 42
int	<pre>MPI_File_read_shared_c(MPI_File fh, void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Status *status)</pre>	43 44 45
int	MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)	46
int	MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence)	47 48

1	int	MPI_File_set_atomicity(MPI_File fh, int flag)
2 3	int	MPI_File_set_info(MPI_File fh, MPI_Info info)
4	int	MPI_File_set_size(MPI_File fh, MPI_Offset size)
5 6	int	MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
7		<pre>MPI_Datatype filetype, const char *datarep, MPI_Info info)</pre>
8 9	int	MPI_File_sync(MPI_File fh)
10 11	int	<pre>MPI_File_write(MPI_File fh, const void *buf, int count,</pre>
12 13	int	<pre>MPI_File_write_all(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>
14 15 16	int	<pre>MPI_File_write_all_begin(MPI_File fh, const void *buf, int count, MPI_Datatype datatype)</pre>
17 18 19	int	<pre>MPI_File_write_all_begin_c(MPI_File fh, const void *buf,</pre>
20 21	int	<pre>MPI_File_write_all_c(MPI_File fh, const void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Status *status)</pre>
22 23 24	int	<pre>MPI_File_write_all_end(MPI_File fh, const void *buf, MPI_Status *status)</pre>
25 26	int	<pre>MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf,</pre>
27 28 29	int	<pre>MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf,</pre>
30 31	int	<pre>MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset,</pre>
32 33 34	int	<pre>MPI_File_write_at_all_begin_c(MPI_File fh, MPI_Offset offset,</pre>
35 36 37	int	<pre>MPI_File_write_at_all_c(MPI_File fh, MPI_Offset offset,</pre>
38 39 40	int	<pre>MPI_File_write_at_all_end(MPI_File fh, const void *buf, MPI_Status *status)</pre>
41 42	int	<pre>MPI_File_write_at_c(MPI_File fh, MPI_Offset offset, const void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Status *status)</pre>
43 44 45	int	<pre>MPI_File_write_c(MPI_File fh, const void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Status *status)</pre>
46 47 48	int	MPI_File_write_ordered(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

<pre>int MPI_File_write_shared(MPI_File fh, const void *buf, int count,</pre>	int	<pre>MPI_File_write_ordered_begin(MPI_File fh, const void *buf, int count, MPI_Datatype datatype)</pre>	1 $2$
<pre>MPI_Datatype datatype, MPI_Status *status) int MPI_File_write_ordered_end(MPI_File fh, const void *buf,</pre>	int	-	3 4 5
<pre>MPI_Status *status) int MPI_File_write_shared(MPI_File fh, const void *buf, int count,</pre>	int		6 7
<pre>int WPI_FINE_Write_shared(WPI_FINE in, const void *buf, int count,</pre>	int		8 9 10
<pre>int MPI_File_write_shared_c(MPI_File fh, const void *buf, MPI_Count count,</pre>	int		11 12 13
<pre>int NPI_Register_datarep(const char *datarep,</pre>	int		13 14 15
<pre>int MPI_Register_datarep_c(const char *datarep,</pre>	int	<pre>MPI_Datarep_conversion_function *read_conversion_fn, MPI_Datarep_conversion_function *write_conversion_fn, MPI_Datarep_extent_function *dtype_file_extent_fn,</pre>	16 17 18 19 20 21
<pre>int MPI_Status_f082f(const MPI_F08_status *f08_status, MPI_Fint *f_status) int MPI_Status_f2f08(const MPI_Fint *f_status, MPI_F08_status *f08_status) int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype) int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype) int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *datatype) A.3.14 Tools / Profiling Interface C Bindings int MPI_Pcontrol(const int level,) A.3.15 Tools / MPI Tool Information Interface C Bindings int MPI_T_category_changed(int *stamp) int MPI_T category get categories(int cat index, int len, int indices[])</pre>		<pre>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)</pre>	22 23 24 25 26 27 28
<pre>int MPI_Status_f2f08(const MPI_Fint *f_status, MPI_F08_status *f08_status) int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype) int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype) int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *datatype) A.3.14 Tools / Profiling Interface C Bindings int MPI_Pcontrol(const int level,) A.3.15 Tools / MPI Tool Information Interface C Bindings int MPI_T_category_changed(int *stamp) int MPI_T category get categories(int cat index, int len, int indices[])</pre>			29
<pre>int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype) int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype) int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype) int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *datatype) A.3.14 Tools / Profiling Interface C Bindings int MPI_Pcontrol(const int level,) A.3.15 Tools / MPI Tool Information Interface C Bindings int MPI_T_category_changed(int *stamp) int MPI_T category get_categories(int_cat_indexint_lenint_indices[])</pre>			30 31
<pre>int MPI_Type_create_130_complex(int p, int 1, MPI_Datatype *newtype) int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype) int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype) A.3.14 Tools / Profiling Interface C Bindings int MPI_Pcontrol(const int level,) A.3.15 Tools / MPI Tool Information Interface C Bindings int MPI_T_category_changed(int *stamp) int MPI_T_category_get_categories(int_cat_indexint_lenint_indices[])</pre>			32
<pre>int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype) int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *datatype) A.3.14 Tools / Profiling Interface C Bindings int MPI_Pcontrol(const int level,) A.3.15 Tools / MPI Tool Information Interface C Bindings int MPI_T_category_changed(int *stamp) int MPI_T_category_get_categories(int_cat_index, int_len, int_indices[])</pre>	int	<pre>MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype)</pre>	33 34
<pre>int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *Mewtype) int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *datatype) A.3.14 Tools / Profiling Interface C Bindings int MPI_Pcontrol(const int level,) A.3.15 Tools / MPI Tool Information Interface C Bindings int MPI_T_category_changed(int *stamp) int MPI_T_category_get_categories(int_cat_index, int_len, int_indices[])</pre>	int	<pre>MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)</pre>	35
<pre>int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *datatype) A.3.14 Tools / Profiling Interface C Bindings int MPI_Pcontrol(const int level,) A.3.15 Tools / MPI Tool Information Interface C Bindings int MPI_T_category_changed(int *stamp) int MPI_T_category_get_categories(int_cat_index, int_len, int_indices[])</pre>	int	<pre>MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)</pre>	36
<pre>A.3.14 Tools / Profiling Interface C Bindings int MPI_Pcontrol(const int level,) A.3.15 Tools / MPI Tool Information Interface C Bindings int MPI_T_category_changed(int *stamp) int MPI_T_category_get_categories(int_cat_index, int_len, int_indices[])</pre>	int	<pre>MPI_Type_match_size(int typeclass, int size, MPI_Datatype *datatype)</pre>	37 38 39
<pre>int MPI_Pcontrol(const int level,) A.3.15 Tools / MPI Tool Information Interface C Bindings int MPI_T_category_changed(int *stamp) int MPI_T_category_get_categories(int_cat_index, int_len, int_indices[])</pre>	A.3.	14 Tools / Profiling Interface C Bindings	40
A.3.15 Tools / MPI Tool Information Interface C Bindings int MPI_T_category_changed(int *stamp) int MPI_T_category_get_categories(int_cat_index, int_len, int_indices[])	int	MPI_Pcontrol(const int level,)	41 42 43
int MPI T category get categories(int cat index, int len, int indices[])	A.3.	15 Tools / MPI Tool Information Interface C Bindings	44 45
int MPI T category get categories(int cat index, int len, int indices[])	int	MPI_T_category_changed(int *stamp)	46
	int	<pre>MPI_T_category_get_categories(int cat_index, int len, int indices[])</pre>	47 48

1	int	<pre>MPI_T_category_get_cvars(int cat_index, int len, int indices[])</pre>
2 3	int	<pre>MPI_T_category_get_events(int cat_index, int len, int indices[])</pre>
4	int	<pre>MPI_T_category_get_index(const char *name, int *cat_index)</pre>
5 6 7 8	int	<pre>MPI_T_category_get_info(int cat_index, char *name, int *name_len,</pre>
9	int	MPI_T_category_get_num(int *num_cat)
10 11	int	MPI_T_category_get_num_events(int cat_index, int *num_events)
12 13	int	<pre>MPI_T_category_get_pvars(int cat_index, int len, int indices[])</pre>
13	int	MPI_T_cvar_get_index(const char *name, int *cvar_index)
15 16 17 18	int	<pre>MPI_T_cvar_get_info(int cvar_index, char *name, int *name_len,</pre>
19	int	MPI_T_cvar_get_num(int *num_cvar)
20 21 22	int	<pre>MPI_T_cvar_handle_alloc(int cvar_index, void *obj_handle, MPI_T_cvar_handle *handle, int *count)</pre>
23	int	MPI_T_cvar_handle_free(MPI_T_cvar_handle *handle)
24 25	int	<pre>MPI_T_cvar_read(MPI_T_cvar_handle handle, void *buf)</pre>
26	int	<pre>MPI_T_cvar_write(MPI_T_cvar_handle handle, const void *buf)</pre>
27 28 29	int	<pre>MPI_T_enum_get_info(MPI_T_enum enumtype, int *num, char *name,</pre>
30 31	int	<pre>MPI_T_enum_get_item(MPI_T_enum enumtype, int index, int *value,</pre>
32 33 34 35	int	<pre>MPI_T_event_callback_get_info(</pre>
36 37 38 39	int	<pre>MPI_T_event_callback_set_info(     MPI_T_event_registration event_registration,     MPI_T_cb_safety cb_safety, MPI_Info info)</pre>
40	int	<pre>MPI_T_event_copy(MPI_T_event_instance event_instance, void *buffer)</pre>
41 42	int	<pre>MPI_T_event_get_index(const char *name, int *event_index)</pre>
43 44 45 46 47	int	<pre>MPI_T_event_get_info(int event_index, char *name, int *name_len,</pre>
48		

int	MPI_T_event_get_num(int *num_events)	1
int	<pre>MPI_T_event_get_source(MPI_T_event_instance event_instance,</pre>	2 3 4
int	<pre>MPI_T_event_get_timestamp(MPI_T_event_instance event_instance,</pre>	5 6
int	<pre>MPI_T_event_handle_alloc(int event_index, void *obj_handle, MPI_Info info, MPI_T_event_registration *event_registration)</pre>	7 8 9
int	<pre>MPI_T_event_handle_free(MPI_T_event_registration event_registration,</pre>	10 11 12
int	<pre>MPI_T_event_handle_get_info(</pre>	13 14 15 16
int	<pre>MPI_T_event_handle_set_info(</pre>	17 18 19
int	<pre>MPI_T_event_read(MPI_T_event_instance event_instance,</pre>	19 20 21
int	<pre>MPI_T_event_register_callback(</pre>	22 23 24 25
int	<pre>MPI_T_event_set_dropped_handler(     MPI_T_event_registration event_registration,     MPI_T_event_dropped_cb_function dropped_cb_function)</pre>	26 27 28 29
int	MPI_T_finalize(void)	30 31
int	MPI_T_init_thread(int required, int *provided)	32
int	<pre>MPI_T_pvar_get_index(const char *name, int var_class, int *pvar_index)</pre>	33 34
int	<pre>MPI_T_pvar_get_info(int pvar_index, char *name, int *name_len,</pre>	35 36 37 38
int	MPI_T_pvar_get_num(int *num_pvar)	39 40
int	<pre>MPI_T_pvar_handle_alloc(MPI_T_pvar_session pe_session, int pvar_index, void *obj_handle, MPI_T_pvar_handle *handle, int *count)</pre>	41 42
int	<pre>MPI_T_pvar_handle_free(MPI_T_pvar_session pe_session, MPI_T_pvar_handle *handle)</pre>	43 44 45
int	<pre>MPI_T_pvar_read(MPI_T_pvar_session pe_session,</pre>	46 47 48

1 2	int	<pre>MPI_T_pvar_readreset(MPI_T_pvar_session pe_session,</pre>
3 4 5	int	<pre>MPI_T_pvar_reset(MPI_T_pvar_session pe_session,</pre>
6	int	MPI_T_pvar_session_create(MPI_T_pvar_session *pe_session)
7 8	int	<pre>MPI_T_pvar_session_free(MPI_T_pvar_session *pe_session)</pre>
9 10 11	int	<pre>MPI_T_pvar_start(MPI_T_pvar_session pe_session,</pre>
12 13	int	<pre>MPI_T_pvar_stop(MPI_T_pvar_session pe_session,</pre>
14 15 16	int	<pre>MPI_T_pvar_write(MPI_T_pvar_session pe_session,</pre>
17 18 19 20	int	<pre>MPI_T_source_get_info(int source_index, char *name, int *name_len,</pre>
21 22	int	MPI_T_source_get_num(int *num_sources)
23 24	int	<pre>MPI_T_source_get_timestamp(int source_index, MPI_Count *timestamp)</pre>
25 26	A.3.	16 Deprecated C Bindings
26 27		16 Deprecated C Bindings MPI_Attr_delete(MPI_Comm comm, int keyval)
26	int	
26 27 28 29 30	int int	MPI_Attr_delete(MPI_Comm comm, int keyval)
26 27 28 29	int int int	MPI_Attr_delete(MPI_Comm comm, int keyval) MPI_Attr_get(MPI_Comm comm, int keyval, void *attribute_val, int *flag)
26 27 28 29 30 31 32 33 34 35	int int int int	<pre>MPI_Attr_delete(MPI_Comm comm, int keyval) MPI_Attr_get(MPI_Comm comm, int keyval, void *attribute_val, int *flag) MPI_Attr_put(MPI_Comm comm, int keyval, void *attribute_val) MPI_DUP_FN(MPI_Comm oldcomm, int keyval, void *extra_state,</pre>
26 27 28 29 30 31 32 33 34	int int int int	<pre>MPI_Attr_delete(MPI_Comm comm, int keyval) MPI_Attr_get(MPI_Comm comm, int keyval, void *attribute_val, int *flag) MPI_Attr_put(MPI_Comm comm, int keyval, void *attribute_val) MPI_DUP_FN(MPI_Comm oldcomm, int keyval, void *extra_state,</pre>
26 27 28 29 30 31 32 33 34 35 36 37	int int int int int	<pre>MPI_Attr_delete(MPI_Comm comm, int keyval) MPI_Attr_get(MPI_Comm comm, int keyval, void *attribute_val, int *flag) MPI_Attr_put(MPI_Comm comm, int keyval, void *attribute_val) MPI_DUP_FN(MPI_Comm oldcomm, int keyval, void *extra_state,</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41	int int int int int	<pre>MPI_Attr_delete(MPI_Comm comm, int keyval) MPI_Attr_get(MPI_Comm comm, int keyval, void *attribute_val, int *flag) MPI_Attr_put(MPI_Comm comm, int keyval, void *attribute_val) MPI_DUP_FN(MPI_Comm oldcomm, int keyval, void *extra_state,</pre>
26 27 28 30 31 32 33 34 35 36 37 38 39 40 41	int int int int int int	<pre>MPI_Attr_delete(MPI_Comm comm, int keyval) MPI_Attr_get(MPI_Comm comm, int keyval, void *attribute_val, int *flag) MPI_Attr_put(MPI_Comm comm, int keyval, void *attribute_val) MPI_DUP_FN(MPI_Comm oldcomm, int keyval, void *extra_state,</pre>

A.4. FORTRAN 2008 BINDINGS WITH THE MPI_F08 MODULE	911
A.4 Fortran 2008 Bindings with the mpi_f08 Module	1
A.4.1 Point-to-Point Communication Fortran 2008 Bindings	2
-	4
<pre>MPI_Bsend(buf, count, datatype, dest, tag, comm, ierror)     TYPE(*), DIMENSION(), INTENT(IN) :: buf</pre>	5
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count	6
TYPE(MPI_Datatype), INTENT(IN) :: datatype	7
INTEGER, INTENT(IN) :: dest, tag	8
TYPE(MPI_Comm), INTENT(IN) :: comm	9 10
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10
MPI_Bsend(buf, count, datatype, dest, tag, comm, ierror)	12
TYPE(*), DIMENSION(), INTENT(IN) :: buf	13
INTEGER, INTENT(IN) :: count, dest, tag	14
TYPE(MPI_Datatype), INTENT(IN) :: datatype	15
TYPE(MPI_Comm), INTENT(IN) :: comm	16 17
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	17
MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror)	19
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	20
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype	21
INTEGER, INTENT(IN) :: dest, tag	22
TYPE(MPI_Comm), INTENT(IN) :: comm	23
TYPE(MPI_Request), INTENT(OUT) :: request	24 25
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	25
MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror)	27
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	28
INTEGER, INTENT(IN) :: count, dest, tag	29
TYPE(MPI_Datatype), INTENT(IN) :: datatype	30
TYPE(MPI_Comm), INTENT(IN) :: comm	31
TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror	32 33
	34
MPI_Buffer_attach(buffer, size, ierror)	35
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buffer	36
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: size INTEGER, OPTIONAL, INTENT(OUT) :: ierror	37
	38
MPI_Buffer_attach(buffer, size, ierror)	39 40
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buffer INTEGER, INTENT(IN) :: size	40
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	42
	43
MPI_Buffer_detach(buffer_addr, size, ierror)	44
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR TYPE(C_PTR), INTENT(OUT) :: buffer_addr	45
INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size	46
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	47 48
	-40

```
1
 MPI_Buffer_detach(buffer_addr, size, ierror)
\mathbf{2}
 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
3
 TYPE(C_PTR), INTENT(OUT) :: buffer_addr
4
 INTEGER, INTENT(OUT) :: size
5
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
 MPI_Cancel(request, ierror)
7
 TYPE(MPI_Request), INTENT(IN) :: request
8
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10
 MPI_Get_count(status, datatype, count, ierror)
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
 MPI_Get_count(status, datatype, count, ierror)
16
 TYPE(MPI_Status), INTENT(IN) :: status
17
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
18
 INTEGER, INTENT(OUT) :: count
19
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
 MPI_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror)
22
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
23
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
24
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
25
 INTEGER, INTENT(IN) :: dest, tag
26
 TYPE(MPI_Comm), INTENT(IN) :: comm
27
 TYPE(MPI_Request), INTENT(OUT) :: request
28
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
 MPI_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror)
30
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
31
 INTEGER, INTENT(IN) :: count, dest, tag
32
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
33
 TYPE(MPI_Comm), INTENT(IN) :: comm
34
 TYPE(MPI_Request), INTENT(OUT) :: request
35
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
 MPI_Improbe(source, tag, comm, flag, message, status, ierror)
38
 INTEGER, INTENT(IN) :: source, tag
39
 TYPE(MPI_Comm), INTENT(IN) :: comm
40
 LOGICAL, INTENT(OUT) :: flag
41
 TYPE(MPI_Message), INTENT(OUT) :: message
42
 TYPE(MPI_Status) :: status
43
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
 MPI_Imrecv(buf, count, datatype, message, request, ierror)
45
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
46
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
47
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
48
```

```
1
 TYPE(MPI_Message), INTENT(INOUT) :: message
 TYPE(MPI_Request), INTENT(OUT) :: request
 2
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 4
MPI_Imrecv(buf, count, datatype, message, request, ierror)
 5
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
 6
 INTEGER, INTENT(IN) :: count
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 TYPE(MPI_Message), INTENT(INOUT) :: message
 9
 TYPE(MPI_Request), INTENT(OUT) :: request
 10
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 11
MPI_Iprobe(source, tag, comm, flag, status, ierror)
 12
 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
MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror)
 19
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
 20
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
 21
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 22
 INTEGER, INTENT(IN) :: source, tag
 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)
 28
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
 29
 INTEGER, INTENT(IN) :: count, source, tag
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 30
 31
 TYPE(MPI_Comm), INTENT(IN) :: comm
 32
 TYPE(MPI_Request), INTENT(OUT) :: request
 33
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 34
MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror)
 35
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
 36
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
 37
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 38
 INTEGER, INTENT(IN) :: dest, tag
 39
 TYPE(MPI_Comm), INTENT(IN) :: comm
 40
 TYPE(MPI_Request), INTENT(OUT) :: request
 41
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 42
 43
MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror)
 44
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
 45
 INTEGER, INTENT(IN) :: count, dest, tag
 46
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 47
 TYPE(MPI_Comm), INTENT(IN) :: comm
 48
```

```
1
 TYPE(MPI_Request), INTENT(OUT) :: request
2
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
 MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror)
4
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
5
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
6
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
7
 INTEGER, INTENT(IN) :: dest, tag
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)
13
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
14
 INTEGER, INTENT(IN) :: count, dest, tag
15
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
16
 TYPE(MPI_Comm), INTENT(IN) :: comm
17
 TYPE(MPI_Request), INTENT(OUT) :: request
18
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
 MPI_Isendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
20
 recvcount, recvtype, source, recvtag, comm, request, ierror)
21
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
22
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
23
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
24
 INTEGER, INTENT(IN) :: dest, sendtag, source, recvtag
25
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
26
 TYPE(MPI_Comm), INTENT(IN) :: comm
27
 TYPE(MPI_Request), INTENT(OUT) :: request
28
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
 MPI_Isendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
31
 recvcount, recvtype, source, recvtag, comm, request, ierror)
32
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
33
 INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source,
34
 recvtag
35
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
36
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
37
 TYPE(MPI_Comm), INTENT(IN) :: comm
38
 TYPE(MPI_Request), INTENT(OUT) :: request
39
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
 MPI_Isendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
41
 comm, request, ierror)
42
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
43
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
44
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
45
 INTEGER, INTENT(IN) :: dest, sendtag, source, recvtag
46
 TYPE(MPI_Comm), INTENT(IN) :: comm
47
 TYPE(MPI_Request), INTENT(OUT) :: request
48
```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1 2 MPI_Isendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag, comm, request, ierror) 4 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf 5 INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag 6 TYPE(MPI_Datatype), INTENT(IN) :: datatype 7 TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request 9 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 10 11 MPI_Issend(buf, count, datatype, dest, tag, comm, request, ierror) TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf 1213 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count 14TYPE(MPI_Datatype), INTENT(IN) :: datatype 15INTEGER, INTENT(IN) :: dest, tag 16TYPE(MPI_Comm), INTENT(IN) :: comm 17TYPE(MPI_Request), INTENT(OUT) :: request 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 19 MPI_Issend(buf, count, datatype, dest, tag, comm, request, ierror) 20TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf 21INTEGER, INTENT(IN) :: count, dest, tag 22 TYPE(MPI_Datatype), INTENT(IN) :: datatype 23TYPE(MPI_Comm), INTENT(IN) :: comm 24TYPE(MPI_Request), INTENT(OUT) :: request 25INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2627MPI_Mprobe(source, tag, comm, message, status, ierror) 28INTEGER, INTENT(IN) :: source, tag 29 TYPE(MPI_Comm), INTENT(IN) :: comm 30 TYPE(MPI_Message), INTENT(OUT) :: message 31TYPE(MPI_Status) :: status 32 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 33 MPI_Mrecv(buf, count, datatype, message, status, ierror) 34 TYPE(*), DIMENSION(..) :: buf 35 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count 36 TYPE(MPI_Datatype), INTENT(IN) :: datatype 37 TYPE(MPI_Message), INTENT(INOUT) :: message 38 TYPE(MPI_Status) :: status 39 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 40 41 MPI_Mrecv(buf, count, datatype, message, status, ierror) 42TYPE(*), DIMENSION(..) :: buf 43 INTEGER, INTENT(IN) :: count 44 TYPE(MPI_Datatype), INTENT(IN) :: datatype 45TYPE(MPI_Message), INTENT(INOUT) :: message 46TYPE(MPI_Status) :: status 47INTEGER, OPTIONAL, INTENT(OUT) :: ierror 48

```
1
 MPI_Probe(source, tag, comm, status, ierror)
\mathbf{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
 MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)
7
 TYPE(*), DIMENSION(..) :: buf
8
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
9
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
10
 INTEGER, INTENT(IN) :: source, tag
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)
16
 TYPE(*), DIMENSION(..) :: buf
17
 INTEGER, INTENT(IN) :: count, source, tag
18
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
19
 TYPE(MPI_Comm), INTENT(IN) :: comm
20
 TYPE(MPI_Status) :: status
21
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
 MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror)
23
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
24
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
25
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
26
 INTEGER, INTENT(IN) :: source, tag
27
 TYPE(MPI_Comm), INTENT(IN) :: comm
28
 TYPE(MPI_Request), INTENT(OUT) :: request
29
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31
 MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror)
32
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
33
 INTEGER, INTENT(IN) :: count, source, tag
34
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
35
 TYPE(MPI_Comm), INTENT(IN) :: comm
36
 TYPE(MPI_Request), INTENT(OUT) :: request
37
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
 MPI_Request_free(request, ierror)
39
 TYPE(MPI_Request), INTENT(INOUT) :: request
40
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
 MPI_Request_get_status(request, flag, status, ierror)
43
 TYPE(MPI_Request), INTENT(IN) :: request
44
 LOGICAL, INTENT(OUT) :: flag
45
 TYPE(MPI_Status) :: status
46
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
 MPI_Rsend(buf, count, datatype, dest, tag, comm, ierror)
48
```

TYPE(*), DIMENSION(..), INTENT(IN) :: buf 1 2 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 5 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 6 MPI_Rsend(buf, count, datatype, dest, tag, comm, ierror) TYPE(*), DIMENSION(..), INTENT(IN) :: buf 9 INTEGER, INTENT(IN) :: count, dest, tag 10 TYPE(MPI_Datatype), INTENT(IN) :: datatype 11 TYPE(MPI_Comm), INTENT(IN) :: comm 12INTEGER, OPTIONAL, INTENT(OUT) :: ierror 13 MPI_Rsend_init(buf, count, datatype, dest, tag, comm, request, ierror) 1415TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 16INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count 17TYPE(MPI_Datatype), INTENT(IN) :: datatype 18 INTEGER, INTENT(IN) :: dest, tag 19 TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request 2021INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22 MPI_Rsend_init(buf, count, datatype, dest, tag, comm, request, ierror) 23TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 24INTEGER, INTENT(IN) :: count, dest, tag 25TYPE(MPI_Datatype), INTENT(IN) :: datatype 26TYPE(MPI_Comm), INTENT(IN) :: comm 27TYPE(MPI_Request), INTENT(OUT) :: request 28INTEGER, OPTIONAL, INTENT(OUT) :: ierror 29 30 MPI_Send(buf, count, datatype, dest, tag, comm, ierror) 31TYPE(*), DIMENSION(..), INTENT(IN) :: buf 32 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count 33 TYPE(MPI_Datatype), INTENT(IN) :: datatype 34 INTEGER, INTENT(IN) :: dest, tag 35 TYPE(MPI_Comm), INTENT(IN) :: comm 36 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 37 MPI_Send(buf, count, datatype, dest, tag, comm, ierror) 38 TYPE(*), DIMENSION(...), INTENT(IN) :: buf 39 INTEGER, INTENT(IN) :: count, dest, tag 40 TYPE(MPI_Datatype), INTENT(IN) :: datatype 41 TYPE(MPI_Comm), INTENT(IN) :: comm 42INTEGER, OPTIONAL, INTENT(OUT) :: ierror 43 44MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror) 45TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 46INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count 47TYPE(MPI_Datatype), INTENT(IN) :: datatype 48

```
1
 INTEGER, INTENT(IN) :: dest, tag
2
 TYPE(MPI_Comm), INTENT(IN) :: comm
3
 TYPE(MPI_Request), INTENT(OUT) :: request
4
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
 MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)
6
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
7
 INTEGER, INTENT(IN) :: count, dest, tag
8
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
9
 TYPE(MPI_Comm), INTENT(IN) :: comm
10
 TYPE(MPI_Request), INTENT(OUT) :: request
11
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13
 MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
14
 recvcount, recvtype, source, recvtag, comm, status, ierror)
15
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
16
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
17
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
18
 INTEGER, INTENT(IN) :: dest, sendtag, source, recvtag
19
 TYPE(*), DIMENSION(...) :: recvbuf
20
 TYPE(MPI_Comm), INTENT(IN) :: comm
21
 TYPE(MPI_Status) :: status
22
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
 MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
24
 recvcount, recvtype, source, recvtag, comm, status, ierror)
25
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
26
 INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source,
27
 recvtag
28
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
29
 TYPE(*), DIMENSION(..) :: recvbuf
30
 TYPE(MPI_Comm), INTENT(IN) :: comm
31
 TYPE(MPI_Status) :: status
32
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
 MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
35
 comm, status, ierror)
36
 TYPE(*), DIMENSION(..) :: buf
37
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
38
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
39
 INTEGER, INTENT(IN) :: dest, sendtag, source, recvtag
40
 TYPE(MPI_Comm), INTENT(IN) :: comm
41
 TYPE(MPI_Status) :: status
42
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
 MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
44
 comm, status, ierror)
45
 TYPE(*), DIMENSION(..) :: buf
46
 INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag
47
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
48
```

TYPE(MPI_Comm), INTENT(IN) :: comm	1
TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror	2 3
INIEGER, OFIIONAL, INIENI(UOI) IEIIOI	4
MPI_Ssend(buf, count, datatype, dest, tag, comm, ierror)	5
TYPE(*), DIMENSION(), INTENT(IN) :: buf	6
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count	7
TYPE(MPI_Datatype), INTENT(IN) :: datatype	
INTEGER, INTENT(IN) :: dest, tag	8
TYPE(MPI_Comm), INTENT(IN) :: comm	9
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10
	11
MPI_Ssend(buf, count, datatype, dest, tag, comm, ierror)	12
TYPE(*), DIMENSION(), INTENT(IN) :: buf	13
INTEGER, INTENT(IN) :: count, dest, tag	14
TYPE(MPI_Datatype), INTENT(IN) :: datatype	15
TYPE(MPI_Comm), INTENT(IN) :: comm	16
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	17
MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror)	18 19
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	20
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count	20
TYPE(MPI_Datatype), INTENT(IN) :: datatype	21
INTEGER, INTENT(IN) :: dest, tag	22
TYPE(MPI_Comm), INTENT(IN) :: comm	23 24
TYPE(MPI_Request), INTENT(OUT) :: request	24 25
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	25 26
	20
MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror)	
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	28
INTEGER, INTENT(IN) :: count, dest, 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_Start(request, ierror)	34
TYPE(MPI_Request), INTENT(INOUT) :: request	35
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	36
INTEGER, OFFICIARE, INTENT(COT) TETTOT	37
<pre>MPI_Startall(count, array_of_requests, ierror)</pre>	38
INTEGER, INTENT(IN) :: count	39
TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)	40
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	41
MPI_Test(request, flag, status, ierror)	42
TYPE(MPI_Request), INTENT(INOUT) :: request	43
• •	44
LOGICAL, INTENT(OUT) :: flag	45
TYPE(MPI_Status) :: status	46
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	47
	48

```
1
 MPI_Test_cancelled(status, flag, ierror)
\mathbf{2}
 TYPE(MPI_Status), INTENT(IN) :: status
3
 LOGICAL, INTENT(OUT) :: flag
4
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
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
 MPI_Testsome(incount, array_of_requests, outcount, array_of_indices,
20
 array_of_statuses, ierror)
21
 INTEGER, INTENT(IN) :: incount
22
 TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)
23
 INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
24
 TYPE(MPI_Status) :: array_of_statuses(*)
25
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27
 MPI_Wait(request, status, ierror)
28
 TYPE(MPI_Request), INTENT(INOUT) :: request
29
 TYPE(MPI_Status) :: status
30
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
 MPI_Waitall(count, array_of_requests, array_of_statuses, ierror)
32
 INTEGER, INTENT(IN) :: count
33
 TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
34
 TYPE(MPI_Status) :: array_of_statuses(*)
35
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
 MPI_Waitany(count, array_of_requests, index, status, ierror)
38
 INTEGER, INTENT(IN) :: count
39
 TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
40
 INTEGER, INTENT(OUT) :: index
41
 TYPE(MPI_Status) :: status
42
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
 MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices,
44
 array_of_statuses, ierror)
45
 INTEGER, INTENT(IN) :: incount
46
 TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)
47
 INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
48
```

```
1
 TYPE(MPI_Status) :: array_of_statuses(*)
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 2
 4
A.4.2 Partitioned Communication Fortran 2008 Bindings
 5
 6
MPI_Parrived(request, partition, flag, ierror)
 TYPE(MPI_Request), INTENT(INOUT) :: request
 INTEGER, INTENT(IN) :: partition
 9
 LOGICAL, INTENT(OUT) :: flag
 10
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 11
MPI_Pready(partition, request, ierror)
 12
 INTEGER, INTENT(IN) :: partition
 13
 TYPE(MPI_Request), INTENT(INOUT) :: request
 14
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 15
 16
MPI_Pready_list(length, array_of_partitions, request, ierror)
 17
 INTEGER, INTENT(IN) :: length, array_of_partitions(length)
 18
 TYPE(MPI_Request), INTENT(INOUT) :: request
 19
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 20
MPI_Pready_range(partition_low, partition_high, request, ierror)
 21
 INTEGER, INTENT(IN) :: partition_low, partition_high
 22
 TYPE(MPI_Request), INTENT(INOUT) :: request
 23
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 24
 25
MPI_Precv_init(buf, partitions, count, datatype, dest, tag, comm, info,
 26
 request, ierror)
 27
 TYPE(*), DIMENSION(..), INTENT(IN) :: buf
 28
 INTEGER, INTENT(IN) :: partitions, dest, 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
MPI_Psend_init(buf, partitions, count, datatype, dest, tag, comm, info,
 36
 request, ierror)
 37
 TYPE(*), DIMENSION(...), INTENT(IN) :: buf
 38
 INTEGER, INTENT(IN) :: partitions, dest, tag
 39
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
 40
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 41
 TYPE(MPI_Comm), INTENT(IN) :: comm
 42
 TYPE(MPI_Info), INTENT(IN) :: info
 43
 TYPE(MPI_Request), INTENT(OUT) :: request
 44
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 45
 46
 47
 48
```

1 A.4.3 Datatypes Fortran 2008 Bindings  $\mathbf{2}$ INTEGER(KIND=MPI_ADDRESS_KIND) MPI_Aint_add(base, disp) 3 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: base, disp 4 5INTEGER(KIND=MPI_ADDRESS_KIND) MPI_Aint_diff(addr1, addr2) 6 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: addr1, addr2 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 12MPI_Get_elements(status, datatype, count, ierror) 13 TYPE(MPI_Status), INTENT(IN) :: status 14TYPE(MPI_Datatype), INTENT(IN) :: datatype 15INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: count 16INTEGER, OPTIONAL, INTENT(OUT) :: ierror 17 MPI_Get_elements(status, datatype, count, ierror) 18 TYPE(MPI_Status), INTENT(IN) :: status 19 TYPE(MPI_Datatype), INTENT(IN) :: datatype 20INTEGER, INTENT(OUT) :: count 21INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22 23MPI_Get_elements_x(status, datatype, count, ierror) 24TYPE(MPI_Status), INTENT(IN) :: status 25TYPE(MPI_Datatype), INTENT(IN) :: datatype 26INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: count 27INTEGER, OPTIONAL, INTENT(OUT) :: ierror 28MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror) 29 TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf 30 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: incount, outsize 31 TYPE(MPI_Datatype), INTENT(IN) :: datatype 32 TYPE(*), DIMENSION(..) :: outbuf 33 INTEGER(KIND=MPI_COUNT_KIND), INTENT(INOUT) :: position 34 TYPE(MPI_Comm), INTENT(IN) :: comm 35INTEGER, OPTIONAL, INTENT(OUT) :: ierror 36 37 MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror) 38 TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf 39 INTEGER, INTENT(IN) :: incount, outsize 40TYPE(MPI_Datatype), INTENT(IN) :: datatype 41 TYPE(*), DIMENSION(...) :: outbuf 42INTEGER, INTENT(INOUT) :: position 43 TYPE(MPI_Comm), INTENT(IN) :: comm 44 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 45 MPI_Pack_external(datarep, inbuf, incount, datatype, outbuf, outsize, 46position, ierror) 47 CHARACTER(LEN=*), INTENT(IN) :: datarep 48

TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf 1 2 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 5 6 INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Pack_external(datarep, inbuf, incount, datatype, outbuf, outsize, position, ierror) a CHARACTER(LEN=*), INTENT(IN) :: datarep 10 TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf 11 INTEGER, INTENT(IN) :: incount 12TYPE(MPI_Datatype), INTENT(IN) :: datatype 13 TYPE(*), DIMENSION(..) :: outbuf 14INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: outsize 15INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position 16INTEGER, OPTIONAL, INTENT(OUT) :: ierror 17 18 MPI_Pack_external_size(datarep, incount, datatype, size, ierror) 19 CHARACTER(LEN=*), INTENT(IN) :: datarep INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: incount 2021TYPE(MPI_Datatype), INTENT(IN) :: datatype 22 INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size 23INTEGER, OPTIONAL, INTENT(OUT) :: ierror 24MPI_Pack_external_size(datarep, incount, datatype, size, ierror) 25CHARACTER(LEN=*), INTENT(IN) :: datarep 26INTEGER, INTENT(IN) :: incount 27TYPE(MPI_Datatype), INTENT(IN) :: datatype 28INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size 29 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 30 31MPI_Pack_size(incount, datatype, comm, size, ierror) 32 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: incount 33 TYPE(MPI_Datatype), INTENT(IN) :: datatype 34 TYPE(MPI_Comm), INTENT(IN) :: comm 35 INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size 36 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 37 MPI_Pack_size(incount, datatype, comm, size, ierror) 38 INTEGER, INTENT(IN) :: incount 39 TYPE(MPI_Datatype), INTENT(IN) :: datatype 40 TYPE(MPI_Comm), INTENT(IN) :: comm 41 INTEGER, INTENT(OUT) :: size 42INTEGER, OPTIONAL, INTENT(OUT) :: ierror 43 44 MPI_Type_commit(datatype, ierror) 45TYPE(MPI_Datatype), INTENT(INOUT) :: datatype 46INTEGER, OPTIONAL, INTENT(OUT) :: ierror 47MPI_Type_contiguous(count, oldtype, newtype, ierror) 48

```
1
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
2
 TYPE(MPI_Datatype), INTENT(IN) :: oldtype
3
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
4
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
 MPI_Type_contiguous(count, oldtype, newtype, ierror)
6
 INTEGER, INTENT(IN) :: count
7
 TYPE(MPI_Datatype), INTENT(IN) :: oldtype
8
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
9
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
 MPI_Type_create_darray(size, rank, ndims, array_of_gsizes,
12
 array_of_distribs, array_of_dargs, array_of_psizes, order,
13
 oldtype, newtype, ierror)
14
 INTEGER, INTENT(IN) :: size, rank, ndims, array_of_distribs(ndims),
15
 array_of_dargs(ndims), array_of_psizes(ndims), order
16
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: array_of_gsizes(ndims)
17
 TYPE(MPI_Datatype), INTENT(IN) :: oldtype
18
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
19
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
 MPI_Type_create_darray(size, rank, ndims, array_of_gsizes,
21
 array_of_distribs, array_of_dargs, array_of_psizes, order,
22
 oldtype, newtype, ierror)
23
 INTEGER, INTENT(IN) :: size, rank, ndims, array_of_gsizes(ndims),
24
 array_of_distribs(ndims), array_of_dargs(ndims),
25
 array_of_psizes(ndims), order
26
 TYPE(MPI_Datatype), INTENT(IN) :: oldtype
27
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
28
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
 MPI_Type_create_hindexed(count, array_of_blocklengths,
31
 array_of_displacements, oldtype, newtype, ierror)
32
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count,
33
 array_of_blocklengths(count), array_of_displacements(count)
34
 TYPE(MPI_Datatype), INTENT(IN) :: oldtype
35
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
36
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
 MPI_Type_create_hindexed(count, array_of_blocklengths,
38
 array_of_displacements, oldtype, newtype, ierror)
39
 INTEGER, INTENT(IN) :: count, array_of_blocklengths(count)
40
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
41
 array_of_displacements(count)
42
 TYPE(MPI_Datatype), INTENT(IN) :: oldtype
43
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
44
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
 MPI_Type_create_hindexed_block(count, blocklength, array_of_displacements,
47
 oldtype, newtype, ierror)
48
```

```
1
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count, blocklength,
 2
 array_of_displacements(count)
 TYPE(MPI_Datatype), INTENT(IN) :: oldtype
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
 \mathbf{4}
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 5
 6
MPI_Type_create_hindexed_block(count, blocklength, array_of_displacements,
 oldtype, newtype, ierror)
 INTEGER, INTENT(IN) :: count, blocklength
 9
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
 10
 array_of_displacements(count)
 11
 TYPE(MPI_Datatype), INTENT(IN) :: oldtype
 12
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
 13
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 14
 15
MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,
 16
 ierror)
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count, blocklength, stride
 17
 18
 TYPE(MPI_Datatype), INTENT(IN) :: oldtype
 19
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 20
 21
MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,
 22
 ierror)
 23
 INTEGER, INTENT(IN) :: count, blocklength
 24
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: stride
 25
 TYPE(MPI_Datatype), INTENT(IN) :: oldtype
 26
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
 27
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 28
 29
MPI_Type_create_indexed_block(count, blocklength, array_of_displacements,
 30
 oldtype, newtype, ierror)
 31
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count, blocklength,
 32
 array_of_displacements(count)
 33
 TYPE(MPI_Datatype), INTENT(IN) :: oldtype
 34
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
 35
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 36
MPI_Type_create_indexed_block(count, blocklength, array_of_displacements,
 37
 oldtype, newtype, ierror)
 38
 INTEGER, INTENT(IN) :: count, blocklength,
 39
 array_of_displacements(count)
 40
 TYPE(MPI_Datatype), INTENT(IN) :: oldtype
 41
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
 42
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 43
 44
MPI_Type_create_resized(oldtype, lb, extent, newtype, ierror)
 45
 TYPE(MPI_Datatype), INTENT(IN) :: oldtype
 46
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: lb, extent
 47
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
 48
```

```
1
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
 MPI_Type_create_resized(oldtype, lb, extent, newtype, ierror)
3
 TYPE(MPI_Datatype), INTENT(IN) :: oldtype
4
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: lb, extent
5
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
6
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
 MPI_Type_create_struct(count, array_of_blocklengths,
9
 array_of_displacements, array_of_types, newtype, ierror)
10
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count,
11
 array_of_blocklengths(count), array_of_displacements(count)
12
 TYPE(MPI_Datatype), INTENT(IN) :: array_of_types(count)
13
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
14
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
 MPI_Type_create_struct(count, array_of_blocklengths,
16
 array_of_displacements, array_of_types, newtype, ierror)
17
 INTEGER, INTENT(IN) :: count, array_of_blocklengths(count)
18
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
19
 array_of_displacements(count)
20
 TYPE(MPI_Datatype), INTENT(IN) :: array_of_types(count)
21
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
22
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
24
 MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,
25
 array_of_starts, order, oldtype, newtype, ierror)
26
 INTEGER, INTENT(IN) :: ndims, array_of_sizes(ndims),
27
 array_of_subsizes(ndims), array_of_starts(ndims), order
28
 TYPE(MPI_Datatype), INTENT(IN) :: oldtype
29
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
30
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
 MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,
32
 array_of_starts, order, oldtype, newtype, ierror)
33
 INTEGER, INTENT(IN) :: ndims, order
34
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: array_of_sizes(ndims),
35
 array_of_subsizes(ndims), array_of_starts(ndims)
36
 TYPE(MPI_Datatype), INTENT(IN) :: oldtype
37
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
38
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
 MPI_Type_dup(oldtype, newtype, ierror)
41
 TYPE(MPI_Datatype), INTENT(IN) :: oldtype
42
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
43
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
 MPI_Type_free(datatype, ierror)
45
 TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
46
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
```

<pre>MPI_Type_get_contents(datatype, max_integers, max_addresses, max_datatypes,</pre>	1
<pre>array_of_integers, array_of_addresses, array_of_datatypes,</pre>	2
ierror)	3
TYPE(MPI_Datatype), INTENT(IN) :: datatype	4
<pre>INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes</pre>	5
<pre>INTEGER, INTENT(OUT) :: array_of_integers(max_integers)</pre>	6
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::	7
<pre>array_of_addresses(max_addresses)</pre>	8
TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes)	9
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10
MPI_Type_get_contents(datatype, max_integers, max_addresses,	11 12
<pre>max_large_counts, max_datatypes, array_of_integers,</pre>	12
array_of_addresses, array_of_large_counts, array_of_datatypes,	13
ierror)	14
TYPE(MPI_Datatype), INTENT(IN) :: datatype	16
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: max_integers,	10
max_addresses, max_large_counts, max_datatypes	18
INTEGER, INTENT(OUT) :: array_of_integers(max_integers)	19
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::	20
array_of_addresses(max_addresses)	20 21
INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) ::	21
array_of_large_counts(max_large_counts)	22
TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes)	20
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	24
	26
MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes,	20
combiner, ierror)	28
TYPE(MPI_Datatype), INTENT(IN) :: datatype	29
<pre>INTEGER, INTENT(OUT) :: num_integers, num_addresses, num_datatypes,</pre>	30
	31
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	32
MPI_Type_get_envelope(datatype, num_integers, num_addresses,	33
num_large_counts, num_datatypes, combiner, ierror)	34
TYPE(MPI_Datatype), INTENT(IN) :: datatype	35
INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: num_integers,	36
num_addresses, num_large_counts, num_datatypes	37
INTEGER, INTENT(OUT) :: combiner	38
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	39
	40
MPI_Type_get_extent(datatype, lb, extent, ierror)	41
TYPE(MPI_Datatype), INTENT(IN) :: datatype	42
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: lb, extent	43
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	44
MPI_Type_get_extent(datatype, lb, extent, ierror)	45
TYPE(MPI_Datatype), INTENT(IN) :: datatype	46
INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: lb, extent	47
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	48

```
1
 MPI_Type_get_extent_x(datatype, lb, extent, ierror)
\mathbf{2}
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
3
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: lb, extent
4
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
 MPI_Type_get_true_extent(datatype, true_lb, true_extent, ierror)
6
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
7
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: true_lb, true_extent
8
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10
 MPI_Type_get_true_extent(datatype, true_lb, true_extent, ierror)
11
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
12
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: true_lb, true_extent
13
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
 MPI_Type_get_true_extent_x(datatype, true_lb, true_extent, ierror)
15
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
16
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: true_lb, true_extent
17
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
 MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements,
20
 oldtype, newtype, ierror)
21
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count,
22
 array_of_blocklengths(count), array_of_displacements(count)
23
 TYPE(MPI_Datatype), INTENT(IN) :: oldtype
24
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
25
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
 MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements,
27
 oldtype, newtype, ierror)
28
 INTEGER, INTENT(IN) :: count, array_of_blocklengths(count),
29
 array_of_displacements(count)
30
 TYPE(MPI_Datatype), INTENT(IN) :: oldtype
31
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
32
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
 MPI_Type_size(datatype, size, ierror)
35
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
36
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
37
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
 MPI_Type_size(datatype, size, ierror)
39
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
40
 INTEGER, INTENT(OUT) :: size
41
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
 MPI_Type_size_x(datatype, size, ierror)
44
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
45
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
46
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
 MPI_Type_vector(count, blocklength, stride, oldtype, newtype, ierror)
48
```

```
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count, blocklength, stride
 2
 TYPE(MPI_Datatype), INTENT(IN) :: oldtype
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 5
MPI_Type_vector(count, blocklength, stride, oldtype, newtype, ierror)
 6
 INTEGER, INTENT(IN) :: count, blocklength, stride
 TYPE(MPI_Datatype), INTENT(IN) :: oldtype
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
 9
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 10
 11
MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm,
 12
 ierror)
 13
 TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
 14
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: insize, outcount
 15
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(INOUT) :: position
 16
 TYPE(*), DIMENSION(..) :: outbuf
 17
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 18
 TYPE(MPI_Comm), INTENT(IN) :: comm
 19
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 20
MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm,
 21
 ierror)
 22
 TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
 23
 INTEGER, INTENT(IN) :: insize, outcount
 24
 INTEGER, INTENT(INOUT) :: position
 25
 TYPE(*), DIMENSION(..) :: outbuf
 26
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 27
 TYPE(MPI_Comm), INTENT(IN) :: comm
 28
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 29
 30
MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outcount,
 31
 datatype, ierror)
 32
 CHARACTER(LEN=*), INTENT(IN) :: datarep
 33
 TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
 34
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: insize
 35
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position
 36
 TYPE(*), DIMENSION(..) :: outbuf
 37
 INTEGER, INTENT(IN) :: outcount
 38
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 39
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 40
MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outcount,
 41
 datatype, ierror)
 42
 CHARACTER(LEN=*), INTENT(IN) :: datarep
 43
 TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
 44
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: insize, outcount
 45
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(INOUT) :: position
 46
 TYPE(*), DIMENSION(..) :: outbuf
 47
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 48
```

```
1
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
2
3
 A.4.4 Collective Communication Fortran 2008 Bindings
4
5
 MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
6
 comm, ierror)
7
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
8
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
9
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
10
 TYPE(*), DIMENSION(...) :: recvbuf
11
 TYPE(MPI_Comm), INTENT(IN) :: comm
12
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
 MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
14
 comm, ierror)
15
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
16
 INTEGER, INTENT(IN) :: sendcount, recvcount
17
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
18
 TYPE(*), DIMENSION(...) :: recvbuf
19
 TYPE(MPI_Comm), INTENT(IN) :: comm
20
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
 MPI_Allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
23
 recvtype, comm, info, request, ierror)
24
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
25
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
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
 MPI_Allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
33
 recvtype, comm, info, request, ierror)
34
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
35
 INTEGER, INTENT(IN) :: sendcount, recvcount
36
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
37
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
38
 TYPE(MPI_Comm), INTENT(IN) :: comm
39
 TYPE(MPI_Info), INTENT(IN) :: info
40
 TYPE(MPI_Request), INTENT(OUT) :: request
41
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
 MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
44
 recvtype, comm, ierror)
45
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
46
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcounts(*)
47
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
48
```

```
1
 TYPE(*), DIMENSION(..) :: recvbuf
 2
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
 TYPE(MPI_Comm), INTENT(IN) :: comm
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
 6
 recvtype, comm, ierror)
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
 INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
 9
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 10
 TYPE(*), DIMENSION(...) :: recvbuf
 11
 TYPE(MPI_Comm), INTENT(IN) :: comm
 12
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 13
MPI_Allgatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
 14
 15
 displs, recvtype, comm, info, request, ierror)
 16
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
 17
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
 18
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 19
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
 20
 21
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
 22
 TYPE(MPI_Comm), INTENT(IN) :: comm
 TYPE(MPI_Info), INTENT(IN) :: info
 23
 24
 TYPE(MPI_Request), INTENT(OUT) :: request
 25
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 26
MPI_Allgatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
 27
 displs, recvtype, comm, info, request, ierror)
 28
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
 29
 INTEGER, INTENT(IN) :: sendcount
 30
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 31
 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
 32
 INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
 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_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror)
 39
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
 40
 TYPE(*), DIMENSION(..) :: recvbuf
 41
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
 42
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 43
 TYPE(MPI_Op), INTENT(IN) :: op
 44
 TYPE(MPI_Comm), INTENT(IN) :: comm
 45
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 46
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror)
 47
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
 48
```

```
1
 TYPE(*), DIMENSION(...) :: recvbuf
2
 INTEGER, INTENT(IN) :: count
3
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
4
 TYPE(MPI_Op), INTENT(IN) :: op
5
 TYPE(MPI_Comm), INTENT(IN) :: comm
6
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
 MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info,
8
 request, ierror)
9
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
10
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
11
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
12
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
13
 TYPE(MPI_Op), INTENT(IN) :: op
14
 TYPE(MPI_Comm), INTENT(IN) :: comm
15
 TYPE(MPI_Info), INTENT(IN) :: info
16
 TYPE(MPI_Request), INTENT(OUT) :: request
17
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
 MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info,
20
 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_Info), INTENT(IN) :: info
28
 TYPE(MPI_Request), INTENT(OUT) :: request
29
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
 MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
31
 comm, ierror)
32
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
33
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
34
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
35
 TYPE(*), DIMENSION(..) :: recvbuf
36
 TYPE(MPI_Comm), INTENT(IN) :: comm
37
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
 MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
40
 comm, ierror)
41
 TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
42
 INTEGER, INTENT(IN) :: sendcount, recvcount
43
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
44
 TYPE(*), DIMENSION(...) :: recvbuf
45
 TYPE(MPI_Comm), INTENT(IN) :: comm
46
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
```

```
1
MPI_Alltoall_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
 2
 recvtype, comm, info, request, ierror)
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
 4
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 5
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 TYPE(MPI_Comm), INTENT(IN) :: comm
 TYPE(MPI_Info), INTENT(IN) :: info
 TYPE(MPI_Request), INTENT(OUT) :: request
 10
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 11
MPI_Alltoall_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
 12
 recvtype, comm, info, request, ierror)
 13
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
 14
 INTEGER, INTENT(IN) :: sendcount, recvcount
 15
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 16
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 17
 TYPE(MPI_Comm), INTENT(IN) :: comm
 18
 TYPE(MPI_Info), INTENT(IN) :: info
 19
 TYPE(MPI_Request), INTENT(OUT) :: request
 20
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 21
MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
 22
 23
 rdispls, recvtype, comm, ierror)
 24
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
 25
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*),
 26
 recvcounts(*)
 27
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
 28
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 29
 TYPE(*), DIMENSION(..) :: recvbuf
 TYPE(MPI_Comm), INTENT(IN) :: comm
 30
 31
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 32
MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
 33
 rdispls, recvtype, comm, ierror)
 34
 TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
 35
 INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
 36
 rdispls(*)
 37
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 38
 TYPE(*), DIMENSION(...) :: recvbuf
 39
 TYPE(MPI_Comm), INTENT(IN) :: comm
 40
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 41
 42
MPI_Alltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
 43
 recvcounts, rdispls, recvtype, comm, info, request, ierror)
 44
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
 45
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
 46
 sendcounts(*), recvcounts(*)
 47
```

48

```
1
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
2
 rdispls(*)
3
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
4
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
5
 TYPE(MPI_Comm), INTENT(IN) :: comm
6
 TYPE(MPI_Info), INTENT(IN) :: info
7
 TYPE(MPI_Request), INTENT(OUT) :: request
8
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
 MPI_Alltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
10
 recvcounts, rdispls, recvtype, comm, info, request, ierror)
11
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
12
 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
13
 recvcounts(*), rdispls(*)
14
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
15
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
16
 TYPE(MPI_Comm), INTENT(IN) :: comm
17
 TYPE(MPI_Info), INTENT(IN) :: info
18
 TYPE(MPI_Request), INTENT(OUT) :: request
19
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
 MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
22
 rdispls, recvtypes, comm, ierror)
23
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
24
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*),
25
 recvcounts(*)
26
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
27
 TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
28
 TYPE(*), DIMENSION(..) :: recvbuf
29
 TYPE(MPI_Comm), INTENT(IN) :: comm
30
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
 MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
32
 rdispls, recvtypes, comm, ierror)
33
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
34
 INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
35
 rdispls(*)
36
 TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
37
 TYPE(*), DIMENSION(..) :: recvbuf
38
 TYPE(MPI_Comm), INTENT(IN) :: comm
39
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
 MPI_Alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
42
 recvcounts, rdispls, recvtypes, comm, info, request, ierror)
43
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
44
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
45
 sendcounts(*), recvcounts(*)
46
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
47
 rdispls(*)
48
```

```
1
 TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
 2
 recvtypes(*)
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 TYPE(MPI_Comm), INTENT(IN) :: comm
 TYPE(MPI_Info), INTENT(IN) :: info
 5
 6
 TYPE(MPI_Request), INTENT(OUT) :: request
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
 recvcounts, rdispls, recvtypes, comm, info, request, ierror)
 10
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
 11
 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
 12
 recvcounts(*), rdispls(*)
 13
 TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
 14
 recvtypes(*)
 15
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 16
 TYPE(MPI_Comm), INTENT(IN) :: comm
 17
 TYPE(MPI_Info), INTENT(IN) :: info
 18
 TYPE(MPI_Request), INTENT(OUT) :: request
 19
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 20
 21
MPI_Barrier(comm, ierror)
 22
 TYPE(MPI_Comm), INTENT(IN) :: comm
 23
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 24
MPI_Barrier_init(comm, info, request, ierror)
 25
 TYPE(MPI_Comm), INTENT(IN) :: comm
 26
 TYPE(MPI_Info), INTENT(IN) :: info
 27
 TYPE(MPI_Request), INTENT(OUT) :: request
 28
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 29
 30
MPI_Bcast(buffer, count, datatype, root, comm, ierror)
 31
 TYPE(*), DIMENSION(..) :: buffer
 32
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
 33
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 34
 INTEGER, INTENT(IN) :: root
 35
 TYPE(MPI_Comm), INTENT(IN) :: comm
 36
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 37
MPI_Bcast(buffer, count, datatype, root, comm, ierror)
 38
 TYPE(*), DIMENSION(..) :: buffer
 39
 INTEGER, INTENT(IN) :: count, root
 40
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 41
 TYPE(MPI_Comm), INTENT(IN) :: comm
 42
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 43
 44
MPI_Bcast_init(buffer, count, datatype, root, comm, info, request, ierror)
 45
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
 46
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
 47
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 48
```

```
1
 INTEGER, INTENT(IN) :: root
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
 MPI_Bcast_init(buffer, count, datatype, root, comm, info, request, ierror)
7
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
8
 INTEGER, INTENT(IN) :: count, root
9
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
10
 TYPE(MPI_Comm), INTENT(IN) :: comm
11
 TYPE(MPI_Info), INTENT(IN) :: info
12
 TYPE(MPI_Request), INTENT(OUT) :: request
13
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
 MPI_Exscan(sendbuf, recvbuf, count, datatype, op, comm,
 ierror)
16
 TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
17
 TYPE(*), DIMENSION(...) :: recvbuf
18
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
19
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
20
 TYPE(MPI_Op), INTENT(IN) :: op
21
 TYPE(MPI_Comm), INTENT(IN) :: comm
22
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
 MPI_Exscan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
24
 TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
25
 TYPE(*), DIMENSION(..) :: recvbuf
26
 INTEGER, INTENT(IN) :: count
27
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
28
 TYPE(MPI_Op), INTENT(IN) :: op
29
 TYPE(MPI_Comm), INTENT(IN) :: comm
30
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32
 MPI_Exscan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
33
 ierror)
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_Info), INTENT(IN) :: info
41
 TYPE(MPI_Request), INTENT(OUT) :: request
42
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
 MPI_Exscan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
44
 ierror)
45
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
46
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
47
 INTEGER, INTENT(IN) :: count
48
```

```
1
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 2
 TYPE(MPI_Op), INTENT(IN) :: op
 TYPE(MPI_Comm), INTENT(IN) :: comm
 TYPE(MPI_Info), INTENT(IN) :: info
 TYPE(MPI_Request), INTENT(OUT) :: request
 5
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 6
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
 root, comm, ierror)
 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
 INTEGER, INTENT(IN) :: root
 14
 TYPE(MPI_Comm), INTENT(IN) :: comm
 15
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 16
 17
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
 18
 root, comm, ierror)
 19
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
 INTEGER, INTENT(IN) :: sendcount, recvcount, root
 20
 21
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 22
 TYPE(*), DIMENSION(..) :: recvbuf
 23
 TYPE(MPI_Comm), INTENT(IN) :: comm
 24
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 25
MPI_Gather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
 26
 root, comm, info, request, ierror)
 27
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
 28
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
 29
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 30
 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
 31
 INTEGER, INTENT(IN) :: root
 32
 TYPE(MPI_Comm), INTENT(IN) :: comm
 33
 TYPE(MPI_Info), INTENT(IN) :: info
 34
 TYPE(MPI_Request), INTENT(OUT) :: request
 35
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 36
 37
MPI_Gather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
 38
 root, comm, info, request, ierror)
 39
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
 40
 INTEGER, INTENT(IN) :: sendcount, recvcount, root
 41
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 42
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 43
 TYPE(MPI_Comm), INTENT(IN) :: comm
 44
 TYPE(MPI_Info), INTENT(IN) :: info
 45
 TYPE(MPI_Request), INTENT(OUT) :: request
 46
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 47
 48
```

```
1
 MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
\mathbf{2}
 recvtype, root, comm, ierror)
3
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
4
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcounts(*)
5
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
6
 TYPE(*), DIMENSION(..) :: recvbuf
7
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
8
 INTEGER, INTENT(IN) :: root
9
 TYPE(MPI_Comm), INTENT(IN) :: comm
10
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
 MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
12
 recvtype, root, comm, ierror)
13
 TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
14
 INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*), root
15
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
16
 TYPE(*), DIMENSION(..) :: recvbuf
17
 TYPE(MPI_Comm), INTENT(IN) :: comm
18
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20
 MPI_Gatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
21
 recvtype, root, comm, info, request, ierror)
22
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
23
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
24
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
25
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
26
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
27
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
28
 INTEGER, INTENT(IN) :: root
29
 TYPE(MPI_Comm), INTENT(IN) :: comm
30
 TYPE(MPI_Info), INTENT(IN) :: info
31
 TYPE(MPI_Request), INTENT(OUT) :: request
32
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
 MPI_Gatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
34
 recvtype, root, comm, info, request, ierror)
35
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
36
 INTEGER, INTENT(IN) :: sendcount, root
37
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
38
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
39
 INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
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_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
46
 comm, request, ierror)
47
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
48
```

```
1
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
 2
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 TYPE(MPI_Comm), INTENT(IN) :: comm
 TYPE(MPI_Request), INTENT(OUT) :: request
 5
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 6
MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
 comm, request, ierror)
 9
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
 10
 INTEGER, INTENT(IN) :: sendcount, recvcount
 11
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 12
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 13
 TYPE(MPI_Comm), INTENT(IN) :: comm
 14
 TYPE(MPI_Request), INTENT(OUT) :: request
 15
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 16
 17
MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
 18
 recvtype, comm, request, ierror)
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
 19
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
 20
 21
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 22
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 23
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
 ^{24}
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
 25
 TYPE(MPI_Comm), INTENT(IN) :: comm
 26
 TYPE(MPI_Request), INTENT(OUT) :: request
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 27
 28
MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
 29
 recvtype, comm, request, ierror)
 30
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
 31
 INTEGER, INTENT(IN) :: sendcount
 32
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 33
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 34
 INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
 35
 TYPE(MPI_Comm), INTENT(IN) :: comm
 36
 TYPE(MPI_Request), INTENT(OUT) :: request
 37
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 38
 39
MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request,
 40
 ierror)
 41
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
 42
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 43
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
 44
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 45
 TYPE(MPI_Op), INTENT(IN) :: op
 46
 TYPE(MPI_Comm), INTENT(IN) :: comm
 47
 TYPE(MPI_Request), INTENT(OUT) :: request
 48
```

```
1
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
 MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request,
3
 ierror)
4
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
5
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
6
 INTEGER, INTENT(IN) :: count
7
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
8
 TYPE(MPI_Op), INTENT(IN) :: op
9
 TYPE(MPI_Comm), INTENT(IN) :: comm
10
 TYPE(MPI_Request), INTENT(OUT) :: request
11
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13
 MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
14
 comm, request, ierror)
15
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
16
17
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
18
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
19
 TYPE(MPI_Comm), INTENT(IN) :: comm
20
 TYPE(MPI_Request), INTENT(OUT) :: request
21
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
 MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
23
 comm, request, ierror)
24
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
25
 INTEGER, INTENT(IN) :: sendcount, recvcount
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)
34
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
35
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
36
 sendcounts(*), 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
 MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
45
 rdispls, recvtype, comm, request, ierror)
46
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
47
48
```

```
INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
 1
 recvcounts(*), rdispls(*)
 2
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 4
 TYPE(MPI_Comm), INTENT(IN) :: comm
 5
 TYPE(MPI_Request), INTENT(OUT) :: request
 6
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
 9
 recvcounts, rdispls, recvtypes, comm, request, ierror)
 10
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
 11
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
 12
 sendcounts(*), recvcounts(*)
 13
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
 14
 rdispls(*)
 15
 TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
 16
 recvtypes(*)
 17
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 18
 TYPE(MPI_Comm), INTENT(IN) :: comm
 19
 TYPE(MPI_Request), INTENT(OUT) :: request
 20
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 21
 22
MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
 23
 recvcounts, rdispls, recvtypes, comm, request, ierror)
 24
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
 25
 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
 26
 recvcounts(*), rdispls(*)
 TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
 27
 28
 recvtypes(*)
 29
 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
 TYPE(MPI_Comm), INTENT(IN) :: comm
 30
 31
 TYPE(MPI_Request), INTENT(OUT) :: request
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 32
 33
MPI_Ibarrier(comm, request, ierror)
 34
 TYPE(MPI_Comm), INTENT(IN) :: comm
 35
 TYPE(MPI_Request), INTENT(OUT) :: request
 36
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 37
 38
MPI_Ibcast(buffer, count, datatype, root, comm, request, ierror)
 39
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
 40
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
 41
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 42
 INTEGER, INTENT(IN) :: root
 43
 TYPE(MPI_Comm), INTENT(IN) :: comm
 44
 TYPE(MPI_Request), INTENT(OUT) :: request
 45
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 46
MPI_Ibcast(buffer, count, datatype, root, comm, request, ierror)
 47
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
 48
```

```
1
 INTEGER, INTENT(IN) :: count, root
2
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
3
 TYPE(MPI_Comm), INTENT(IN) :: comm
4
 TYPE(MPI_Request), INTENT(OUT) :: request
5
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
 MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
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
15
16
 MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, 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_Request), INTENT(OUT) :: request
24
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
 MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
26
 root, comm, request, ierror)
27
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
28
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
29
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
30
 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
31
 INTEGER, INTENT(IN) :: root
32
 TYPE(MPI_Comm), INTENT(IN) :: comm
33
 TYPE(MPI_Request), INTENT(OUT) :: request
34
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
 MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
37
 root, comm, request, ierror)
38
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
39
 INTEGER, INTENT(IN) :: sendcount, recvcount, root
40
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
41
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
42
 TYPE(MPI_Comm), INTENT(IN) :: comm
43
 TYPE(MPI_Request), INTENT(OUT) :: request
44
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
 MPI_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
46
 recvtype, root, comm, request, ierror)
47
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
48
```

1 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount 2 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 3 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*) 4 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*) 56 INTEGER, INTENT(IN) :: root TYPE(MPI_Comm), INTENT(IN) :: comm 7 TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror 9 10 MPI_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, 11 recvtype, root, comm, request, ierror) 12TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 13 INTEGER, INTENT(IN) :: sendcount, root 14TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 15TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 16INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*) 17TYPE(MPI_Comm), INTENT(IN) :: comm 18 TYPE(MPI_Request), INTENT(OUT) :: request 19 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2021MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request, 22 ierror) 23TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 24TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf 25INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count 26TYPE(MPI_Datatype), INTENT(IN) :: datatype 27TYPE(MPI_Op), INTENT(IN) :: op 28INTEGER, INTENT(IN) :: root 29 TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request 30 31INTEGER, OPTIONAL, INTENT(OUT) :: ierror 32 MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request, 33 ierror) 34 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 35 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 36 INTEGER, INTENT(IN) :: count, root 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 4243 MPI_Ireduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm, 44request, ierror) 45TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 46TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 47INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*) 48

```
1
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
2
 TYPE(MPI_Op), INTENT(IN) :: op
3
 TYPE(MPI_Comm), INTENT(IN) :: comm
4
 TYPE(MPI_Request), INTENT(OUT) :: request
5
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
 MPI_Ireduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
7
 request, ierror)
8
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
9
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
10
 INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
11
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
12
 TYPE(MPI_Op), INTENT(IN) :: op
13
 TYPE(MPI_Comm), INTENT(IN) :: comm
14
 TYPE(MPI_Request), INTENT(OUT) :: request
15
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
 MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
18
 request, ierror)
19
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
20
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
21
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
22
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
23
 TYPE(MPI_Op), INTENT(IN) :: op
24
 TYPE(MPI_Comm), INTENT(IN) :: comm
25
 TYPE(MPI_Request), INTENT(OUT) :: request
26
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
 MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
28
 request, ierror)
29
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
30
 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
31
 INTEGER, INTENT(IN) :: recvcount
32
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
33
 TYPE(MPI_Op), INTENT(IN) :: op
34
 TYPE(MPI_Comm), INTENT(IN) :: comm
35
 TYPE(MPI_Request), INTENT(OUT) :: request
36
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
 MPI_Iscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
39
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
40
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
41
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
42
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
43
 TYPE(MPI_Op), INTENT(IN) :: op
44
 TYPE(MPI_Comm), INTENT(IN) :: comm
45
 TYPE(MPI_Request), INTENT(OUT) :: request
46
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
 MPI_Iscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
48
```

```
TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
 1
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 2
 INTEGER, INTENT(IN) :: count
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 TYPE(MPI_Op), INTENT(IN) :: op
 5
 TYPE(MPI_Comm), INTENT(IN) :: comm
 6
 TYPE(MPI_Request), INTENT(OUT) :: request
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
 10
 root, comm, request, ierror)
 11
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
 12
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
 13
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 14
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 15
 INTEGER, INTENT(IN) :: root
 16
 TYPE(MPI_Comm), INTENT(IN) :: comm
 17
 TYPE(MPI_Request), INTENT(OUT) :: request
 18
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 19
MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
 20
 21
 root, comm, 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
 TYPE(MPI_Request), INTENT(OUT) :: request
 27
 28
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 29
MPI_Iscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
 30
 recvtype, root, comm, request, ierror)
 31
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
 32
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*)
 33
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
 34
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 35
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 36
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
 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
 42
MPI_Iscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
 43
 recvtype, root, comm, request, ierror)
 44
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
 45
 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
 46
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 47
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 48
```

```
1
 INTEGER, INTENT(IN) :: recvcount, root
2
 TYPE(MPI_Comm), INTENT(IN) :: comm
3
 TYPE(MPI_Request), INTENT(OUT) :: request
4
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
 MPI_Op_commutative(op, commute, ierror)
6
 TYPE(MPI_Op), INTENT(IN) :: op
7
 LOGICAL, INTENT(OUT) :: commute
8
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10
 MPI_Op_create(user_fn, commute, op, ierror)
11
 PROCEDURE(MPI_User_function), INTENT(IN) :: user_fn
12
 LOGICAL, INTENT(IN) :: commute
13
 TYPE(MPI_Op), INTENT(OUT) :: op
14
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
 MPI_Op_create_c(user_fn, commute, op, ierror)
16
 PROCEDURE(MPI_User_function_c), INTENT(IN) :: user_fn
17
 LOGICAL, INTENT(IN) :: commute
18
 TYPE(MPI_Op), INTENT(OUT) :: op
19
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
 MPI_Op_free(op, ierror)
22
 TYPE(MPI_Op), INTENT(INOUT) :: op
23
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
 MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
25
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
26
 TYPE(*), DIMENSION(..) :: recvbuf
27
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
28
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
 TYPE(MPI_Op), INTENT(IN) :: op
30
 INTEGER, INTENT(IN) :: root
31
 TYPE(MPI_Comm), INTENT(IN) :: comm
32
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
 MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
35
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
36
 TYPE(*), DIMENSION(...) :: recvbuf
37
 INTEGER, INTENT(IN) :: count, root
38
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
39
 TYPE(MPI_Op), INTENT(IN) :: op
40
 TYPE(MPI_Comm), INTENT(IN) :: comm
41
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
 MPI_Reduce_init(sendbuf, recvbuf, count, datatype, op, root, comm, info,
43
 request, ierror)
44
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
45
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
46
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
47
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
48
```

```
1
 TYPE(MPI_Op), INTENT(IN) :: op
 2
 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_Reduce_init(sendbuf, recvbuf, count, datatype, op, root, comm, info,
 request, ierror)
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
 10
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 11
 INTEGER, INTENT(IN) :: count, root
 12
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 13
 TYPE(MPI_Op), INTENT(IN) :: op
 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_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror)
 TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
 20
 21
 TYPE(*), DIMENSION(..) :: inoutbuf
 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, OPTIONAL, INTENT(OUT) :: ierror
 26
MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror)
 27
 TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
 28
 TYPE(*), DIMENSION(..) :: inoutbuf
 29
 INTEGER, INTENT(IN) :: count
 30
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 31
 TYPE(MPI_Op), INTENT(IN) :: op
 32
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 33
 34
MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
 35
 ierror)
 36
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
 37
 TYPE(*), DIMENSION(..) :: recvbuf
 38
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcounts(*)
 39
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 40
 TYPE(MPI_Op), INTENT(IN) :: op
 41
 TYPE(MPI_Comm), INTENT(IN) :: comm
 42
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 43
MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
 44
 ierror)
 45
 TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
 46
 TYPE(*), DIMENSION(..) :: recvbuf
 47
 INTEGER, INTENT(IN) :: recvcounts(*)
 48
```

```
1
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
2
 TYPE(MPI_Op), INTENT(IN) :: op
3
 TYPE(MPI_Comm), INTENT(IN) :: comm
4
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
 MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
6
 ierror)
7
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
8
 TYPE(*), DIMENSION(...) :: recvbuf
9
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
10
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
11
 TYPE(MPI_Op), INTENT(IN) :: op
12
 TYPE(MPI_Comm), INTENT(IN) :: comm
13
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
 MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
16
 ierror)
17
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
18
 TYPE(*), DIMENSION(...) :: recvbuf
19
 INTEGER, INTENT(IN) :: recvcount
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
 MPI_Reduce_scatter_block_init(sendbuf, recvbuf, recvcount, datatype, op,
25
 comm, info, request, ierror)
26
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
27
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
28
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
29
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
30
 TYPE(MPI_Op), INTENT(IN) :: op
31
 TYPE(MPI_Comm), INTENT(IN) :: comm
32
 TYPE(MPI_Info), INTENT(IN) :: info
33
 TYPE(MPI_Request), INTENT(OUT) :: request
34
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
 MPI_Reduce_scatter_block_init(sendbuf, recvbuf, recvcount, datatype, op,
37
 comm, info, request, ierror)
38
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
39
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
40
 INTEGER, INTENT(IN) :: recvcount
41
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
42
 TYPE(MPI_Op), INTENT(IN) :: op
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
```

```
MPI_Reduce_scatter_init(sendbuf, recvbuf, recvcounts, datatype, op, comm,
 2
 info, request, ierror)
 3
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 4
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
 5
 6
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 TYPE(MPI_Op), INTENT(IN) :: op
 7
 TYPE(MPI_Comm), INTENT(IN) :: comm
 TYPE(MPI_Info), INTENT(IN) :: info
 9
 10
 TYPE(MPI_Request), INTENT(OUT) :: request
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 11
 12
MPI_Reduce_scatter_init(sendbuf, recvbuf, recvcounts, datatype, op, comm,
 13
 info, request, ierror)
 14
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
 15
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 16
 INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
 17
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 18
 TYPE(MPI_Op), INTENT(IN) :: op
 19
 TYPE(MPI_Comm), INTENT(IN) :: comm
 20
 TYPE(MPI_Info), INTENT(IN) :: info
 21
 TYPE(MPI_Request), INTENT(OUT) :: request
 22
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 23
 24
MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
 TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
 25
 26
 TYPE(*), DIMENSION(..) :: recvbuf
 27
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
 28
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 29
 TYPE(MPI_Op), INTENT(IN) :: op
 TYPE(MPI_Comm), INTENT(IN) :: comm
 30
 31
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 32
MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
 33
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
 34
 TYPE(*), DIMENSION(..) :: recvbuf
 35
 INTEGER, INTENT(IN) :: count
 36
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 37
 TYPE(MPI_Op), INTENT(IN) :: op
 38
 TYPE(MPI_Comm), INTENT(IN) :: comm
 39
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 40
 41
MPI_Scan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
 42
 ierror)
 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
```

```
1
 TYPE(MPI_Comm), INTENT(IN) :: comm
2
 TYPE(MPI_Info), INTENT(IN) :: info
3
 TYPE(MPI_Request), INTENT(OUT) :: request
4
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
 MPI_Scan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
6
 ierror)
7
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
8
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
9
 INTEGER, 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_Info), INTENT(IN) :: info
14
 TYPE(MPI_Request), INTENT(OUT) :: request
15
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
 MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
18
 root, comm, ierror)
19
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
20
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
21
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
22
 TYPE(*), DIMENSION(..) :: recvbuf
23
 INTEGER, INTENT(IN) :: root
24
 TYPE(MPI_Comm), INTENT(IN) :: comm
25
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
 MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
27
 root, comm, ierror)
28
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
29
 INTEGER, INTENT(IN) :: sendcount, recvcount, root
30
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
31
 TYPE(*), DIMENSION(..) :: recvbuf
32
 TYPE(MPI_Comm), INTENT(IN) :: comm
33
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
 MPI_Scatter_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
36
 recvtype, root, comm, info, request, ierror)
37
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
38
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
39
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
40
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
41
 INTEGER, INTENT(IN) :: root
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
 MPI_Scatter_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
47
 recvtype, root, comm, info, request, ierror)
48
```

TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 1 2 INTEGER, INTENT(IN) :: sendcount, recvcount, root TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf TYPE(MPI_Comm), INTENT(IN) :: comm 5 TYPE(MPI_Info), INTENT(IN) :: info 6 TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, 10 recvtype, root, comm, ierror) 11 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf 12INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*), recvcount 13 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*) 14TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 15TYPE(*), DIMENSION(..) :: recvbuf 16INTEGER, INTENT(IN) :: root 17TYPE(MPI_Comm), INTENT(IN) :: comm 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 19 MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, 2021recvtype, root, comm, ierror) 22 TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf 23INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root 24TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 25TYPE(*), DIMENSION(..) :: recvbuf 26TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2728 MPI_Scatterv_init(sendbuf, sendcounts, displs, sendtype, recvbuf, 29 recvcount, recvtype, root, comm, info, request, ierror) 30 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 31INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*) 32 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*) 33 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 34 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf 35 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount 36 INTEGER, INTENT(IN) :: root 37 TYPE(MPI_Comm), INTENT(IN) :: comm 38 TYPE(MPI_Info), INTENT(IN) :: info 39 TYPE(MPI_Request), INTENT(OUT) :: request 40 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 41 42MPI_Scatterv_init(sendbuf, sendcounts, displs, sendtype, recvbuf, 43 recvcount, recvtype, root, comm, info, request, ierror) 44TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 45INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*) 46TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 47TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 48

```
1
 INTEGER, INTENT(IN) :: recvcount, root
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
7
 A.4.5 Groups, Contexts, Communicators, and Caching Fortran 2008 Bindings
8
9
 MPI_COMM_DUP_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,
10
 attribute_val_out, flag, ierror)
11
 TYPE(MPI_Comm) :: oldcomm
12
 INTEGER :: comm_keyval, ierror
13
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
14
 attribute_val_out
15
 LOGICAL :: flag
16
 MPI_COMM_NULL_COPY_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,
17
 attribute_val_out, flag, ierror)
18
 TYPE(MPI_Comm) :: oldcomm
19
 INTEGER :: comm_keyval, ierror
20
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
21
 attribute_val_out
22
 LOGICAL :: flag
23
24
 MPI_COMM_NULL_DELETE_FN(comm, comm_keyval, attribute_val, extra_state,
25
 ierror)
26
 TYPE(MPI_Comm) :: comm
27
 INTEGER :: comm_keyval, ierror
28
 INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
29
 MPI_Comm_compare(comm1, comm2, result, ierror)
30
 TYPE(MPI_Comm), INTENT(IN) :: comm1, comm2
31
 INTEGER, INTENT(OUT) :: result
32
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
 MPI_Comm_create(comm, group, newcomm, ierror)
35
 TYPE(MPI_Comm), INTENT(IN) :: comm
36
 TYPE(MPI_Group), INTENT(IN) :: group
37
 TYPE(MPI_Comm), INTENT(OUT) :: newcomm
38
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
 MPI_Comm_create_from_group(group, stringtag, info, errhandler, newcomm,
40
41
 ierror)
 TYPE(MPI_Group), INTENT(IN) :: group
42
 CHARACTER(LEN=*), INTENT(IN) :: stringtag
43
 TYPE(MPI_Info), INTENT(IN) :: info
44
 TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
45
 TYPE(MPI_Comm), INTENT(OUT) :: newcomm
46
47
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

```
1
MPI_Comm_create_group(comm, group, tag, newcomm, ierror)
 2
 TYPE(MPI_Comm), INTENT(IN) :: comm
 TYPE(MPI_Group), INTENT(IN) :: group
 INTEGER, INTENT(IN) :: tag
 TYPE(MPI_Comm), INTENT(OUT) :: newcomm
 5
 6
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Comm_create_keyval(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval,
 8
 extra_state, ierror)
 9
 PROCEDURE(MPI_Comm_copy_attr_function), INTENT(IN) :: comm_copy_attr_fn
 10
 PROCEDURE(MPI_Comm_delete_attr_function), INTENT(IN) ::
 11
 comm_delete_attr_fn
 12
 INTEGER, INTENT(OUT) :: comm_keyval
 13
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
 14
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 15
 16
MPI_Comm_delete_attr(comm, comm_keyval, ierror)
 17
 TYPE(MPI_Comm), INTENT(IN) :: comm
 18
 INTEGER, INTENT(IN) :: comm_keyval
 19
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 20
MPI_Comm_dup(comm, newcomm, ierror)
 21
 TYPE(MPI_Comm), INTENT(IN) :: comm
 22
 TYPE(MPI_Comm), INTENT(OUT) :: newcomm
 23
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 24
 25
MPI_Comm_dup_with_info(comm, info, newcomm, ierror)
 26
 TYPE(MPI_Comm), INTENT(IN) :: comm
 27
 TYPE(MPI_Info), INTENT(IN) :: info
 28
 TYPE(MPI_Comm), INTENT(OUT) :: newcomm
 29
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 30
MPI_Comm_free(comm, ierror)
 31
 TYPE(MPI_Comm), INTENT(INOUT) :: comm
 32
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 33
 34
MPI_Comm_free_keyval(comm_keyval, ierror)
 35
 INTEGER, INTENT(INOUT) :: comm_keyval
 36
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 37
MPI_Comm_get_attr(comm, comm_keyval, attribute_val, flag, ierror)
 38
 TYPE(MPI_Comm), INTENT(IN) :: comm
 39
 INTEGER, INTENT(IN) :: comm_keyval
 40
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
 41
 LOGICAL, INTENT(OUT) :: flag
 42
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 43
 44
MPI_Comm_get_info(comm, info_used, ierror)
 45
 TYPE(MPI_Comm), INTENT(IN) :: comm
 46
 TYPE(MPI_Info), INTENT(OUT) :: info_used
 47
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 48
```

```
1
 MPI_Comm_get_name(comm, comm_name, resultlen, ierror)
\mathbf{2}
 TYPE(MPI_Comm), INTENT(IN) :: comm
3
 CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: comm_name
4
 INTEGER, INTENT(OUT) :: resultlen
5
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
 MPI_Comm_group(comm, group, ierror)
7
 TYPE(MPI_Comm), INTENT(IN) :: comm
8
 TYPE(MPI_Group), INTENT(OUT) :: group
9
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
 MPI_Comm_idup(comm, newcomm, request, ierror)
12
 TYPE(MPI_Comm), INTENT(IN) :: comm
13
 TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS :: newcomm
14
 TYPE(MPI_Request), INTENT(OUT) :: request
15
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
 MPI_Comm_idup_with_info(comm, info, newcomm, request, ierror)
17
 TYPE(MPI_Comm), INTENT(IN) :: comm
18
 TYPE(MPI_Info), INTENT(IN) :: info
19
 TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS :: newcomm
20
 TYPE(MPI_Request), INTENT(OUT) :: request
21
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
 MPI_Comm_rank(comm, rank, ierror)
24
 TYPE(MPI_Comm), INTENT(IN) :: comm
25
 INTEGER, INTENT(OUT) :: rank
26
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
 MPI_Comm_remote_group(comm, group, ierror)
28
 TYPE(MPI_Comm), INTENT(IN) :: comm
29
 TYPE(MPI_Group), INTENT(OUT) :: group
30
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32
 MPI_Comm_remote_size(comm, size, ierror)
33
 TYPE(MPI_Comm), INTENT(IN) :: comm
34
 INTEGER, INTENT(OUT) :: size
35
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
 MPI_Comm_set_attr(comm, comm_keyval, attribute_val, ierror)
37
 TYPE(MPI_Comm), INTENT(IN) :: comm
38
 INTEGER, INTENT(IN) :: comm_keyval
39
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
40
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
 MPI_Comm_set_info(comm, info, ierror)
43
 TYPE(MPI_Comm), INTENT(IN) :: comm
44
 TYPE(MPI_Info), INTENT(IN) :: info
45
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
 MPI_Comm_set_name(comm, comm_name, ierror)
47
 TYPE(MPI_Comm), INTENT(IN) :: comm
48
```

CHARACTER(LEN=*), INTENT(IN) :: comm_name INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1 2
	3
MPI_Comm_size(comm, size, ierror)	4
TYPE(MPI_Comm), INTENT(IN) :: comm	5
INTEGER, INTENT(OUT) :: size	6
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	7
MPI_Comm_split(comm, color, key, newcomm, ierror)	8
TYPE(MPI_Comm), INTENT(IN) :: comm	9
INTEGER, INTENT(IN) :: color, key	10
TYPE(MPI_Comm), INTENT(OUT) :: newcomm	11
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	12
MPI_Comm_split_type(comm, split_type, key, info, newcomm, ierror)	13
TYPE(MPI_Comm), INTENT(IN) :: comm	14
INTEGER, INTENT(IN) :: split_type, key	15 16
TYPE(MPI_Info), INTENT(IN) :: info	10
TYPE(MPI_Comm), INTENT(OUT) :: newcomm	18
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	19
NDT (Lame to st interv(same flag, issues)	20
<pre>MPI_Comm_test_inter(comm, flag, ierror)     TYPE(MPI_Comm), INTENT(IN) :: comm</pre>	21
LOGICAL, INTENT(OUT) :: flag	22
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	23
	24
<pre>MPI_Group_compare(group1, group2, result, ierror)</pre>	25
TYPE(MPI_Group), INTENT(IN) :: group1, group2	26
INTEGER, INTENT(OUT) :: result	27
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	28
MPI_Group_difference(group1, group2, newgroup, ierror)	29
TYPE(MPI_Group), INTENT(IN) :: group1, group2	30
TYPE(MPI_Group), INTENT(OUT) :: newgroup	31
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	32 33
MPI_Group_excl(group, n, ranks, newgroup, ierror)	34
TYPE(MPI_Group), INTENT(IN) :: group	35
INTEGER, INTENT(IN) :: n, ranks(n)	36
TYPE(MPI_Group), INTENT(OUT) :: newgroup	37
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	38
MPI_Group_free(group, ierror)	39
TYPE(MPI_Group), INTENT(INOUT) :: group	40
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	41
	42
<pre>MPI_Group_from_session_pset(session, pset_name, newgroup, ierror)</pre>	43
TYPE(MPI_Session), INTENT(IN) :: session	44
CHARACTER(LEN=*), INTENT(IN) :: pset_name	45
TYPE(MPI_Group), INTENT(OUT) :: newgroup	46
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	47
	48

```
1
 MPI_Group_incl(group, n, ranks, newgroup, ierror)
\mathbf{2}
 TYPE(MPI_Group), INTENT(IN) :: group
3
 INTEGER, INTENT(IN) :: n, ranks(n)
4
 TYPE(MPI_Group), INTENT(OUT) :: newgroup
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
6
 MPI_Group_intersection(group1, group2, newgroup, ierror)
7
 TYPE(MPI_Group), INTENT(IN) :: group1, group2
8
 TYPE(MPI_Group), INTENT(OUT) :: newgroup
9
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
 MPI_Group_range_excl(group, n, ranges, newgroup, ierror)
12
 TYPE(MPI_Group), INTENT(IN) :: group
13
 INTEGER, INTENT(IN) :: n, ranges(3, n)
14
 TYPE(MPI_Group), INTENT(OUT) :: newgroup
15
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
 MPI_Group_range_incl(group, n, ranges, newgroup, ierror)
17
 TYPE(MPI_Group), INTENT(IN) :: group
18
 INTEGER, INTENT(IN) :: n, ranges(3, n)
19
 TYPE(MPI_Group), INTENT(OUT) :: newgroup
20
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
 MPI_Group_rank(group, rank, ierror)
23
 TYPE(MPI_Group), INTENT(IN) :: group
24
 INTEGER, INTENT(OUT) :: rank
25
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
 MPI_Group_size(group, size, ierror)
27
 TYPE(MPI_Group), INTENT(IN) :: group
28
 INTEGER, INTENT(OUT) :: size
29
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31
 MPI_Group_translate_ranks(group1, n, ranks1, group2, ranks2, ierror)
32
 TYPE(MPI_Group), INTENT(IN) :: group1, group2
33
 INTEGER, INTENT(IN) :: n, ranks1(n)
34
 INTEGER, INTENT(OUT) :: ranks2(n)
35
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
 MPI_Group_union(group1, group2, newgroup, ierror)
37
 TYPE(MPI_Group), INTENT(IN) :: group1, group2
38
 TYPE(MPI_Group), INTENT(OUT) :: newgroup
39
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
 MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader,
42
 tag, newintercomm, ierror)
43
 TYPE(MPI_Comm), INTENT(IN) :: local_comm, peer_comm
44
 INTEGER, INTENT(IN) :: local_leader, remote_leader, tag
45
 TYPE(MPI_Comm), INTENT(OUT) :: newintercomm
46
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
```

```
1
MPI_Intercomm_create_from_groups(local_group, local_leader, remote_group,
 \mathbf{2}
 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
 10
 11
MPI_Intercomm_merge(intercomm, high, newintracomm, ierror)
 12
 TYPE(MPI_Comm), INTENT(IN) :: intercomm
 13
 LOGICAL, INTENT(IN) :: high
 14
 TYPE(MPI_Comm), INTENT(OUT) :: newintracomm
 15
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 16
 17
MPI_TYPE_DUP_FN(oldtype, type_keyval, extra_state, attribute_val_in,
 18
 attribute_val_out, flag, ierror)
 19
 TYPE(MPI_Datatype) :: oldtype
 20
 INTEGER :: type_keyval, ierror
 21
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
 22
 attribute_val_out
 23
 LOGICAL :: flag
 24
MPI_TYPE_NULL_COPY_FN(oldtype, type_keyval, extra_state, attribute_val_in,
 25
 attribute_val_out, flag, ierror)
 26
 TYPE(MPI_Datatype) :: oldtype
 27
 INTEGER :: type_keyval, ierror
 28
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
 29
 attribute_val_out
 30
 LOGICAL :: flag
 31
 32
MPI_TYPE_NULL_DELETE_FN(datatype, type_keyval, attribute_val, extra_state,
 33
 ierror)
 34
 TYPE(MPI_Datatype) :: datatype
 35
 INTEGER :: type_keyval
 36
 INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
 37
 INTEGER, INTENT(OUT) :: ierror
 38
MPI_Type_create_keyval(type_copy_attr_fn, type_delete_attr_fn, type_keyval,
 39
 extra_state, ierror)
 40
 PROCEDURE(MPI_Type_copy_attr_function), INTENT(IN) :: type_copy_attr_fn
 41
 PROCEDURE(MPI_Type_delete_attr_function), INTENT(IN) ::
 42
 type_delete_attr_fn
 43
 INTEGER, INTENT(OUT) :: type_keyval
 44
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
 45
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 46
 47
MPI_Type_delete_attr(datatype, type_keyval, ierror)
 48
```

```
1
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
2
 INTEGER, INTENT(IN) :: type_keyval
3
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
 MPI_Type_free_keyval(type_keyval, ierror)
5
 INTEGER, INTENT(INOUT) :: type_keyval
6
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
 MPI_Type_get_attr(datatype, type_keyval, attribute_val, flag, ierror)
9
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
10
 INTEGER, INTENT(IN) :: type_keyval
11
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
12
 LOGICAL, INTENT(OUT) :: flag
13
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
 MPI_Type_get_name(datatype, type_name, resultlen, ierror)
15
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
16
 CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: type_name
17
 INTEGER, INTENT(OUT) :: resultlen
18
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20
 MPI_Type_set_attr(datatype, type_keyval, attribute_val, ierror)
21
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
22
 INTEGER, INTENT(IN) :: type_keyval
23
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
24
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
 MPI_Type_set_name(datatype, type_name, ierror)
26
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
 CHARACTER(LEN=*), INTENT(IN) :: type_name
28
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
 MPI_WIN_DUP_FN(oldwin, win_keyval, extra_state, attribute_val_in,
31
 attribute_val_out, flag, ierror)
32
 TYPE(MPI_Win) :: oldwin
33
 INTEGER :: win_keyval, ierror
34
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
35
 attribute_val_out
36
 LOGICAL :: flag
37
 MPI_WIN_NULL_COPY_FN(oldwin, win_keyval, extra_state, attribute_val_in,
38
 attribute_val_out, flag, ierror)
39
 TYPE(MPI_Win) :: oldwin
40
 INTEGER :: win_keyval, ierror
41
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
42
 attribute_val_out
43
 LOGICAL :: flag
44
45
 MPI_WIN_NULL_DELETE_FN(win, win_keyval, attribute_val, extra_state, ierror)
46
 TYPE(MPI_Win) :: win
47
 INTEGER :: win_keyval, ierror
48
```

```
1
 INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
 2
MPI_Win_create_keyval(win_copy_attr_fn, win_delete_attr_fn, win_keyval,
 extra_state, ierror)
 PROCEDURE(MPI_Win_copy_attr_function), INTENT(IN) :: win_copy_attr_fn
 5
 PROCEDURE(MPI_Win_delete_attr_function), INTENT(IN) ::
 6
 win_delete_attr_fn
 INTEGER, INTENT(OUT) :: win_keyval
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 10
 11
MPI_Win_delete_attr(win, win_keyval, ierror)
 TYPE(MPI_Win), INTENT(IN) :: win
 12
 13
 INTEGER, INTENT(IN) :: win_keyval
 14
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 15
MPI_Win_free_keyval(win_keyval, ierror)
 16
 INTEGER, INTENT(INOUT) :: win_keyval
 17
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 18
 19
MPI_Win_get_attr(win, win_keyval, attribute_val, flag, ierror)
 20
 TYPE(MPI_Win), INTENT(IN) :: win
 21
 INTEGER, INTENT(IN) :: win_keyval
 22
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
 23
 LOGICAL, INTENT(OUT) :: flag
 24
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 25
MPI_Win_get_name(win, win_name, resultlen, ierror)
 26
 TYPE(MPI_Win), INTENT(IN) :: win
 27
 CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: win_name
 28
 INTEGER, INTENT(OUT) :: resultlen
 29
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 30
 31
MPI_Win_set_attr(win, win_keyval, attribute_val, ierror)
 32
 TYPE(MPI_Win), INTENT(IN) :: win
 33
 INTEGER, INTENT(IN) :: win_keyval
 34
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
 35
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 36
MPI_Win_set_name(win, win_name, ierror)
 37
 TYPE(MPI_Win), INTENT(IN) :: win
 38
 CHARACTER(LEN=*), INTENT(IN) :: win_name
 39
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 40
 41
 42
A.4.6 Process Topologies Fortran 2008 Bindings
 43
MPI_Cart_coords(comm, rank, maxdims, coords, ierror)
 44
 TYPE(MPI_Comm), INTENT(IN) :: comm
 45
 INTEGER, INTENT(IN) :: rank, maxdims
 46
 INTEGER, INTENT(OUT) :: coords(maxdims)
 47
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 48
```

```
1
 MPI_Cart_create(comm_old, ndims, dims, periods, reorder, comm_cart, ierror)
\mathbf{2}
 TYPE(MPI_Comm), INTENT(IN) :: comm_old
3
 INTEGER, INTENT(IN) :: ndims, dims(ndims)
4
 LOGICAL, INTENT(IN) :: periods(ndims), reorder
5
 TYPE(MPI_Comm), INTENT(OUT) :: comm_cart
6
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
 MPI_Cart_get(comm, maxdims, dims, periods, coords, ierror)
8
 TYPE(MPI_Comm), INTENT(IN) :: comm
9
 INTEGER, INTENT(IN) :: maxdims
10
 INTEGER, INTENT(OUT) :: dims(maxdims), coords(maxdims)
11
 LOGICAL, INTENT(OUT) :: periods(maxdims)
12
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
 MPI_Cart_map(comm, ndims, dims, periods, newrank, ierror)
15
 TYPE(MPI_Comm), INTENT(IN) :: comm
16
 INTEGER, INTENT(IN) :: ndims, dims(ndims)
17
 LOGICAL, INTENT(IN) :: periods(ndims)
18
 INTEGER, INTENT(OUT) :: newrank
19
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
 MPI_Cart_rank(comm, coords, rank, ierror)
21
 TYPE(MPI_Comm), INTENT(IN) :: comm
22
 INTEGER, INTENT(IN) :: coords(*)
23
 INTEGER, INTENT(OUT) :: rank
24
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26
 MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror)
27
 TYPE(MPI_Comm), INTENT(IN) :: comm
28
 INTEGER, INTENT(IN) :: direction, disp
29
 INTEGER, INTENT(OUT) :: rank_source, rank_dest
30
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
 MPI_Cart_sub(comm, remain_dims, newcomm, ierror)
32
 TYPE(MPI_Comm), INTENT(IN) :: comm
33
 LOGICAL, INTENT(IN) :: remain_dims(*)
34
 TYPE(MPI_Comm), INTENT(OUT) :: newcomm
35
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
 MPI_Cartdim_get(comm, ndims, ierror)
38
 TYPE(MPI_Comm), INTENT(IN) :: comm
39
 INTEGER, INTENT(OUT) :: ndims
40
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
 MPI_Dims_create(nnodes, ndims, dims, ierror)
42
 INTEGER, INTENT(IN) :: nnodes, ndims
43
 INTEGER, INTENT(INOUT) :: dims(ndims)
44
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
 MPI_Dist_graph_create(comm_old, n, sources, degrees, destinations, weights,
47
 info, reorder, comm_dist_graph, ierror)
48
```

```
1
 TYPE(MPI_Comm), INTENT(IN) :: comm_old
 2
 INTEGER, INTENT(IN) :: n, sources(n), degrees(n), destinations(*),
 weights(*)
 TYPE(MPI_Info), INTENT(IN) :: info
 LOGICAL, INTENT(IN) :: reorder
 5
 TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
 6
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Dist_graph_create_adjacent(comm_old, indegree, sources, sourceweights,
 9
 outdegree, destinations, destweights, info, reorder,
 10
 comm_dist_graph, ierror)
 11
 TYPE(MPI_Comm), INTENT(IN) :: comm_old
 12
 INTEGER, INTENT(IN) :: indegree, sources(indegree), sourceweights(*),
 13
 outdegree, destinations(outdegree), destweights(*)
 14
 TYPE(MPI_Info), INTENT(IN) :: info
 15
 LOGICAL, INTENT(IN) :: reorder
 16
 TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
 17
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 18
 19
MPI_Dist_graph_neighbors(comm, maxindegree, sources, sourceweights,
 maxoutdegree, destinations, destweights, ierror)
 20
 21
 TYPE(MPI_Comm), INTENT(IN) :: comm
 22
 INTEGER, INTENT(IN) :: maxindegree, maxoutdegree
 23
 INTEGER, INTENT(OUT) :: sources(maxindegree),
 24
 destinations(maxoutdegree)
 25
 INTEGER :: sourceweights(*), destweights(*)
 26
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 27
MPI_Dist_graph_neighbors_count(comm, indegree, outdegree, weighted, ierror)
 28
 TYPE(MPI_Comm), INTENT(IN) :: comm
 29
 INTEGER, INTENT(OUT) :: indegree, outdegree
 30
 LOGICAL, INTENT(OUT) :: weighted
 31
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 32
 33
MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph,
 34
 ierror)
 35
 TYPE(MPI_Comm), INTENT(IN) :: comm_old
 36
 INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
 37
 LOGICAL, INTENT(IN) :: reorder
 38
 TYPE(MPI_Comm), INTENT(OUT) :: comm_graph
 39
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 40
MPI_Graph_get(comm, maxindex, maxedges, index, edges, ierror)
 41
 TYPE(MPI_Comm), INTENT(IN) :: comm
 42
 INTEGER, INTENT(IN) :: maxindex, maxedges
 43
 INTEGER, INTENT(OUT) :: index(maxindex), edges(maxedges)
 44
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 45
 46
MPI_Graph_map(comm, nnodes, index, edges, newrank, ierror)
 47
 TYPE(MPI_Comm), INTENT(IN) :: comm
 48
```

```
1
 INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
2
 INTEGER, INTENT(OUT) :: newrank
3
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
 MPI_Graph_neighbors(comm, rank, maxneighbors, neighbors, ierror)
5
 TYPE(MPI_Comm), INTENT(IN) :: comm
6
 INTEGER, INTENT(IN) :: rank, maxneighbors
7
 INTEGER, INTENT(OUT) :: neighbors(maxneighbors)
8
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10
 MPI_Graph_neighbors_count(comm, rank, nneighbors, ierror)
11
 TYPE(MPI_Comm), INTENT(IN) :: comm
12
 INTEGER, INTENT(IN) :: rank
13
 INTEGER, INTENT(OUT) :: nneighbors
14
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
 MPI_Graphdims_get(comm, nnodes, nedges, ierror)
16
 TYPE(MPI_Comm), INTENT(IN) :: comm
17
 INTEGER, INTENT(OUT) :: nnodes, nedges
18
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20
 MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
21
 recvtype, comm, request, ierror)
22
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
23
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
24
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
25
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
26
 TYPE(MPI_Comm), INTENT(IN) :: comm
27
 TYPE(MPI_Request), INTENT(OUT) :: request
28
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
 MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
30
 recvtype, comm, request, ierror)
31
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
32
 INTEGER, INTENT(IN) :: sendcount, recvcount
33
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
34
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
35
 TYPE(MPI_Comm), INTENT(IN) :: comm
36
 TYPE(MPI_Request), INTENT(OUT) :: request
37
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
 MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
40
 displs, recvtype, comm, request, ierror)
41
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
42
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
43
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
44
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
45
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
46
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
47
 TYPE(MPI_Comm), INTENT(IN) :: comm
48
```

```
1
 TYPE(MPI_Request), INTENT(OUT) :: request
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 2
MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
 displs, recvtype, comm, request, ierror)
 5
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
 6
 INTEGER, INTENT(IN) :: sendcount
 7
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 9
 INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
 10
 TYPE(MPI_Comm), INTENT(IN) :: comm
 11
 TYPE(MPI_Request), INTENT(OUT) :: request
 12
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 13
 14
MPI_Ineighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
 15
 recvtype, comm, request, ierror)
 16
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
 17
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
 18
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 19
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 TYPE(MPI_Comm), INTENT(IN) :: comm
 20
 21
 TYPE(MPI_Request), INTENT(OUT) :: request
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 22
 23
MPI_Ineighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
 24
 recvtype, comm, request, ierror)
 25
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
 26
 INTEGER, 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_Request), INTENT(OUT) :: request
 31
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 32
 33
MPI_Ineighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
 34
 recvcounts, rdispls, recvtype, comm, request, ierror)
 35
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
 36
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
 37
 sendcounts(*), recvcounts(*)
 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_Request), INTENT(OUT) :: request
 44
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 45
MPI_Ineighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
 46
 recvcounts, rdispls, recvtype, comm, request, ierror)
 47
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
 48
```

```
1
 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
2
 recvcounts(*), rdispls(*)
3
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
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
 MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
9
 recvcounts, rdispls, recvtypes, comm, request, ierror)
10
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
11
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
12
 sendcounts(*), recvcounts(*)
13
 INTEGER(KIND=MPI ADDRESS KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
14
 rdispls(*)
15
 TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
16
 recvtypes(*)
17
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
18
 TYPE(MPI_Comm), INTENT(IN) :: comm
19
 TYPE(MPI_Request), INTENT(OUT) :: request
20
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
 MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
23
 recvcounts, rdispls, recvtypes, comm, request, ierror)
24
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
25
26
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
27
 rdispls(*)
28
 TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
29
 recvtypes(*)
30
 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
31
 TYPE(MPI_Comm), INTENT(IN) :: comm
32
 TYPE(MPI_Request), INTENT(OUT) :: request
33
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
 MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
35
 recvtype, comm, ierror)
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_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
44
 recvtype, comm, ierror)
45
 TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
46
 INTEGER, INTENT(IN) :: sendcount, recvcount
47
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
48
```

```
1
 TYPE(*), DIMENSION(..) :: recvbuf
 TYPE(MPI_Comm), INTENT(IN) :: comm
 2
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Neighbor_allgather_init(sendbuf, sendcount, sendtype, recvbuf,
 5
 recvcount, recvtype, comm, info, request, ierror)
 6
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 9
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 10
 TYPE(MPI_Comm), INTENT(IN) :: comm
 11
 TYPE(MPI_Info), INTENT(IN) :: info
 12
 TYPE(MPI_Request), INTENT(OUT) :: request
 13
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 14
 15
MPI_Neighbor_allgather_init(sendbuf, sendcount, sendtype, recvbuf,
 16
 recvcount, recvtype, comm, info, request, ierror)
 17
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
 18
 INTEGER, INTENT(IN) :: sendcount, recvcount
 19
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 20
 21
 TYPE(MPI_Comm), INTENT(IN) :: comm
 22
 TYPE(MPI_Info), INTENT(IN) :: info
 23
 TYPE(MPI_Request), INTENT(OUT) :: request
 24
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 25
MPI_Neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
 26
 displs, recvtype, comm, ierror)
 27
 TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
 28
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcounts(*)
 29
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 30
 TYPE(*), DIMENSION(..) :: recvbuf
 31
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
 32
 TYPE(MPI_Comm), INTENT(IN) :: comm
 33
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 34
 35
MPI_Neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
 36
 displs, recvtype, comm, ierror)
 37
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
 38
 INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
 39
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 40
 TYPE(*), DIMENSION(..) :: recvbuf
 41
 TYPE(MPI_Comm), INTENT(IN) :: comm
 42
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 43
MPI_Neighbor_allgatherv_init(sendbuf, sendcount, sendtype, recvbuf,
 44
 recvcounts, displs, recvtype, comm, info, request, ierror)
 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
```

```
1
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
2
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
3
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
4
 TYPE(MPI_Comm), INTENT(IN) :: comm
5
 TYPE(MPI_Info), INTENT(IN) :: info
6
 TYPE(MPI_Request), INTENT(OUT) :: request
7
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
 MPI_Neighbor_allgatherv_init(sendbuf, sendcount, sendtype, recvbuf,
9
 recvcounts, displs, recvtype, comm, info, request, ierror)
10
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
11
 INTEGER, INTENT(IN) :: sendcount, displs(*)
12
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
13
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
14
 INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
15
 TYPE(MPI_Comm), INTENT(IN) :: comm
16
 TYPE(MPI_Info), INTENT(IN) :: info
17
 TYPE(MPI_Request), INTENT(OUT) :: request
18
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20
 MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
21
 recvtype, comm, ierror)
22
 TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
23
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
24
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
25
 TYPE(*), DIMENSION(..) :: recvbuf
26
 TYPE(MPI_Comm), INTENT(IN) :: comm
27
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
 MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
29
 recvtype, comm, ierror)
30
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
31
 INTEGER, INTENT(IN) :: sendcount, recvcount
32
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
33
 TYPE(*), DIMENSION(..) :: recvbuf
34
 TYPE(MPI_Comm), INTENT(IN) :: comm
35
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
 MPI_Neighbor_alltoal1_init(sendbuf, sendcount, sendtype, recvbuf,
38
 recvcount, recvtype, comm, info, request, ierror)
39
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
40
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
41
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
42
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
43
 TYPE(MPI_Comm), INTENT(IN) :: comm
44
 TYPE(MPI_Info), INTENT(IN) :: info
45
 TYPE(MPI_Request), INTENT(OUT) :: request
46
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
```

```
1
MPI_Neighbor_alltoall_init(sendbuf, sendcount, sendtype, recvbuf,
 2
 recvcount, recvtype, comm, info, request, ierror)
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
 INTEGER, INTENT(IN) :: sendcount, recvcount
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 5
 6
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 TYPE(MPI_Comm), INTENT(IN) :: comm
 TYPE(MPI_Info), INTENT(IN) :: info
 TYPE(MPI_Request), INTENT(OUT) :: request
 9
 10
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 11
MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
 12
 recvcounts, rdispls, recvtype, comm, ierror)
 13
 TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
 14
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*),
 15
 recvcounts(*)
 16
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
 17
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 18
 TYPE(*), DIMENSION(..) :: recvbuf
 19
 TYPE(MPI_Comm), INTENT(IN) :: comm
 20
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 21
 22
MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
 23
 recvcounts, rdispls, recvtype, comm, ierror)
 24
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
 INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
 25
 26
 rdispls(*)
 27
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 28
 TYPE(*), DIMENSION(..) :: recvbuf
 29
 TYPE(MPI_Comm), INTENT(IN) :: comm
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 30
 31
MPI_Neighbor_alltoallv_init(sendbuf, sendcounts, sdispls, sendtype,
 32
 recvbuf, recvcounts, rdispls, recvtype, comm, info, request,
 33
 ierror)
 34
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
 35
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
 36
 sendcounts(*), recvcounts(*)
 37
 INTEGER(KIND=MP1_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_Info), INTENT(IN) :: info
 43
 TYPE(MPI_Request), INTENT(OUT) :: request
 44
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 45
 46
MPI_Neighbor_alltoallv_init(sendbuf, sendcounts, sdispls, sendtype,
 47
 recvbuf, recvcounts, rdispls, recvtype, comm, info, request,
 48
```

```
1
 ierror)
2
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
3
 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
4
 recvcounts(*), 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
 MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
12
 recvcounts, rdispls, recvtypes, comm, ierror)
13
 TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
14
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*),
15
 recvcounts(*)
16
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
17
 TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
18
 TYPE(*), DIMENSION(..) :: recvbuf
19
 TYPE(MPI_Comm), INTENT(IN) :: comm
20
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
 MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
23
 recvcounts, rdispls, recvtypes, comm, ierror)
24
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
25
 INTEGER, INTENT(IN) :: sendcounts(*), recvcounts(*)
26
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
27
 TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
28
 TYPE(*), DIMENSION(..) :: recvbuf
29
 TYPE(MPI_Comm), INTENT(IN) :: comm
30
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
 MPI_Neighbor_alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes,
32
 recvbuf, recvcounts, rdispls, recvtypes, comm, info, request,
33
 ierror)
34
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
35
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
36
 sendcounts(*), recvcounts(*)
37
 INTEGER(KIND=MP1_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
38
 rdispls(*)
39
 TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
40
 recvtypes(*)
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
47
48
```

```
1
MPI_Neighbor_alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes,
 \mathbf{2}
 recvbuf, recvcounts, rdispls, recvtypes, comm, info, request,
 ierror)
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
 4
 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
 5
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
 6
 rdispls(*)
 7
 TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
 9
 recvtypes(*)
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 10
 11
 TYPE(MPI_Comm), INTENT(IN) :: comm
 TYPE(MPI_Info), INTENT(IN) :: info
 12
 TYPE(MPI_Request), INTENT(OUT) :: request
 13
 14
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 15
MPI_Topo_test(comm, status, ierror)
 16
 TYPE(MPI_Comm), INTENT(IN) :: comm
 17
 INTEGER, INTENT(OUT) :: status
 18
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 19
 20
 21
A.4.7 MPI Environmental Management Fortran 2008 Bindings
 22
DOUBLE PRECISION MPI_Wtick()
 23
 24
DOUBLE PRECISION MPI_Wtime()
 25
MPI_Add_error_class(errorclass, ierror)
 26
 INTEGER, INTENT(OUT) :: errorclass
 27
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 28
 29
MPI_Add_error_code(errorclass, errorcode, ierror)
 30
 INTEGER, INTENT(IN) :: errorclass
 31
 INTEGER, INTENT(OUT) :: errorcode
 32
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 33
MPI_Add_error_string(errorcode, string, ierror)
 34
 INTEGER, INTENT(IN) :: errorcode
 35
 CHARACTER(LEN=*), INTENT(IN) :: string
 36
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 37
 38
MPI_Alloc_mem(size, info, baseptr, ierror)
 39
 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
 40
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
 41
 TYPE(MPI_Info), INTENT(IN) :: info
 42
 TYPE(C_PTR), INTENT(OUT) :: baseptr
 43
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 44
MPI_Comm_call_errhandler(comm, errorcode, ierror)
 45
 TYPE(MPI_Comm), INTENT(IN) :: comm
 46
 INTEGER, INTENT(IN) :: errorcode
 47
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 48
```

```
1
 MPI_Comm_create_errhandler(comm_errhandler_fn, errhandler, ierror)
\mathbf{2}
 PROCEDURE(MPI_Comm_errhandler_function), INTENT(IN) ::
3
 comm_errhandler_fn
4
 TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
5
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
 MPI_Comm_get_errhandler(comm, errhandler, ierror)
7
 TYPE(MPI_Comm), INTENT(IN) :: comm
8
 TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
9
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
 MPI_Comm_set_errhandler(comm, errhandler, ierror)
12
 TYPE(MPI_Comm), INTENT(IN) :: comm
13
 TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
14
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
 MPI_Errhandler_free(errhandler, ierror)
16
 TYPE(MPI_Errhandler), INTENT(INOUT) :: errhandler
17
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
 MPI_Error_class(errorcode, errorclass, ierror)
20
 INTEGER, INTENT(IN) :: errorcode
21
 INTEGER, INTENT(OUT) :: errorclass
22
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
 MPI_Error_string(errorcode, string, resultlen, ierror)
24
 INTEGER, INTENT(IN) :: errorcode
25
 CHARACTER(LEN=MPI_MAX_ERROR_STRING), INTENT(OUT) :: string
26
 INTEGER, INTENT(OUT) :: resultlen
27
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
 MPI_File_call_errhandler(fh, errorcode, ierror)
30
 TYPE(MPI_File), INTENT(IN) :: fh
31
 INTEGER, INTENT(IN) :: errorcode
32
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
 MPI_File_create_errhandler(file_errhandler_fn, errhandler, ierror)
34
 PROCEDURE(MPI_File_errhandler_function), INTENT(IN) ::
35
 file_errhandler_fn
36
 TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
37
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
 MPI_File_get_errhandler(file, errhandler, ierror)
40
 TYPE(MPI_File), INTENT(IN) :: file
41
 TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
42
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
 MPI_File_set_errhandler(file, errhandler, ierror)
44
 TYPE(MPI_File), INTENT(IN) :: file
45
 TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
46
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
```

```
1
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
 MPI_Win_set_errhandler(win, errhandler, ierror)
3
 TYPE(MPI_Win), INTENT(IN) :: win
4
 TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
5
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
7
8
 A.4.8 The Info Object Fortran 2008 Bindings
9
 MPI_Info_create(info, ierror)
10
 TYPE(MPI_Info), INTENT(OUT) :: info
11
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13
 MPI_Info_create_env(info, ierror)
14
 TYPE(MPI_Info), INTENT(OUT) :: info
15
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
 MPI_Info_delete(info, key, ierror)
 TYPE(MPI_Info), INTENT(IN) :: info
18
 CHARACTER(LEN=*), INTENT(IN) :: key
19
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
 MPI_Info_dup(info, newinfo, ierror)
22
 TYPE(MPI_Info), INTENT(IN) :: info
23
 TYPE(MPI_Info), INTENT(OUT) :: newinfo
24
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26
 MPI_Info_free(info, ierror)
 TYPE(MPI_Info), INTENT(INOUT) :: info
27
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
 MPI_Info_get(info, key, valuelen, value, flag, ierror)
30
 TYPE(MPI_Info), INTENT(IN) :: info
31
 CHARACTER(LEN=*), INTENT(IN) :: key
32
 INTEGER, INTENT(IN) :: valuelen
33
 CHARACTER(LEN=valuelen), INTENT(OUT) :: value
34
 LOGICAL, INTENT(OUT) :: flag
35
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
 MPI_Info_get_nkeys(info, nkeys, ierror)
 TYPE(MPI_Info), INTENT(IN) :: info
38
 INTEGER, INTENT(OUT) :: nkeys
39
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
 MPI_Info_get_nthkey(info, n, key, ierror)
42
 TYPE(MPI_Info), INTENT(IN) :: info
43
 INTEGER, INTENT(IN) :: n
44
 CHARACTER(LEN=*), INTENT(OUT) :: key
45
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
 MPI_Info_get_string(info, key, buflen, value, flag, ierror)
48
 TYPE(MPI_Info), INTENT(IN) :: info
```

```
1
 CHARACTER(LEN=*), INTENT(IN) :: key
 2
 INTEGER, INTENT(INOUT) :: buflen
 CHARACTER(LEN=*), INTENT(OUT) :: value
 LOGICAL, INTENT(OUT) :: flag
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 5
 6
MPI_Info_get_valuelen(info, key, valuelen, flag, ierror)
 TYPE(MPI_Info), INTENT(IN) :: info
 CHARACTER(LEN=*), INTENT(IN) :: key
 INTEGER, INTENT(OUT) :: valuelen
 10
 LOGICAL, INTENT(OUT) :: flag
 11
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 12
 13
MPI_Info_set(info, key, value, ierror)
 14
 TYPE(MPI_Info), INTENT(IN) :: info
 15
 CHARACTER(LEN=*), INTENT(IN) :: key, value
 16
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 17
 18
A.4.9 Process Creation and Management Fortran 2008 Bindings
 19
 20
MPI_Abort(comm, errorcode, ierror)
 21
 TYPE(MPI_Comm), INTENT(IN) :: comm
 22
 INTEGER, INTENT(IN) :: errorcode
 23
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 24
MPI_Close_port(port_name, ierror)
 25
 CHARACTER(LEN=*), INTENT(IN) :: port_name
 26
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 27
 28
MPI_Comm_accept(port_name, info, root, comm, newcomm, ierror)
 29
 CHARACTER(LEN=*), INTENT(IN) :: port_name
 30
 TYPE(MPI_Info), INTENT(IN) :: info
 31
 INTEGER, INTENT(IN) :: root
 32
 TYPE(MPI_Comm), INTENT(IN) :: comm
 33
 TYPE(MPI_Comm), INTENT(OUT) :: newcomm
 34
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 35
MPI_Comm_connect(port_name, info, root, comm, newcomm, ierror)
 36
 CHARACTER(LEN=*), INTENT(IN) :: port_name
 37
 TYPE(MPI_Info), INTENT(IN) :: info
 38
 INTEGER, INTENT(IN) :: root
 39
 TYPE(MPI_Comm), INTENT(IN) :: comm
 40
 TYPE(MPI_Comm), INTENT(OUT) :: newcomm
 41
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 42
 43
MPI_Comm_disconnect(comm, ierror)
 44
 TYPE(MPI_Comm), INTENT(INOUT) :: comm
 45
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 46
MPI_Comm_get_parent(parent, ierror)
 47
 TYPE(MPI_Comm), INTENT(OUT) :: parent
 48
```

```
1
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
 MPI_Comm_join(fd, intercomm, ierror)
3
 INTEGER, INTENT(IN) :: fd
4
 TYPE(MPI_Comm), INTENT(OUT) :: intercomm
5
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
7
 MPI_Comm_spawn(command, argv, maxprocs, info, root, comm, intercomm,
8
 array_of_errcodes, ierror)
9
 CHARACTER(LEN=*), INTENT(IN) :: command, argv(*)
10
 INTEGER, INTENT(IN) :: maxprocs, root
11
 TYPE(MPI_Info), INTENT(IN) :: info
12
 TYPE(MPI_Comm), INTENT(IN) :: comm
13
 TYPE(MPI_Comm), INTENT(OUT) :: intercomm
14
 INTEGER :: array_of_errcodes(*)
15
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
 MPI_Comm_spawn_multiple(count, array_of_commands, array_of_argv,
17
 array_of_maxprocs, array_of_info, root, comm, intercomm,
18
 array_of_errcodes, ierror)
19
 INTEGER, INTENT(IN) :: count, array_of_maxprocs(*), root
20
 CHARACTER(LEN=*), INTENT(IN) :: array_of_commands(*),
21
 array_of_argv(count, *)
22
 TYPE(MPI_Info), INTENT(IN) :: array_of_info(*)
23
 TYPE(MPI_Comm), INTENT(IN) :: comm
24
 TYPE(MPI_Comm), INTENT(OUT) :: intercomm
25
 INTEGER :: array_of_errcodes(*)
26
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
 MPI_Finalize(ierror)
29
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
 MPI_Finalized(flag, ierror)
31
 LOGICAL, INTENT(OUT) :: flag
32
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
 MPI_Init(ierror)
35
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
 MPI_Init_thread(required, provided, ierror)
37
 INTEGER, INTENT(IN) :: required
38
 INTEGER, INTENT(OUT) :: provided
39
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
 MPI_Initialized(flag, ierror)
42
 LOGICAL, INTENT(OUT) :: flag
43
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
 MPI_Is_thread_main(flag, ierror)
45
 LOGICAL, INTENT(OUT) :: flag
46
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
```

MDT Lashun name (zami za name infa namt name iaman)	1
MPI_Lookup_name(service_name, info, port_name, ierror)	2
CHARACTER(LEN=*), INTENT(IN) :: service_name	3
TYPE(MPI_Info), INTENT(IN) :: info	4
CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name	5
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	6
MPI_Open_port(info, port_name, ierror)	
TYPE(MPI_Info), INTENT(IN) :: info	7
CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name	8
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	9
	10
<pre>MPI_Publish_name(service_name, info, port_name, ierror)</pre>	11
CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name	12
TYPE(MPI_Info), INTENT(IN) :: info	13
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	14
MPI_Query_thread(provided, ierror)	15
INTEGER, INTENT(OUT) :: provided	16
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	17
	18
MPI_Session_finalize(session, ierror)	19
TYPE(MPI_Session), INTENT(INOUT) :: session	20
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	21
MPI_Session_get_info(session, info_used, ierror)	22
TYPE(MPI_Session), INTENT(IN) :: session	23
TYPE(MPI_Info), INTENT(OUT) :: info_used	24
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	25
INIEGER, OFFICIARE, INTENT(COF) TEFFOF	26
<pre>MPI_Session_get_nth_pset(session, info, n, pset_len, pset_name, ierror)</pre>	27
TYPE(MPI_Session), INTENT(IN) :: session	28
TYPE(MPI_Info), INTENT(IN) :: info	29
INTEGER, INTENT(IN) :: n	30
INTEGER, INTENT(INOUT) :: pset_len	31
CHARACTER(LEN=*), INTENT(OUT) :: pset_name	32
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	33
MPI_Session_get_num_psets(session, info, npset_names, ierror)	34
TYPE(MPI_Session), INTENT(IN) :: session	35
TYPE(MPI_Session), INTENT(IN) :: Session TYPE(MPI_Info), INTENT(IN) :: info	36
INTEGER, INTENT(OUT) :: npset_names	37
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	38
INTEGER, OFFTONAL, INTENT(001) TETTOT	39
<pre>MPI_Session_get_pset_info(session, pset_name, info, ierror)</pre>	40
TYPE(MPI_Session), INTENT(IN) :: session	41
CHARACTER(LEN=*), INTENT(IN) :: pset_name	42
TYPE(MPI_Info), INTENT(OUT) :: info	43
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	44
MDT Coggion init(info on hondlon cossis is is a	45
MPI_Session_init(info, errhandler, session, ierror)	46
TYPE(MPI_Info), INTENT(IN) :: info	47
TYPE(MPI_Errhandler), INTENT(IN) :: errhandler	48

```
1
 TYPE(MPI_Session), INTENT(OUT) :: session
\mathbf{2}
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
 MPI_Unpublish_name(service_name, info, port_name, ierror)
4
 CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name
5
 TYPE(MPI_Info), INTENT(IN) :: info
6
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
9
 A.4.10 One-Sided Communications Fortran 2008 Bindings
10
 MPI_Accumulate(origin_addr, origin_count, origin_datatype, target_rank,
11
 target_disp, target_count, target_datatype, op, win, ierror)
12
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
13
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
14
 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
15
 INTEGER, INTENT(IN) :: target_rank
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
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
 MPI_Accumulate(origin_addr, origin_count, origin_datatype, target_rank,
22
 target_disp, target_count, target_datatype, op, win, ierror)
23
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
24
 INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
25
 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
26
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
27
 TYPE(MPI_Op), INTENT(IN) :: op
28
 TYPE(MPI_Win), INTENT(IN) :: win
29
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
 MPI_Compare_and_swap(origin_addr, compare_addr, result_addr, datatype,
31
 target_rank, target_disp, win, ierror)
32
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr,
33
 compare_addr
34
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
35
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
36
 INTEGER, INTENT(IN) :: target_rank
37
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
38
 TYPE(MPI_Win), INTENT(IN) :: win
39
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
 MPI_Fetch_and_op(origin_addr, result_addr, datatype, target_rank,
42
 target_disp, op, win, ierror)
43
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
44
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
45
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
46
 INTEGER, INTENT(IN) :: target_rank
47
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
48
```

```
1
 TYPE(MPI_Op), INTENT(IN) :: op
 TYPE(MPI_Win), INTENT(IN) :: win
 2
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 4
MPI_Get(origin_addr, origin_count, origin_datatype, target_rank,
 5
 target_disp, target_count, target_datatype, win, ierror)
 6
 TYPE(*), DIMENSION(...), 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_Get(origin_addr, origin_count, origin_datatype, target_rank,
 15
 target_disp, target_count, target_datatype, win, ierror)
 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: origin_addr
 16
 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
 TYPE(MPI_Win), INTENT(IN) :: win
 20
 21
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 22
MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
 23
 result_count, result_datatype, target_rank, target_disp,
 24
 target_count, target_datatype, op, win, ierror)
 25
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
 26
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, result_count,
 27
 target_count
 28
 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
 29
 target_datatype
 30
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
 31
 INTEGER, INTENT(IN) :: target_rank
 32
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
 33
 TYPE(MPI_Op), INTENT(IN) :: op
 34
 TYPE(MPI_Win), INTENT(IN) :: win
 35
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 36
 37
MPI_Get_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, ierror)
 40
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
 41
 INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
 42
 target_count
 43
 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
 44
 target_datatype
 45
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
 46
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
 47
 TYPE(MPI_Op), INTENT(IN) :: op
 48
```

```
1
 TYPE(MPI_Win), INTENT(IN) :: win
2
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
 MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
4
 target_disp, target_count, target_datatype, win, ierror)
5
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
6
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
7
 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
8
 INTEGER, INTENT(IN) :: target_rank
9
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
10
 TYPE(MPI_Win), INTENT(IN) :: win
11
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13
 MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
14
 target_disp, target_count, target_datatype, win, ierror)
15
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
16
 INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
17
 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
18
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
19
 TYPE(MPI_Win), INTENT(IN) :: win
20
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
 MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
22
 target_disp, target_count, target_datatype, op, win, request,
23
 ierror)
24
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
25
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
26
 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
27
 INTEGER, INTENT(IN) :: target_rank
28
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
29
 TYPE(MPI_Op), INTENT(IN) :: op
30
 TYPE(MPI_Win), INTENT(IN) :: win
31
 TYPE(MPI_Request), INTENT(OUT) :: request
32
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
 MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
35
 target_disp, target_count, target_datatype, op, win, request,
36
 ierror)
37
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
38
 INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
39
 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
40
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
41
 TYPE(MPI_Op), INTENT(IN) :: op
42
 TYPE(MPI_Win), INTENT(IN) :: win
43
 TYPE(MPI_Request), INTENT(OUT) :: request
44
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
 MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank,
46
 target_disp, target_count, target_datatype, win, request,
47
 ierror)
48
```

TYPE(*), DIMENSION(...), ASYNCHRONOUS :: origin_addr 1 2 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count 3 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype INTEGER, INTENT(IN) :: target_rank 4 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp 5 6 TYPE(MPI_Win), INTENT(IN) :: win TYPE(MPI_Request), INTENT(OUT) :: request 7 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 9 MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank, 10 target_disp, target_count, target_datatype, win, request, 11 ierror) 12TYPE(*), DIMENSION(...), ASYNCHRONOUS :: origin_addr 13 INTEGER, INTENT(IN) :: origin_count, target_rank, target_count 14TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype 15INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp 16TYPE(MPI_Win), INTENT(IN) :: win 17TYPE(MPI_Request), INTENT(OUT) :: request 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 19 MPI_Rget_accumulate(origin_addr, origin_count, origin_datatype, 2021result_addr, result_count, result_datatype, target_rank, 22 target_disp, target_count, target_datatype, op, win, request, 23ierror) 24TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr 25INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, result_count, 26target_count 27TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype, 28 target_datatype 29 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr 30 INTEGER, INTENT(IN) :: target_rank INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp 31TYPE(MPI_Op), INTENT(IN) :: op 32 33 TYPE(MPI_Win), INTENT(IN) :: win 34 TYPE(MPI_Request), INTENT(OUT) :: request 35 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 36 MPI_Rget_accumulate(origin_addr, origin_count, origin_datatype, 37 result_addr, result_count, result_datatype, target_rank, 38 target_disp, target_count, target_datatype, op, win, request, 39 ierror) 40 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr 41 INTEGER, INTENT(IN) :: origin_count, result_count, target_rank, 42target_count 43 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype, 44 target_datatype 45TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr 46INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp 47TYPE(MPI_Op), INTENT(IN) :: op 48

```
1
 TYPE(MPI_Win), INTENT(IN) :: win
\mathbf{2}
 TYPE(MPI_Request), INTENT(OUT) :: request
3
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
 MPI_Rput(origin_addr, origin_count, origin_datatype, target_rank,
5
 target_disp, target_count, target_datatype, win, request,
6
 ierror)
7
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
8
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
9
 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
10
 INTEGER, INTENT(IN) :: target_rank
11
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
12
 TYPE(MPI_Win), INTENT(IN) :: win
13
 TYPE(MPI_Request), INTENT(OUT) :: request
14
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
 MPI_Rput(origin_addr, origin_count, origin_datatype, target_rank,
17
 target_disp, target_count, target_datatype, win, request,
18
 ierror)
19
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
20
 INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
21
 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
22
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
23
 TYPE(MPI_Win), INTENT(IN) :: win
24
 TYPE(MPI_Request), INTENT(OUT) :: request
25
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
 MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror)
27
 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
28
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
29
 INTEGER, INTENT(IN) :: disp_unit
30
 TYPE(MPI_Info), INTENT(IN) :: info
31
 TYPE(MPI_Comm), INTENT(IN) :: comm
32
 TYPE(C_PTR), INTENT(OUT) :: baseptr
33
 TYPE(MPI_Win), INTENT(OUT) :: win
34
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
 MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror)
37
 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
38
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size, disp_unit
39
 TYPE(MPI_Info), INTENT(IN) :: info
40
 TYPE(MPI_Comm), INTENT(IN) :: comm
41
 TYPE(C_PTR), INTENT(OUT) :: baseptr
42
 TYPE(MPI_Win), INTENT(OUT) :: win
43
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
 MPI_Win_allocate_shared(size, disp_unit, info, comm, baseptr, win, ierror)
45
 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
46
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
47
 INTEGER, INTENT(IN) :: disp_unit
48
```

TYPE(MPI_Info), INTENT(IN) :: info 1 TYPE(MPI_Comm), INTENT(IN) :: comm 2 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, disp_unit 9 TYPE(MPI_Info), INTENT(IN) :: info 10 TYPE(MPI_Comm), INTENT(IN) :: comm 11 TYPE(C_PTR), INTENT(OUT) :: baseptr 12TYPE(MPI_Win), INTENT(OUT) :: win 13 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1415MPI_Win_attach(win, base, size, ierror) 16TYPE(MPI_Win), INTENT(IN) :: win 17TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base 18 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size 19 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 20MPI_Win_complete(win, ierror) 21TYPE(MPI_Win), INTENT(IN) :: win 22 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2324MPI_Win_create(base, size, disp_unit, info, comm, win, ierror) 25TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base 26INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size 27INTEGER, INTENT(IN) :: disp_unit 28TYPE(MPI_Info), INTENT(IN) :: info 29 TYPE(MPI_Comm), INTENT(IN) :: comm 30 TYPE(MPI_Win), INTENT(OUT) :: win 31INTEGER, OPTIONAL, INTENT(OUT) :: ierror 32 MPI_Win_create(base, size, disp_unit, info, comm, win, ierror) 33 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: base 34 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size, disp_unit 35 TYPE(MPI_Info), INTENT(IN) :: info 36 TYPE(MPI_Comm), INTENT(IN) :: comm 37 TYPE(MPI_Win), INTENT(OUT) :: win 38 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 39 40 MPI_Win_create_dynamic(info, comm, win, ierror) 41 TYPE(MPI_Info), INTENT(IN) :: info 42TYPE(MPI_Comm), INTENT(IN) :: comm 43 TYPE(MPI_Win), INTENT(OUT) :: win 44INTEGER, OPTIONAL, INTENT(OUT) :: ierror 45MPI_Win_detach(win, base, ierror) 46TYPE(MPI_Win), INTENT(IN) :: win 47TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base 48

```
1
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
 MPI_Win_fence(assert, win, ierror)
3
 INTEGER, INTENT(IN) :: assert
4
 TYPE(MPI_Win), INTENT(IN) :: win
5
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
\overline{7}
 MPI_Win_flush(rank, win, ierror)
8
 INTEGER, INTENT(IN) :: rank
9
 TYPE(MPI_Win), INTENT(IN) :: win
10
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
 MPI_Win_flush_all(win, ierror)
12
 TYPE(MPI_Win), INTENT(IN) :: win
13
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
 MPI_Win_flush_local(rank, win, ierror)
16
 INTEGER, INTENT(IN) :: rank
17
 TYPE(MPI_Win), INTENT(IN) :: win
18
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
 MPI_Win_flush_local_all(win, ierror)
20
 TYPE(MPI_Win), INTENT(IN) :: win
21
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
 MPI_Win_free(win, ierror)
24
 TYPE(MPI_Win), INTENT(INOUT) :: win
25
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
 MPI_Win_get_group(win, group, ierror)
27
 TYPE(MPI_Win), INTENT(IN) :: win
28
 TYPE(MPI_Group), INTENT(OUT) :: group
29
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31
 MPI_Win_get_info(win, info_used, ierror)
32
 TYPE(MPI_Win), INTENT(IN) :: win
33
 TYPE(MPI_Info), INTENT(OUT) :: info_used
34
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
 MPI_Win_lock(lock_type, rank, assert, win, ierror)
36
 INTEGER, INTENT(IN) :: lock_type, rank, assert
37
 TYPE(MPI_Win), INTENT(IN) :: win
38
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
 MPI_Win_lock_all(assert, win, ierror)
41
 INTEGER, INTENT(IN) :: assert
42
 TYPE(MPI_Win), INTENT(IN) :: win
43
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
 MPI_Win_post(group, assert, win, ierror)
45
 TYPE(MPI_Group), INTENT(IN) :: group
46
47
 INTEGER, INTENT(IN) :: assert
 TYPE(MPI_Win), INTENT(IN) :: win
48
```

1 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2 MPI_Win_set_info(win, info, ierror) TYPE(MPI_Win), INTENT(IN) :: win TYPE(MPI_Info), INTENT(IN) :: info INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Win_shared_query(win, rank, size, disp_unit, baseptr, ierror) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR TYPE(MPI_Win), INTENT(IN) :: win 10 INTEGER, INTENT(IN) :: rank 11 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size INTEGER, INTENT(OUT) :: disp_unit 1213 TYPE(C_PTR), INTENT(OUT) :: baseptr 14INTEGER, OPTIONAL, INTENT(OUT) :: ierror 15MPI_Win_shared_query(win, rank, size, disp_unit, baseptr, ierror) 16USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR 17 TYPE(MPI_Win), INTENT(IN) :: win 18 INTEGER, INTENT(IN) :: rank 19 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size, disp_unit 20TYPE(C_PTR), INTENT(OUT) :: baseptr 21INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22 23MPI_Win_start(group, assert, win, ierror) 24TYPE(MPI_Group), INTENT(IN) :: group 25INTEGER, INTENT(IN) :: assert 26TYPE(MPI_Win), INTENT(IN) :: win 27INTEGER, OPTIONAL, INTENT(OUT) :: ierror 28MPI_Win_sync(win, ierror) 29 TYPE(MPI_Win), INTENT(IN) :: win 30 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 3132 MPI_Win_test(win, flag, ierror) 33 TYPE(MPI_Win), INTENT(IN) :: win 34 LOGICAL, INTENT(OUT) :: flag 35INTEGER, OPTIONAL, INTENT(OUT) :: ierror 36 MPI_Win_unlock(rank, win, ierror) 37 INTEGER, INTENT(IN) :: rank 38 TYPE(MPI_Win), INTENT(IN) :: win 39 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 40 41 MPI_Win_unlock_all(win, ierror) 42TYPE(MPI_Win), INTENT(IN) :: win 43 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 44 MPI_Win_wait(win, ierror) 45TYPE(MPI_Win), INTENT(IN) :: win 46INTEGER, OPTIONAL, INTENT(OUT) :: ierror 47

48

```
1
 A.4.11 External Interfaces Fortran 2008 Bindings
\mathbf{2}
 MPI_Grequest_complete(request, ierror)
3
 TYPE(MPI_Request), INTENT(IN) :: request
4
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
6
 MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,
7
 ierror)
8
 PROCEDURE(MPI_Grequest_query_function), INTENT(IN) :: query_fn
9
 PROCEDURE(MPI_Grequest_free_function), INTENT(IN) :: free_fn
10
 PROCEDURE(MPI_Grequest_cancel_function), INTENT(IN) :: cancel_fn
11
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
12
 TYPE(MPI_Request), INTENT(OUT) :: request
13
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
 MPI_Status_set_cancelled(status, flag, ierror)
15
 TYPE(MPI_Status), INTENT(INOUT) :: status
16
 LOGICAL, INTENT(IN) :: flag
17
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
 MPI_Status_set_elements(status, datatype, count, ierror)
20
 TYPE(MPI_Status), INTENT(INOUT) :: status
21
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
22
 INTEGER, INTENT(IN) :: count
23
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
 MPI_Status_set_elements_x(status, datatype, count, ierror)
25
 TYPE(MPI_Status), INTENT(INOUT) :: status
26
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
28
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
31
 A.4.12 I/O Fortran 2008 Bindings
32
 MPI_CONVERSION_FN_NULL(userbuf, datatype, count, filebuf, position,
33
34
 extra_state, ierror)
 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
35
 TYPE(C_PTR), VALUE :: userbuf, filebuf
36
 TYPE(MPI_Datatype) :: datatype
37
 INTEGER(KIND=MPI_COUNT_KIND) :: count
38
 INTEGER(KIND=MPI_OFFSET_KIND) :: position
39
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
40
 INTEGER :: ierror
41
42
 MPI_CONVERSION_FN_NULL(userbuf, datatype, count, filebuf, position,
43
 extra_state, ierror)
44
 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
45
 TYPE(C_PTR), VALUE :: userbuf, filebuf
46
 TYPE(MPI_Datatype) :: datatype
47
 INTEGER :: count, ierror
48
```

1 INTEGER(KIND=MPI_OFFSET_KIND) :: position 2 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state MPI_File_close(fh, ierror) TYPE(MPI_File), INTENT(INOUT) :: fh 5 INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_delete(filename, info, ierror) CHARACTER(LEN=*), INTENT(IN) :: filename TYPE(MPI_Info), INTENT(IN) :: info 10 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 11 MPI_File_get_amode(fh, amode, ierror) 12TYPE(MPI_File), INTENT(IN) :: fh 13 INTEGER, INTENT(OUT) :: amode 14INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1516MPI_File_get_atomicity(fh, flag, ierror) 17 TYPE(MPI_File), INTENT(IN) :: fh 18 LOGICAL, INTENT(OUT) :: flag 19 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 20MPI_File_get_byte_offset(fh, offset, disp, ierror) 21TYPE(MPI_File), INTENT(IN) :: fh 22 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset 23INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp  24 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2526MPI_File_get_group(fh, group, ierror) 27TYPE(MPI_File), INTENT(IN) :: fh 28TYPE(MPI_Group), INTENT(OUT) :: group 29INTEGER, OPTIONAL, INTENT(OUT) :: ierror 30 MPI_File_get_info(fh, info_used, ierror) 31TYPE(MPI_File), INTENT(IN) :: fh 32 TYPE(MPI_Info), INTENT(OUT) :: info_used 33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 34 35MPI_File_get_position(fh, offset, ierror) 36 TYPE(MPI_File), INTENT(IN) :: fh 37 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset 38 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 39 MPI_File_get_position_shared(fh, offset, ierror) 40 TYPE(MPI_File), INTENT(IN) :: fh 41 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset 42INTEGER, OPTIONAL, INTENT(OUT) :: ierror 43 44 MPI_File_get_size(fh, size, ierror) 45TYPE(MPI_File), INTENT(IN) :: fh 46INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: size 47INTEGER, OPTIONAL, INTENT(OUT) :: ierror 48

```
1
 MPI_File_get_type_extent(fh, datatype, extent, ierror)
\mathbf{2}
 TYPE(MPI_File), INTENT(IN) :: fh
3
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
4
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: extent
5
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
 MPI_File_get_type_extent(fh, datatype, extent, ierror)
7
 TYPE(MPI_File), INTENT(IN) :: fh
8
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
9
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: extent
10
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
 MPI_File_get_view(fh, disp, etype, filetype, datarep, ierror)
13
 TYPE(MPI_File), INTENT(IN) :: fh
14
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp
15
 TYPE(MPI_Datatype), INTENT(OUT) :: etype, filetype
16
 CHARACTER(LEN=*), INTENT(OUT) :: datarep
17
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
 MPI_File_iread(fh, buf, count, datatype, request, ierror)
19
 TYPE(MPI_File), INTENT(IN) :: fh
20
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
21
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
22
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
23
 TYPE(MPI_Request), INTENT(OUT) :: request
24
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26
 MPI_File_iread(fh, buf, count, datatype, request, ierror)
27
 TYPE(MPI_File), INTENT(IN) :: fh
28
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
29
 INTEGER, INTENT(IN) :: count
30
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
31
 TYPE(MPI_Request), INTENT(OUT) :: request
32
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
 MPI_File_iread_all(fh, buf, count, datatype, request, ierror)
34
 TYPE(MPI_File), INTENT(IN) :: fh
35
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
36
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
37
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
38
 TYPE(MPI_Request), INTENT(OUT) :: request
39
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
 MPI_File_iread_all(fh, buf, count, datatype, request, ierror)
42
 TYPE(MPI_File), INTENT(IN) :: fh
43
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
44
 INTEGER, INTENT(IN) :: count
45
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
46
 TYPE(MPI_Request), INTENT(OUT) :: request
47
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

1 MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror) 2 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 5 6 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) 10 TYPE(MPI_File), INTENT(IN) :: fh 11 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset 12TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf 13 INTEGER, INTENT(IN) :: count 14TYPE(MPI_Datatype), INTENT(IN) :: datatype 15TYPE(MPI_Request), INTENT(OUT) :: request 16INTEGER, OPTIONAL, INTENT(OUT) :: ierror 17 MPI_File_iread_at_all(fh, offset, buf, count, datatype, request, ierror) 18 19 TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset 2021TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf 22 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count 23TYPE(MPI_Datatype), INTENT(IN) :: datatype  24 TYPE(MPI_Request), INTENT(OUT) :: request 25INTEGER, OPTIONAL, INTENT(OUT) :: ierror 26MPI_File_iread_at_all(fh, offset, buf, count, datatype, request, ierror) 27TYPE(MPI_File), INTENT(IN) :: fh 28 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset 29 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf 30 INTEGER, INTENT(IN) :: count 31TYPE(MPI_Datatype), INTENT(IN) :: datatype 32 TYPE(MPI_Request), INTENT(OUT) :: request 33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 34 35 MPI_File_iread_shared(fh, buf, count, datatype, request, ierror) 36 TYPE(MPI_File), INTENT(IN) :: fh 37 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf 38 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count 39 TYPE(MPI_Datatype), INTENT(IN) :: datatype 40 TYPE(MPI_Request), INTENT(OUT) :: request 41 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 42MPI_File_iread_shared(fh, buf, count, datatype, request, ierror) 43 TYPE(MPI_File), INTENT(IN) :: fh 44TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf 45INTEGER, INTENT(IN) :: count 46TYPE(MPI_Datatype), INTENT(IN) :: datatype 47TYPE(MPI_Request), INTENT(OUT) :: request 48

```
1
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
 MPI_File_iwrite(fh, buf, count, datatype, request, ierror)
3
 TYPE(MPI_File), INTENT(IN) :: fh
4
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
5
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
6
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
7
 TYPE(MPI_Request), INTENT(OUT) :: request
8
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10
 MPI_File_iwrite(fh, buf, count, datatype, request, ierror)
11
 TYPE(MPI_File), INTENT(IN) :: fh
12
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
13
 INTEGER, INTENT(IN) :: count
14
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
15
 TYPE(MPI_Request), INTENT(OUT) :: request
16
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
 MPI_File_iwrite_all(fh, buf, count, datatype, request, ierror)
18
 TYPE(MPI_File), INTENT(IN) :: fh
19
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
20
 INTEGER(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
 MPI_File_iwrite_all(fh, buf, count, datatype, request, ierror)
26
 TYPE(MPI_File), INTENT(IN) :: fh
27
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
28
 INTEGER, INTENT(IN) :: count
29
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
30
 TYPE(MPI_Request), INTENT(OUT) :: request
31
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
 MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror)
33
 TYPE(MPI_File), INTENT(IN) :: fh
34
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
35
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
36
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
37
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
38
 TYPE(MPI_Request), INTENT(OUT) :: request
39
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
 MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror)
42
 TYPE(MPI_File), INTENT(IN) :: fh
43
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
44
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
45
 INTEGER, INTENT(IN) :: count
46
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
47
 TYPE(MPI_Request), INTENT(OUT) :: request
48
```

1 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2 MPI_File_iwrite_at_all(fh, offset, buf, count, datatype, request, ierror) 3 TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset 5 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf 6 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count 7 TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror 10 11 MPI_File_iwrite_at_all(fh, offset, buf, count, datatype, request, ierror) TYPE(MPI_File), INTENT(IN) :: fh 1213 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset 14TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 15INTEGER, INTENT(IN) :: count 16TYPE(MPI_Datatype), INTENT(IN) :: datatype 17TYPE(MPI_Request), INTENT(OUT) :: request 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 19 MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror) 20TYPE(MPI_File), INTENT(IN) :: fh 21TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf 22 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count 23TYPE(MPI_Datatype), INTENT(IN) :: datatype 24TYPE(MPI_Request), INTENT(OUT) :: request 25INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2627MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror) 28 TYPE(MPI_File), INTENT(IN) :: fh 29 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count 30 31TYPE(MPI_Datatype), INTENT(IN) :: datatype 32 TYPE(MPI_Request), INTENT(OUT) :: request 33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 34 MPI_File_open(comm, filename, amode, info, fh, ierror) 35 TYPE(MPI_Comm), INTENT(IN) :: comm 36 CHARACTER(LEN=*), INTENT(IN) :: filename 37 INTEGER, INTENT(IN) :: amode 38 TYPE(MPI_Info), INTENT(IN) :: info 39 TYPE(MPI_File), INTENT(OUT) :: fh 40 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 41 42MPI_File_preallocate(fh, size, ierror) 43 TYPE(MPI_File), INTENT(IN) :: fh 44INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size 45INTEGER, OPTIONAL, INTENT(OUT) :: ierror 46MPI_File_read(fh, buf, count, datatype, status, ierror) 47TYPE(MPI_File), INTENT(IN) :: fh 48

```
1
 TYPE(*), DIMENSION(..) :: buf
2
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
3
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
4
 TYPE(MPI_Status) :: status
5
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
 MPI_File_read(fh, buf, count, datatype, status, ierror)
7
 TYPE(MPI_File), INTENT(IN) :: fh
8
 TYPE(*), DIMENSION(..) :: buf
9
 INTEGER, 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_all(fh, buf, count, datatype, status, ierror)
15
 TYPE(MPI_File), INTENT(IN) :: fh
16
 TYPE(*), DIMENSION(..) :: buf
17
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
18
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
19
 TYPE(MPI_Status) :: status
20
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
 MPI_File_read_all(fh, buf, count, datatype, status, ierror)
22
 TYPE(MPI_File), INTENT(IN) :: fh
23
 TYPE(*), DIMENSION(..) :: buf
24
 INTEGER, INTENT(IN) :: count
25
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
26
 TYPE(MPI_Status) :: status
27
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
 MPI_File_read_all_begin(fh, buf, count, datatype, ierror)
30
 TYPE(MPI_File), INTENT(IN) :: fh
31
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
32
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
33
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
34
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
 MPI_File_read_all_begin(fh, buf, count, datatype, ierror)
36
 TYPE(MPI_File), INTENT(IN) :: fh
37
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
38
 INTEGER, INTENT(IN) :: count
39
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
40
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
 MPI_File_read_all_end(fh, buf, status, ierror)
43
 TYPE(MPI_File), INTENT(IN) :: fh
44
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
45
 TYPE(MPI_Status) :: status
46
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
 MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror)
48
```

```
1
 TYPE(MPI_File), INTENT(IN) :: fh
 2
 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
 5
 TYPE(MPI_Status) :: status
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror)
 TYPE(MPI_File), INTENT(IN) :: fh
 10
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
 11
 TYPE(*), DIMENSION(..) :: buf
 12
 INTEGER, INTENT(IN) :: count
 13
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 14
 TYPE(MPI_Status) :: status
 15
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 16
 17
MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror)
 18
 TYPE(MPI_File), INTENT(IN) :: fh
 19
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
 20
 TYPE(*), DIMENSION(..) :: buf
 21
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
 22
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 23
 TYPE(MPI_Status) :: status
 24
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 25
MPI_File_read_at_all(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
 34
MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror)
 35
 TYPE(MPI_File), INTENT(IN) :: fh
 36
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
 37
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
 38
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
 39
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 40
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 41
MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror)
 42
 TYPE(MPI_File), INTENT(IN) :: fh
 43
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
 44
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
 45
 INTEGER, INTENT(IN) :: count
 46
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 47
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 48
```

```
1
 MPI_File_read_at_all_end(fh, buf, status, ierror)
\mathbf{2}
 TYPE(MPI_File), INTENT(IN) :: fh
3
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
4
 TYPE(MPI_Status) :: status
5
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
 MPI_File_read_ordered(fh, buf, count, datatype, status, ierror)
7
 TYPE(MPI_File), INTENT(IN) :: fh
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_ordered(fh, buf, count, datatype, status, ierror)
15
 TYPE(MPI_File), INTENT(IN) :: fh
16
 TYPE(*), DIMENSION(..) :: buf
17
 INTEGER, INTENT(IN) :: count
18
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
19
 TYPE(MPI_Status) :: status
20
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
 MPI_File_read_ordered_begin(fh, buf, count, datatype, ierror)
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
 MPI_File_read_ordered_begin(fh, buf, count, datatype, ierror)
29
 TYPE(MPI_File), INTENT(IN) :: fh
30
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
31
 INTEGER, INTENT(IN) :: count
32
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
33
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
 MPI_File_read_ordered_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
40
 MPI_File_read_shared(fh, buf, count, datatype, status, ierror)
41
 TYPE(MPI_File), INTENT(IN) :: fh
42
 TYPE(*), DIMENSION(..) :: buf
43
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
44
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
45
 TYPE(MPI_Status) :: status
46
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
 MPI_File_read_shared(fh, buf, count, datatype, status, ierror)
48
```

1 TYPE(MPI_File), INTENT(IN) :: fh 2 TYPE(*), DIMENSION(..) :: buf INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Status) :: status 5 INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_seek(fh, offset, whence, ierror) TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset 10 INTEGER, INTENT(IN) :: whence 11 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1213 MPI_File_seek_shared(fh, offset, whence, ierror) TYPE(MPI_File), INTENT(IN) :: fh 1415INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset 16INTEGER, INTENT(IN) :: whence 17 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 18 MPI_File_set_atomicity(fh, flag, ierror) 19 TYPE(MPI_File), INTENT(IN) :: fh 20LOGICAL, INTENT(IN) :: flag 21INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22 23MPI_File_set_info(fh, info, ierror) 24TYPE(MPI_File), INTENT(IN) :: fh 25TYPE(MPI_Info), INTENT(IN) :: info 26INTEGER, OPTIONAL, INTENT(OUT) :: ierror 27MPI_File_set_size(fh, size, ierror) 28TYPE(MPI_File), INTENT(IN) :: fh 29 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size 30 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 3132 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 35TYPE(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 MPI_File_sync(fh, ierror) 40 TYPE(MPI_File), INTENT(IN) :: fh 41 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4243 MPI_File_write(fh, buf, count, datatype, status, ierror) 44TYPE(MPI_File), INTENT(IN) :: fh 45TYPE(*), DIMENSION(..), INTENT(IN) :: buf 46INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count 47TYPE(MPI_Datatype), INTENT(IN) :: datatype 48

```
1
 TYPE(MPI_Status) :: status
\mathbf{2}
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
 MPI_File_write(fh, buf, count, datatype, status, ierror)
4
 TYPE(MPI_File), INTENT(IN) :: fh
5
 TYPE(*), DIMENSION(..), INTENT(IN) :: buf
6
 INTEGER, INTENT(IN) :: count
7
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
8
 TYPE(MPI_Status) :: status
9
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
 MPI_File_write_all(fh, buf, count, datatype, status, ierror)
12
 TYPE(MPI_File), INTENT(IN) :: fh
13
 TYPE(*), DIMENSION(..), INTENT(IN) :: buf
14
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
15
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
16
 TYPE(MPI_Status) :: status
17
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
 MPI_File_write_all(fh, buf, count, datatype, status, ierror)
19
 TYPE(MPI_File), INTENT(IN) :: fh
20
 TYPE(*), DIMENSION(...), INTENT(IN) :: buf
21
 INTEGER, INTENT(IN) :: count
22
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
23
 TYPE(MPI_Status) :: status
24
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26
 MPI_File_write_all_begin(fh, buf, count, datatype, ierror)
27
 TYPE(MPI_File), INTENT(IN) :: fh
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
28
29
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
30
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
31
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
 MPI_File_write_all_begin(fh, buf, count, datatype, ierror)
33
 TYPE(MPI_File), INTENT(IN) :: fh
34
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
35
 INTEGER, INTENT(IN) :: count
36
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
37
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
 MPI_File_write_all_end(fh, buf, status, ierror)
40
 TYPE(MPI_File), INTENT(IN) :: fh
41
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
42
 TYPE(MPI_Status) :: status
43
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
 MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror)
45
 TYPE(MPI_File), INTENT(IN) :: fh
46
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
47
 TYPE(*), DIMENSION(...), INTENT(IN) :: buf
48
```

```
1
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
 2
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 TYPE(MPI_Status) :: status
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror)
 6
 TYPE(MPI_File), INTENT(IN) :: fh
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
 TYPE(*), DIMENSION(..), INTENT(IN) :: buf
 INTEGER, INTENT(IN) :: count
 10
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 11
 TYPE(MPI_Status) :: status
 12
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 13
MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)
 14
 15
 TYPE(MPI_File), INTENT(IN) :: fh
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
 16
 17
 TYPE(*), DIMENSION(...), INTENT(IN) :: buf
 18
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
 19
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 TYPE(MPI_Status) :: status
 20
 21
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 22
MPI_File_write_at_all(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
 31
MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror)
 32
 TYPE(MPI_File), INTENT(IN) :: fh
 33
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
 34
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
 35
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
 36
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 37
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 38
MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror)
 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, 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
```

```
1
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
2
 TYPE(MPI_Status) :: status
3
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
 MPI_File_write_ordered(fh, buf, count, datatype, status, ierror)
5
 TYPE(MPI_File), INTENT(IN) :: fh
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_ordered(fh, buf, count, datatype, status, ierror)
13
 TYPE(MPI_File), INTENT(IN) :: fh
14
 TYPE(*), DIMENSION(...), INTENT(IN) :: buf
15
 INTEGER, INTENT(IN) :: count
16
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
17
 TYPE(MPI_Status) :: status
18
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
 MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror)
20
 TYPE(MPI_File), INTENT(IN) :: fh
21
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
22
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
23
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
24
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26
 MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror)
27
 TYPE(MPI_File), INTENT(IN) :: fh
28
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
29
 INTEGER, INTENT(IN) :: count
30
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
31
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
 MPI_File_write_ordered_end(fh, buf, status, ierror)
33
 TYPE(MPI_File), INTENT(IN) :: fh
34
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
35
 TYPE(MPI_Status) :: status
36
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
 MPI_File_write_shared(fh, buf, count, datatype, status, ierror)
39
 TYPE(MPI_File), INTENT(IN) :: fh
40
 TYPE(*), DIMENSION(..), INTENT(IN) :: buf
41
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
42
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
43
 TYPE(MPI_Status) :: status
44
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
 MPI_File_write_shared(fh, buf, count, datatype, status, ierror)
46
 TYPE(MPI_File), INTENT(IN) :: fh
47
 TYPE(*), DIMENSION(...), INTENT(IN) :: buf
48
```

```
1
 INTEGER, INTENT(IN) :: count
 2
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 TYPE(MPI_Status) :: status
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 5
MPI_Register_datarep(datarep, read_conversion_fn, write_conversion_fn,
 6
 dtype_file_extent_fn, extra_state, ierror)
 CHARACTER(LEN=*), INTENT(IN) :: datarep
 PROCEDURE(MPI_Datarep_conversion_function), INTENT(IN) ::
 9
 read_conversion_fn, write_conversion_fn
 10
 PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn
 11
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
 12
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 13
 14
MPI_Register_datarep_c(datarep, read_conversion_fn, write_conversion_fn,
 15
 dtype_file_extent_fn, extra_state, ierror)
 16
 CHARACTER(LEN=*), INTENT(IN) :: datarep
 17
 PROCEDURE(MPI_Datarep_conversion_function_c), INTENT(IN) ::
 18
 read_conversion_fn, write_conversion_fn
 19
 PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
 20
 21
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 22
 23
A.4.13 Language Bindings Fortran 2008 Bindings
 24
 25
MPI_F_sync_reg(buf)
 26
 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
 27
MPI_Status_f082f(f08_status, f_status, ierror)
 28
 TYPE(MPI_Status), INTENT(IN) :: f08_status
 29
 INTEGER, INTENT(OUT) :: f_status(MPI_STATUS_SIZE)
 30
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 31
 32
MPI_Status_f2f08(f_status, f08_status, ierror)
 33
 INTEGER, INTENT(IN) :: f_status(MPI_STATUS_SIZE)
 34
 TYPE(MPI_Status), INTENT(OUT) :: f08_status
 35
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 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
MPI_Type_create_f90_real(p, r, newtype, ierror)
 46
 INTEGER, INTENT(IN) :: p, r
 47
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
 48
```

```
1
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
 MPI_Type_match_size(typeclass, size, datatype, ierror)
3
 INTEGER, INTENT(IN) :: typeclass, size
4
 TYPE(MPI_Datatype), INTENT(OUT) :: datatype
5
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
7
8
 A.4.14 Tools / Profiling Interface Fortran 2008 Bindings
9
 MPI_Pcontrol(level)
10
 INTEGER, INTENT(IN) :: level
11
12
13
 A.4.15
 Deprecated Fortran 2008 Bindings
14
15
 MPI_Info_get(info, key, valuelen, value, flag, ierror)
16
 TYPE(MPI_Info), INTENT(IN) :: info
 CHARACTER(LEN=*), INTENT(IN) :: key
17
18
 INTEGER, INTENT(IN) :: valuelen
19
 CHARACTER(LEN=valuelen), INTENT(OUT) :: value
 LOGICAL, INTENT(OUT) :: flag
20
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
 MPI_Info_get_valuelen(info, key, valuelen, flag, ierror)
23
 TYPE(MPI_Info), INTENT(IN) :: info
24
 CHARACTER(LEN=*), INTENT(IN) :: key
25
 INTEGER, INTENT(OUT) :: valuelen
26
 LOGICAL, INTENT(OUT) :: flag
27
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
 MPI_Sizeof(x, size, ierror)
29
30
 TYPE(*), DIMENSION(..) :: x
31
 INTEGER, INTENT(OUT) :: size
32
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
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	999
A.5 Fortran Bindings with mpif.h or the mpi Module	1
A.5.1 Point-to-Point Communication Fortran Bindings	3
MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)	4
<type> BUF(*)</type>	5
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR	7
<pre>MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)</pre>	8 9
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR	10
MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR)	11 12
<type> BUFFER(*)</type>	13
INTEGER SIZE, IERROR	14
MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)	15 16
<type> BUFFER_ADDR(*) INTEGER SIZE, IERROR</type>	10
MPI_CANCEL(REQUEST, IERROR)	18
INTEGER REQUEST, IERROR	19 20
MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)	20
INTEGER STATUS (MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR	22
MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	23 24
<type> BUF(*)</type>	24
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR	26
MPI_IMPROBE(SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS, IERROR)	27
INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR LOGICAL FLAG	28 29
MPI_IMRECV(BUF, COUNT, DATATYPE, MESSAGE, REQUEST, IERROR)	30 31
<type> BUF(*)</type>	32
INTEGER COUNT, DATATYPE, MESSAGE, REQUEST, IERROR	33
MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)	34 35
INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR LOGICAL FLAG	36
	37
<pre>MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)</pre>	38
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR	39 40
MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	41
<pre><type> BUF(*)</type></pre>	42
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR	43 44
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	45
<type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR</type>	46
INIEGER COUNT, DRIRTITE, DEDI, ING, CONT, REQUEDI, IERROR	47 48

1 MPI_ISENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,  $\mathbf{2}$ RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, REQUEST, IERROR) 3 <type> SENDBUF(*), RECVBUF(*) 4 INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE, 5SOURCE, RECVTAG, COMM, REQUEST, IERROR 6 MPI_ISENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, 7 COMM, REQUEST, IERROR) 8 <type> BUF(*) 9 INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, REQUEST, 10 IERROR 11 12MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 13<type> BUF(*) 14INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 15MPI_MPROBE(SOURCE, TAG, COMM, MESSAGE, STATUS, IERROR) 16INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR 1718MPI_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR) 19<type> BUF(*) 20INTEGER COUNT, DATATYPE, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR 21MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR) 22INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR 23 24 MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR) 25<type> BUF(*) 26INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), 27IERROR 28MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR) 29 <type> BUF(*) 30 INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR  31 32 MPI_REQUEST_FREE(REQUEST, IERROR) 33 INTEGER REQUEST, IERROR 34 MPI_REQUEST_GET_STATUS (REQUEST, FLAG, STATUS, IERROR) 35 INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR 36 LOGICAL FLAG 37 38 MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) 39 <type> BUF(*) 40INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR 41 MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 42<type> BUF(*) 43 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 4445MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) 46<type> BUF(*) 47INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR 48

MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF, 1 RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR) 2 <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE, 4 SOURCE. RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR 5MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS, IERROR) <type> BUF(*) 9 INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, 10 STATUS(MPI_STATUS_SIZE), IERROR 11 MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 1213 <type> BUF(*) 14INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 15MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) 16<type> BUF(*) 17INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR 18 19 MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 20<type> BUF(*) 21INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 22 MPI_START(REQUEST, IERROR) 23INTEGER REQUEST, IERROR 2425MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR) 26INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR 27MPI_TEST(REQUEST, FLAG, STATUS, IERROR) 28 INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR 29LOGICAL FLAG 30 31MPI TESTALL (COUNT, ARRAY OF REQUESTS, FLAG, ARRAY OF STATUSES, IERROR) 32 INTEGER COUNT, ARRAY_OF_REQUESTS(*), ARRAY_OF_STATUSES(MPI_STATUS_SIZE, 33 *), IERROR 34 LOGICAL FLAG 35 MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR) 36 INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE), 37 IERROR 38 LOGICAL FLAG 39 40 MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES, 41 ARRAY_OF_STATUSES, IERROR) 42INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*), 43 ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *), IERROR 44 MPI_TEST_CANCELLED(STATUS, FLAG, IERROR) 45INTEGER STATUS(MPI_STATUS_SIZE), IERROR 46LOGICAL FLAG 4748

```
1
 MPI_WAIT(REQUEST, STATUS, IERROR)
\mathbf{2}
 INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
3
 MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)
4
 INTEGER COUNT, ARRAY_OF_REQUESTS(*), ARRAY_OF_STATUSES(MPI_STATUS_SIZE,
5
 *), IERROR
6
\overline{7}
 MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)
8
 INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
9
 IERROR
10
 MPI WAITSOME (INCOUNT, ARRAY OF REQUESTS, OUTCOUNT, ARRAY OF INDICES,
11
 ARRAY_OF_STATUSES, IERROR)
12
 INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
13
 ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *), IERROR
14
15
16
 A.5.2 Partitioned Communication Fortran Bindings
17
 MPI_PARRIVED(REQUEST, PARTITION, FLAG, IERROR)
18
 INTEGER REQUEST, PARTITION, IERROR
19
 LOGICAL FLAG
20
21
 MPI_PREADY(PARTITION, REQUEST, IERROR)
22
 INTEGER PARTITION, REQUEST, IERROR
23
 MPI_PREADY_LIST(LENGTH, ARRAY_OF_PARTITIONS, REQUEST, IERROR)
^{24}
 INTEGER LENGTH, ARRAY_OF_PARTITIONS(*), REQUEST, IERROR
25
26
 MPI_PREADY_RANGE(PARTITION_LOW, PARTITION_HIGH, REQUEST, IERROR)
27
 INTEGER PARTITION_LOW, PARTITION_HIGH, REQUEST, IERROR
28
 MPI_PRECV_INIT(BUF, PARTITIONS, COUNT, DATATYPE, DEST, TAG, COMM, INFO,
29
 REQUEST, IERROR)
30
31
 <type> BUF(*)
 INTEGER PARTITIONS, DATATYPE, DEST, TAG, COMM, INFO, REQUEST, IERROR
32
33
 INTEGER(KIND=MPI_COUNT_KIND) COUNT
34
 MPI_PSEND_INIT(BUF, PARTITIONS, COUNT, DATATYPE, DEST, TAG, COMM, INFO,
35
 REQUEST, IERROR)
36
 <type> BUF(*)
37
 INTEGER PARTITIONS, DATATYPE, DEST, TAG, COMM, INFO, REQUEST, IERROR
38
 INTEGER(KIND=MPI_COUNT_KIND) COUNT
39
40
41
 A.5.3 Datatypes Fortran Bindings
42
 INTEGER(KIND=MPI_ADDRESS_KIND) MPI_AINT_ADD(BASE, DISP)
43
 INTEGER(KIND=MPI_ADDRESS_KIND) BASE, DISP
44
45
 INTEGER(KIND=MPI_ADDRESS_KIND) MPI_AINT_DIFF(ADDR1, ADDR2)
46
 INTEGER(KIND=MPI_ADDRESS_KIND) ADDR1, ADDR2
47
 MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)
48
```

<type> LOCATION(*) INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS</type>	1 2
INTEGER IERROR	3
MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)	4
INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR	5 6
MPI_GET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)	7
INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR	8
INTEGER(KIND=MPI_COUNT_KIND) COUNT	9
	10
<pre>MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)</pre>	11 12
INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR	13
MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,	14
POSITION, IERROR)	15
CHARACTER*(*) DATAREP	16
<type> INBUF(*), OUTBUF(*)</type>	17
INTEGER INCOUNT, DATATYPE, IERROR	18
INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, POSITION	19
MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR)	20 21
CHARACTER*(*) DATAREP	21
INTEGER INCOUNT, DATATYPE, IERROR	23
INTEGER(KIND=MPI_ADDRESS_KIND) SIZE	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
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)	30
INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR	31
MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS, ARRAY_OF_GSIZES,	32 33
ARRAY_OF_DISTRIBS, ARRAY_OF_DARGS, ARRAY_OF_PSIZES, ORDER,	34
OLDTYPE, NEWTYPE, IERROR)	35
INTEGER SIZE, RANK, NDIMS, ARRAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*),	36
ARRAY_OF_DARGS(*), ARRAY_OF_PSIZES(*), ORDER, OLDTYPE,	37
NEWTYPE, IERROR	38
MPI_TYPE_CREATE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,	39
ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)	40
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), OLDTYPE, NEWTYPE, IERROR	41
INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)	42
	43
MPI_TYPE_CREATE_HINDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,	44
OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR	45 46
INTEGER (COONI, BLOCKLENGIR, OLDITTE, NEWITTE, TERROR INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)	40
	48

1	MPI_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,
2	IERROR)
3	INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR
4	INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE
5	MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,
6	OLDTYPE, NEWTYPE, IERROR)
7	INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDTYPE,
8	NEWTYPE, IERROR
9	NEWIIPE, IERROR
10	MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)
11	INTEGER OLDTYPE, NEWTYPE, IERROR
12	INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT
13	
14	MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,
15	ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR)
16	INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_TYPES(*), NEWTYPE,
17	IERROR
18	INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
19	MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES, ARRAY_OF_SUBSIZES,
20	ARRAY_OF_STARTS, ORDER, OLDTYPE, NEWTYPE, IERROR)
21	INTEGER NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*),
22	ARRAY_OF_STARTS(*), ORDER, OLDTYPE, NEWTYPE, IERROR
23	
24	MPI_TYPE_DUP(OLDTYPE, NEWTYPE, IERROR)
25	INTEGER OLDTYPE, NEWTYPE, IERROR
26	MPI_TYPE_FREE(DATATYPE, IERROR)
27	INTEGER DATATYPE, IERROR
28	INIEGEI DATAITE, IEIdion
29	MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,
30	ARRAY_OF_INTEGERS, ARRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES,
31	IERROR)
32	INTEGER DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,
33	ARRAY_OF_INTEGERS(*), ARRAY_OF_DATATYPES(*), IERROR
34	INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_ADDRESSES(*)
35	MDT TYDE CET ENVELODE DATATYDE NUM INTEGEDS NUM ADDRESSES NUM DATATYDES
36	MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES,
37	COMBINER, IERROR)
38	INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER,
39	IERROR
40	MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR)
41	INTEGER DATATYPE, IERROR
42	INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT
43	
44	MPI_TYPE_GET_EXTENT_X(DATATYPE, LB, EXTENT, IERROR)
45	INTEGER DATATYPE, IERROR
46	INTEGER(KIND=MPI_COUNT_KIND) LB, EXTENT
47	MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
48	INTEGER DATATYPE, IERROR

A.5. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE	1005
INTEGER(KIND=MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT	1
MPI_TYPE_GET_TRUE_EXTENT_X(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR) INTEGER DATATYPE, IERROR INTEGER(KIND=MPI_COUNT_KIND) TRUE_LB, TRUE_EXTENT	2 3 4 5
<pre>MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*), OLDTYPE, NEWTYPE, IERROR</pre>	6 7 8 9 10
MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR) INTEGER DATATYPE, SIZE, IERROR	10 11 12
MPI_TYPE_SIZE_X(DATATYPE, SIZE, IERROR) INTEGER DATATYPE, IERROR INTEGER(KIND=MPI_COUNT_KIND) SIZE	13 14 15 16
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR	17 18
MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM, IERROR) <type> INBUF(*), OUTBUF(*) INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR</type>	19 20 21 22 23
<pre>MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, IERROR) CHARACTER*(*) DATAREP <type> INBUF(*), OUTBUF(*) INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION INTEGER OUTCOUNT, DATATYPE, IERROR</type></pre>	24 25 26 27 28 29 30
A.5.4 Collective Communication Fortran Bindings	31 32
<pre>MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR</type></pre>	34 35 36
<pre>MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS RECVTYPE, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COM IERROR</type></pre>	38 39 40
<pre>MPI_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COM INFO, REQUEST, IERROR</type></pre>	43 44 45 M, 46 47 48

```
1006
```

1MPI_ALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,  $\mathbf{2}$ RECVTYPE, COMM, INFO, REQUEST, IERROR) 3 <type> SENDBUF(*), RECVBUF(*) 4 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, 5IERROR 6 MPI ALLREDUCE (SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR) 7 <type> SENDBUF(*), RECVBUF(*) 8 INTEGER COUNT, DATATYPE, OP, COMM, IERROR 9 10 MPI_ALLREDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, 11 REQUEST, IERROR) 12<type> SENDBUF(*), RECVBUF(*) 13INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR 14MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 15COMM, IERROR) 16<type> SENDBUF(*), RECVBUF(*) 17INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR 18 19 MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, 20RDISPLS, RECVTYPE, COMM, IERROR) 21<type> SENDBUF(*), RECVBUF(*) 22INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), 23RECVTYPE, COMM, IERROR 24MPI_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, 25RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR) 26<type> SENDBUF(*), RECVBUF(*) 27INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), 28RECVTYPE, COMM, INFO, REQUEST, IERROR 29 30 MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS, 31RDISPLS, RECVTYPES, COMM, IERROR) 32 <type> SENDBUF(*), RECVBUF(*) 33 INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*), 34 RDISPLS(*), RECVTYPES(*), COMM, IERROR 35 MPI_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, 36 RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST, IERROR) 37 <type> SENDBUF(*), RECVBUF(*) 38 INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*), 39 RDISPLS(*), RECVTYPES(*), COMM, INFO, REQUEST, IERROR 4041 MPI_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 42RECVTYPE, COMM, INFO, REQUEST, IERROR) 43 <type> SENDBUF(*), RECVBUF(*) 44INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, 45IERROR 46MPI_BARRIER(COMM, IERROR) 47INTEGER COMM, IERROR 48

A.5. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE 10	007
MPI_BARRIER_INIT(COMM, INFO, REQUEST, IERROR) INTEGER COMM, INFO, REQUEST, IERROR	1 2
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR) <type> BUFFER(*) INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR</type>	3 4 5 6
MPI_BCAST_INIT(BUFFER, COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR <type> BUFFER(*) INTEGER COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR</type>	8 9
<pre>MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)</pre>	10 11 12 13
<pre>MPI_EXSCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUES IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR</type></pre>	<b>5T</b> , ¹⁴ 15 16 17 18
<pre>MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR</type></pre>	19 20 21 22
<pre>MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,</pre>	27
<pre>MPI_GATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPL RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT COMM, INFO, REQUEST, IERROR</type></pre>	30 31
<pre>MPI_GATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYP ROOT, COMM, INFO, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR</type></pre>	E, 34 35 36 37 38 39
<pre>MPI_IALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERR</type></pre>	40 41 42
MPI_IALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*)</type>	44 45 46 47 48

1 2	INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, REQUEST, IERROR
3 4 5	MPI_IALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)
6 7	<type> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR</type>
8 9 10 11	<pre>MPI_IALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR</type></pre>
12 13 14 15 16 17	<pre>MPI_IALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,</pre>
18 19 20 21 22 23	<pre>MPI_IALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,</pre>
23 24 25	MPI_IBARRIER(COMM, REQUEST, IERROR) INTEGER COMM, REQUEST, IERROR
26 27 28 29	<pre>MPI_IBCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR)</pre>
30 31 32	<pre>MPI_IEXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)</pre>
33 34 35 36 37 38	<pre>MPI_IGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST, IERROR</type></pre>
<ul> <li>39</li> <li>40</li> <li>41</li> <li>42</li> <li>43</li> </ul>	<pre>MPI_IGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,</pre>
44 45 46 47 48	<pre>MPI_IREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST,</pre>

MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, 1 2 REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, REQUEST, IERROR MPI_IREDUCE_SCATTER_BLOCK (SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM, 6 REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER RECVCOUNT, DATATYPE, OP, COMM, REQUEST, IERROR 9 10 MPI_ISCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR) 11 <type> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR 1213 MPI_ISCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 14ROOT, COMM, REQUEST, IERROR) 15<type> SENDBUF(*), RECVBUF(*) 16 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST, 17IERROR 18 19 MPI_ISCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, 20RECVTYPE, ROOT, COMM, REQUEST, IERROR) 21<type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, 22 23COMM, REQUEST, IERROR 24MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR) 25INTEGER OP, IERROR 26LOGICAL COMMUTE 2728 MPI_OP_CREATE(USER_FN, COMMUTE, OP, IERROR) 29 EXTERNAL USER_FN 30 LOGICAL COMMUTE 31INTEGER OP, IERROR 32 MPI_OP_FREE(OP, IERROR) 33 INTEGER OP, IERROR 34 35 MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR) 36 <type> SENDBUF(*), RECVBUF(*) 37 INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR 38 MPI_REDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, INFO, 39 REQUEST, IERROR) 40 <type> SENDBUF(*), RECVBUF(*) 41 INTEGER COUNT, DATATYPE, OP, ROOT, COMM, INFO, REQUEST, IERROR 4243 MPI_REDUCE_LOCAL(INBUF, INOUTBUF, COUNT, DATATYPE, OP, IERROR) 44<type> INBUF(*), INOUTBUF(*) 45INTEGER COUNT, DATATYPE, OP, IERROR 46MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, 47IERROR) 48

```
1
 <type> SENDBUF(*), RECVBUF(*)
\mathbf{2}
 INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR
3
 MPI_REDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
4
 IERROR)
5
 <type> SENDBUF(*), RECVBUF(*)
6
 INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR
7
8
 MPI_REDUCE_SCATTER_BLOCK_INIT(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP,
9
 COMM, INFO, REQUEST, IERROR)
10
 <type> SENDBUF(*), RECVBUF(*)
11
 INTEGER RECVCOUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
12
 MPI_REDUCE_SCATTER_INIT(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
13
 INFO, REQUEST, IERROR)
14
 <type> SENDBUF(*), RECVBUF(*)
15
 INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, INFO, REQUEST, IERROR
16
17
 MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
18
 <type> SENDBUF(*), RECVBUF(*)
19
 INTEGER COUNT, DATATYPE, OP, COMM, IERROR
20
 MPI_SCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,
21
 IERROR)
22
 <type> SENDBUF(*), RECVBUF(*)
23
 INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
24
25
 MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
26
 ROOT, COMM, IERROR)
27
 <type> SENDBUF(*), RECVBUF(*)
28
 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
29
 MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
30
 RECVTYPE, ROOT, COMM, IERROR)
31
 <type> SENDBUF(*), RECVBUF(*)
32
 INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
33
 COMM. IERROR
34
35
 MPI_SCATTERV_INIT(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF,
36
 RECVCOUNT, RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)
37
 <type> SENDBUF(*), RECVBUF(*)
38
 INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
39
 COMM, INFO, REQUEST, IERROR
40
 MPI_SCATTER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
41
 RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)
42
 <type> SENDBUF(*), RECVBUF(*)
43
 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO,
44
 REQUEST, IERROR
45
46
47
48
```

A.5. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE 101	.1
A.5.5 Groups, Contexts, Communicators, and Caching Fortran Bindings	1
MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)	2
INTEGER COMM1, COMM2, RESULT, IERROR	3 4
MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)	5
INTEGER COMM, GROUP, NEWCOMM, IERROR	6
MPI_COMM_CREATE_FROM_GROUP(GROUP, STRINGTAG, INFO, ERRHANDLER, NEWCOMM,	7
IERROR)	8 9
INTEGER GROUP, INFO, ERRHANDLER, NEWCOMM, IERROR	10
CHARACTER*(*) STRINGTAG	11
MPI_COMM_CREATE_GROUP(COMM, GROUP, TAG, NEWCOMM, IERROR) INTEGER COMM, GROUP, TAG, NEWCOMM, IERROR	12 13
MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL	14 • 15
EXTRA_STATE, IERROR)	16
EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN INTEGER COMM_KEYVAL, IERROR	17
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	18 19
MPI_COMM_DELETE_ATTR(COMM, COMM_KEYVAL, IERROR)	20
INTEGER COMM, COMM_KEYVAL, IERROR	21
MPI_COMM_DUP(COMM, NEWCOMM, IERROR)	22
INTEGER COMM, NEWCOMM, IERROR	23 24
MPI_COMM_DUP_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	25
ATTRIBUTE_VAL_OUT, FLAG, IERROR)	26
INTEGER OLDCOMM, COMM_KEYVAL, IERROR	27
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	28 29
ATTRIBUTE_VAL_OUT LOGICAL FLAG	30
	31
MPI_COMM_DUP_WITH_INFO(COMM, INFO, NEWCOMM, IERROR)	32
INTEGER COMM, INFO, NEWCOMM, IERROR	33 34
MPI_COMM_FREE(COMM, IERROR)	35
INTEGER COMM, IERROR	36
MPI_COMM_FREE_KEYVAL(COMM_KEYVAL, IERROR)	37
INTEGER COMM_KEYVAL, IERROR	38
MPI_COMM_GET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)	39 40
INTEGER COMM, COMM_KEYVAL, IERROR	40 41
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL	42
LOGICAL FLAG	43
MPI_COMM_GET_INFO(COMM, INFO_USED, IERROR)	44
INTEGER COMM, INFO_USED, IERROR	45 46
MPI_COMM_GET_NAME(COMM, COMM_NAME, RESULTLEN, IERROR)	40 47
INTEGER COMM, RESULTLEN, IERROR	48

1	CHARACTER*(*) COMM_NAME
$\frac{2}{3}$	MPI_COMM_GROUP(COMM, GROUP, IERROR)
4	INTEGER COMM, GROUP, IERROR
5 6	MPI_COMM_IDUP(COMM, NEWCOMM, REQUEST, IERROR)
7	INTEGER COMM, NEWCOMM, REQUEST, IERROR
8 9	MPI_COMM_IDUP_WITH_INFO(COMM, INFO, NEWCOMM, REQUEST, IERROR) INTEGER COMM, INFO, NEWCOMM, REQUEST, IERROR
10 11	MPI_COMM_NULL_COPY_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
12	ATTRIBUTE_VAL_OUT, FLAG, IERROR) INTEGER OLDCOMM, COMM_KEYVAL, IERROR
13	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
14 15	ATTRIBUTE_VAL_OUT
16	LOGICAL FLAG
17	MPI_COMM_NULL_DELETE_FN(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE,
18 19	IERROR) INTEGER COMM, COMM_KEYVAL, IERROR
20	INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
21	MPI_COMM_RANK(COMM, RANK, IERROR)
22 23	INTEGER COMM, RANK, IERROR
23 24	MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR)
25	INTEGER COMM, GROUP, IERROR
26	MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR)
27 28	INTEGER COMM, SIZE, IERROR
29	MPI_COMM_SET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, IERROR)
30 31	INTEGER COMM, COMM_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
32	
33	MPI_COMM_SET_INFO(COMM, INFO, IERROR) INTEGER COMM, INFO, IERROR
34 35	MPI_COMM_SET_NAME(COMM, COMM_NAME, IERROR)
36	INTEGER COMM, IERROR
37	CHARACTER*(*) COMM_NAME
38 39	MPI_COMM_SIZE(COMM, SIZE, IERROR)
40	INTEGER COMM, SIZE, IERROR
41	MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
42	INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR
$43 \\ 44$	MPI_COMM_SPLIT_TYPE(COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR)
45	INTEGER COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR
46	MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)
47 48	INTEGER COMM, IERROR LOGICAL FLAG
	LOGICAL I BAG

A.5. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE	1013
MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR) INTEGER GROUP1, GROUP2, RESULT, IERROR	1 2
MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR) INTEGER GROUP1, GROUP2, NEWGROUP, IERROR	3 4 5
MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR) INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR	6 7 8
MPI_GROUP_FREE(GROUP, IERROR) INTEGER GROUP, IERROR	8 9 10
MPI_GROUP_FROM_SESSION_PSET(SESSION, PSET_NAME, NEWGROUP, IERROR) INTEGER SESSION, NEWGROUP, IERROR CHARACTER*(*) PSET_NAME	11 12 13 14
MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR) INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR	14 15 16
MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR) INTEGER GROUP1, GROUP2, NEWGROUP, IERROR	17 18 19
<pre>MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR) INTEGER GROUP, N, RANGES(3, *), NEWGROUP, IERROR</pre>	20 21
<pre>MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR) INTEGER GROUP, N, RANGES(3, *), NEWGROUP, IERROR</pre>	22 23 24
MPI_GROUP_RANK(GROUP, RANK, IERROR) INTEGER GROUP, RANK, IERROR	25 26
MPI_GROUP_SIZE(GROUP, SIZE, IERROR) INTEGER GROUP, SIZE, IERROR	27 28 29
<pre>MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR) INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR</pre>	30 31
MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR) INTEGER GROUP1, GROUP2, NEWGROUP, IERROR	32 33 34
MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER TAG, NEWINTERCOMM, IERROR) INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG, NEWINTERCOMM, IERROR	, 35 36 37 38
MPI_INTERCOMM_CREATE_FROM_GROUPS(LOCAL_GROUP, LOCAL_LEADER, REMOTE_GROUP REMOTE_LEADER, STRINGTAG, INFO, ERRHANDLER, NEWINTERCOMM, IERROR)	40
INTEGER LOCAL_GROUP, LOCAL_LEADER, REMOTE_GROUP, REMOTE_LEADER, INFO ERRHANDLER, NEWINTERCOMM, IERROR CHARACTER*(*) STRINGTAG	43
MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR) INTEGER INTERCOMM, NEWINTRACOMM, IERROR LOGICAL HIGH	45 46 47 48

```
1
 MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL,
\mathbf{2}
 EXTRA_STATE, IERROR)
3
 EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN
4
 INTEGER TYPE_KEYVAL, IERROR
5
 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
6
 MPI TYPE DELETE ATTR(DATATYPE, TYPE KEYVAL, IERROR)
7
 INTEGER DATATYPE, TYPE_KEYVAL, IERROR
8
9
 MPI_TYPE_DUP_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
10
 ATTRIBUTE_VAL_OUT, FLAG, IERROR)
11
 INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
12
 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
13
 ATTRIBUTE_VAL_OUT
14
 LOGICAL FLAG
15
 MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)
16
 INTEGER TYPE_KEYVAL, IERROR
17
18
 MPI_TYPE_GET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
19
 INTEGER DATATYPE, TYPE_KEYVAL, IERROR
20
 INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
21
 LOGICAL FLAG
22
 MPI_TYPE_GET_NAME(DATATYPE, TYPE_NAME, RESULTLEN, IERROR)
23
 INTEGER DATATYPE, RESULTLEN, IERROR
24
 CHARACTER*(*) TYPE_NAME
25
26
 MPI_TYPE_NULL_COPY_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
27
 ATTRIBUTE_VAL_OUT, FLAG, IERROR)
28
 INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
29
 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
30
 ATTRIBUTE_VAL_OUT
31
 LOGICAL FLAG
32
 MPI_TYPE_NULL_DELETE_FN(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE,
33
 IERROR)
34
 INTEGER DATATYPE, TYPE_KEYVAL, IERROR
35
 INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
36
37
 MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)
38
 INTEGER DATATYPE, TYPE_KEYVAL, IERROR
39
 INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
40
 MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR)
41
 INTEGER DATATYPE, IERROR
42
 CHARACTER*(*) TYPE_NAME
43
44
 MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,
45
 EXTRA_STATE, IERROR)
46
 EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN
47
 INTEGER WIN_KEYVAL, IERROR
48
```

A.5. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE	1015
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	1
MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR)	2
INTEGER WIN, WIN_KEYVAL, IERROR	3 4
MPI_WIN_DUP_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	5
ATTRIBUTE_VAL_OUT, FLAG, IERROR)	6
INTEGER OLDWIN, WIN_KEYVAL, IERROR	7
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	8
ATTRIBUTE_VAL_OUT	9 10
LOGICAL FLAG	10
MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)	12
INTEGER WIN_KEYVAL, IERROR	13
MPI_WIN_GET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)	14
INTEGER WIN, WIN_KEYVAL, IERROR	15
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL	16
LOGICAL FLAG	17 18
MPI_WIN_GET_NAME(WIN, WIN_NAME, RESULTLEN, IERROR)	18
INTEGER WIN, RESULTLEN, IERROR	20
CHARACTER*(*) WIN_NAME	21
MPI_WIN_NULL_COPY_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	22
ATTRIBUTE_VAL_OUT, FLAG, IERROR)	23
INTEGER OLDWIN, WIN_KEYVAL, IERROR	24
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	25
ATTRIBUTE_VAL_OUT	26 27
LOGICAL FLAG	28
MPI_WIN_NULL_DELETE_FN(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IER	ROR) 29
INTEGER WIN, WIN_KEYVAL, IERROR	30
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	31
MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR)	32
INTEGER WIN, WIN_KEYVAL, IERROR	33
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL	34 35
MPI_WIN_SET_NAME(WIN, WIN_NAME, IERROR)	36
INTEGER WIN, IERROR	37
CHARACTER*(*) WIN_NAME	38
	39
A.5.6 Process Topologies Fortran Bindings	40
	41
MPI_CARTDIM_GET(COMM, NDIMS, IERROR)	42
INTEGER COMM, NDIMS, IERROR	43 44
MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)	45
INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR	46
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IER	ROR) 47
	48

1	INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
2	LOGICAL PERIODS(*), REORDER
3	NDT AND AFT ADMN NAVDING DING DEDIODA ADODDA TEDDOD
4	MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
5	INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
6	LOGICAL PERIODS(*)
7	MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)
8	INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR
9	LOGICAL PERIODS(*)
10	
11	MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
12	INTEGER COMM, COORDS(*), RANK, IERROR
13	MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)
14	INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR
15	
16	MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)
17	INTEGER COMM, NEWCOMM, IERROR
18	LOGICAL REMAIN_DIMS(*)
19	MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
20	INTEGER NNODES, NDIMS, DIMS(*), IERROR
21	
22	MPI_DIST_GRAPH_CREATE(COMM_OLD, N, SOURCES, DEGREES, DESTINATIONS, WEIGHTS,
23	INFO, REORDER, COMM_DIST_GRAPH, IERROR)
24	<pre>INTEGER COMM_OLD, N, SOURCES(*), DEGREES(*), DESTINATIONS(*),</pre>
25	WEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR
26	LOGICAL REORDER
27	MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS,
28	OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER,
29	COMM_DIST_GRAPH, IERROR)
30	INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE,
31	DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH,
32	IERROR
33	LOGICAL REORDER
34	
35	MPI_DIST_GRAPH_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, SOURCEWEIGHTS,
36	MAXOUTDEGREE, DESTINATIONS, DESTWEIGHTS, IERROR)
37	INTEGER COMM, MAXINDEGREE, SOURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE,
38	<pre>DESTINATIONS(*), DESTWEIGHTS(*), IERROR</pre>
39	MPI_DIST_GRAPH_NEIGHBORS_COUNT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR)
40	INTEGER COMM, INDEGREE, OUTDEGREE, IERROR
41	LOGICAL WEIGHTED
42	LOGICAL WEIGHTED
43	MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR)
44	INTEGER COMM, NNODES, NEDGES, IERROR
45	
46	MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH,
47	IERROR) INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR
48	INIEGER COURT_OED, NNODED, INDER(*), EDGED(*), COURT_GRAFH, IERROR

LOGICAL REORDER

MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR) INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR	2 3 4
MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR) INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR	5 6
MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR) INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR	7 8 9
MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR) INTEGER COMM, RANK, NNEIGHBORS, IERROR	10 11
<pre>MPI_INEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR</type></pre>	12 13 14 15 16
<pre>MPI_INEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, REQUEST, IERROR</type></pre>	17 18 19 20 21
<pre>MPI_INEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR</type></pre>	22 23 24 25 26
<pre>MPI_INEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,</pre>	27 28 29 30 31
<pre>MPI_INEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,</pre>	32 33 34 35 36 37 38
MPI_NEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR</type>	39 40 41 42
<pre>MPI_NEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,</pre>	43 44 45 46 47 48

Unofficial Draft for Comment Only

1MPI_NEIGHBOR_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,  $\mathbf{2}$ RECVCOUNTS, DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR) 3 <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, 4 5INFO, REQUEST, IERROR 6 MPI NEIGHBOR ALLGATHER INIT (SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, 7 RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR) 8 <type> SENDBUF(*), RECVBUF(*) 9 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, 10 IERROR 11 12MPI_NEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 13RECVTYPE, COMM, IERROR) 14<type> SENDBUF(*), RECVBUF(*) 15INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR 16MPI_NEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, 17RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR) 18 <type> SENDBUF(*), RECVBUF(*) 19 INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), 20RECVTYPE, COMM, IERROR 2122MPI_NEIGHBOR_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, 23RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST, 24IERROR) 25<type> SENDBUF(*), RECVBUF(*) 26INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), 27RECVTYPE, COMM, INFO, REQUEST, IERROR 28MPI_NEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, 29 RECVCOUNTS, RDISPLS, RECVTYPES, COMM, IERROR) 30 <type> SENDBUF(*), RECVBUF(*) 31INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM, 32 IERROR 33 INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*) 34 35 MPI_NEIGHBOR_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, 36 RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST, 37 IERROR) 38 <type> SENDBUF(*), RECVBUF(*) 39 INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM, 40INFO, REQUEST, IERROR 41 INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*) 42MPI_NEIGHBOR_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, 43 RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR) 44 <tvpe> SENDBUF(*), RECVBUF(*) 45 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, 46 IERROR 47 48

A.5. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE	1019
MPI_TOPO_TEST(COMM, STATUS, IERROR) INTEGER COMM, STATUS, IERROR	1 2 3
A.5.7 MPI Environmental Management Fortran Bindings	4 5
DOUBLE PRECISION MPI_WTICK()	6
DOUBLE PRECISION MPI_WTIME()	7 8
MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR) INTEGER ERRORCLASS, IERROR	9 10 11
MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR) INTEGER ERRORCLASS, ERRORCODE, IERROR	11 12 13
MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR) INTEGER ERRORCODE, IERROR CHARACTER*(*) STRING	14 15 16 17
MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR) INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR INTEGER INFO, IERROR	18 19 20
If the Fortran compiler provides TYPE(C_PTR), then overloaded by: INTERFACE MPI_ALLOC_MEM SUBROUTINE MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)	21 22 23 24
IMPORT :: MPI_ADDRESS_KIND INTEGER :: INFO, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR	24 25 26 27
END SUBROUTINE SUBROUTINE MPI_ALLOC_MEM_CPTR(SIZE, INFO, BASEPTR, IERROR) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	27 28 29
IMPORT :: MPI_ADDRESS_KIND	30 31
INTEGER :: INFO, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE	32
TYPE(C_PTR) :: BASEPTR	33 34
END SUBROUTINE	34
END INTERFACE	36
MPI_COMM_CALL_ERRHANDLER(COMM, ERRORCODE, IERROR)	37
INTEGER COMM, ERRORCODE, IERROR	38
MPI_COMM_CREATE_ERRHANDLER(COMM_ERRHANDLER_FN, ERRHANDLER, IERROR) EXTERNAL COMM_ERRHANDLER_FN INTEGER ERRHANDLER, IERROR	39 40 41
	42
MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR) INTEGER COMM, ERRHANDLER, IERROR	43 44
	45
MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR) INTEGER COMM, ERRHANDLER, IERROR	46 47
MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)	48

```
1
 INTEGER ERRHANDLER, IERROR
\mathbf{2}
 MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
3
 INTEGER ERRORCODE, ERRORCLASS, IERROR
4
5
 MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)
6
 INTEGER ERRORCODE, RESULTLEN, IERROR
7
 CHARACTER*(*) STRING
8
 MPI_FILE_CALL_ERRHANDLER(FH, ERRORCODE, IERROR)
9
 INTEGER FH, ERRORCODE, IERROR
10
11
 MPI_FILE_CREATE_ERRHANDLER(FILE_ERRHANDLER_FN, ERRHANDLER, IERROR)
12
 EXTERNAL FILE_ERRHANDLER_FN
13
 INTEGER ERRHANDLER, IERROR
14
 MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
15
 INTEGER FILE, ERRHANDLER, IERROR
16
17
 MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
18
 INTEGER FILE, ERRHANDLER, IERROR
19
 MPI_FREE_MEM(BASE, IERROR)
20
 <type> BASE(*)
21
 INTEGER IERROR
22
23
 MPI_GET_LIBRARY_VERSION(VERSION, RESULTLEN, IERROR)
^{24}
 CHARACTER*(*) VERSION
25
 INTEGER RESULTLEN, IERROR
26
 MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERROR)
27
 CHARACTER*(*) NAME
28
 INTEGER RESULTLEN, IERROR
29
30
 MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)
^{31}
 INTEGER VERSION, SUBVERSION, IERROR
32
 MPI_SESSION_CALL_ERRHANDLER(SESSION, ERRORCODE, IERROR)
33
34
 INTEGER SESSION, ERRORCODE, IERROR
35
 MPI_SESSION_CREATE_ERRHANDLER(SESSION_ERRHANDLER_FN, ERRHANDLER, IERROR)
36
 EXTERNAL SESSION_ERRHANDLER_FN
37
 INTEGER ERRHANDLER, IERROR
38
 MPI_SESSION_GET_ERRHANDLER(SESSION, ERRHANDLER, IERROR)
39
40
 INTEGER SESSION, ERRHANDLER, IERROR
41
 MPI_SESSION_SET_ERRHANDLER(SESSION, ERRHANDLER, IERROR)
42
 INTEGER SESSION, ERRHANDLER, IERROR
43
44
 MPI_WIN_CALL_ERRHANDLER(WIN, ERRORCODE, IERROR)
45
 INTEGER WIN, ERRORCODE, IERROR
46
 MPI_WIN_CREATE_ERRHANDLER(WIN_ERRHANDLER_FN, ERRHANDLER, IERROR)
47
 EXTERNAL WIN_ERRHANDLER_FN
48
```

```
INTEGER ERRHANDLER, IERROR
MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
 INTEGER WIN, ERRHANDLER, IERROR
MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
 INTEGER WIN, ERRHANDLER, IERROR
A.5.8 The Info Object Fortran Bindings
 10
MPI_INFO_CREATE(INFO, IERROR)
 11
 INTEGER INFO, IERROR
 12
MPI_INFO_CREATE_ENV(INFO, IERROR)
 13
 INTEGER INFO, IERROR
 14
 15
MPI_INFO_DELETE(INFO, KEY, IERROR)
 16
 INTEGER INFO, IERROR
 17
 CHARACTER*(*) KEY
 18
MPI_INFO_DUP(INFO, NEWINFO, IERROR)
 19
 INTEGER INFO, NEWINFO, IERROR
 20
 21
MPI_INFO_FREE(INFO, IERROR)
 22
 INTEGER INFO, IERROR
 23
MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)
 ^{24}
 INTEGER INFO, VALUELEN, IERROR
 25
 26
 CHARACTER*(*) KEY, VALUE
 LOGICAL FLAG
 27
 28
MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)
 29
 INTEGER INFO, NKEYS, IERROR
 30
MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERROR)
 31
 INTEGER INFO, N, IERROR
 32
 33
 CHARACTER*(*) KEY
 34
MPI_INFO_GET_STRING(INFO, KEY, BUFLEN, VALUE, FLAG, IERROR)
 35
 INTEGER INFO, BUFLEN, IERROR
 36
 CHARACTER*(*) KEY, VALUE
 37
 LOGICAL FLAG
 38
 39
MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
 40
 INTEGER INFO, VALUELEN, IERROR
 41
 CHARACTER*(*) KEY
 42
 LOGICAL FLAG
 43
MPI_INFO_SET(INFO, KEY, VALUE, IERROR)
 44
 INTEGER INFO, IERROR
 45
 CHARACTER*(*) KEY, VALUE
 46
 47
```

1  $\mathbf{2}$ 

3

5 6

9

```
1
 A.5.9 Process Creation and Management Fortran Bindings
\mathbf{2}
 MPI_ABORT(COMM, ERRORCODE, IERROR)
3
 INTEGER COMM, ERRORCODE, IERROR
4
\mathbf{5}
 MPI_CLOSE_PORT(PORT_NAME, IERROR)
6
 CHARACTER*(*) PORT_NAME
7
 INTEGER IERROR
8
 MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
9
 CHARACTER*(*) PORT_NAME
10
 INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
11
12
 MPI_COMM_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
13
 CHARACTER*(*) PORT_NAME
14
 INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
15
 MPI_COMM_DISCONNECT(COMM, IERROR)
16
 INTEGER COMM, IERROR
17
18
 MPI_COMM_GET_PARENT(PARENT, IERROR)
19
 INTEGER PARENT, IERROR
20
 MPI_COMM_JOIN(FD, INTERCOMM, IERROR)
21
 INTEGER FD, INTERCOMM, IERROR
22
23
 MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,
^{24}
 ARRAY_OF_ERRCODES, IERROR)
25
 CHARACTER*(*) COMMAND, ARGV(*)
26
 INTEGER MAXPROCS, INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*),
27
 IERROR
28
 MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
29
 ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,
30
 ARRAY_OF_ERRCODES, IERROR)
31
 INTEGER COUNT, ARRAY_OF_MAXPROCS(*), ARRAY_OF_INFO(*), ROOT, COMM,
32
 INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR
33
 CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)
34
35
 MPI_FINALIZE(IERROR)
36
 INTEGER IERROR
37
 MPI_FINALIZED(FLAG, IERROR)
38
 LOGICAL FLAG
39
 INTEGER IERROR
40
41
 MPI_INIT(IERROR)
42
 INTEGER IERROR
43
 MPI_INITIALIZED(FLAG, IERROR)
44
45
 LOGICAL FLAG
46
 INTEGER IERROR
47
 MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)
48
```

INTEGER REQUIRED, PROVIDED, IERROR 1 2 MPI_IS_THREAD_MAIN(FLAG, IERROR) LOGICAL FLAG INTEGER IERROR MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR) CHARACTER*(*) SERVICE_NAME, PORT_NAME INTEGER INFO, IERROR MPI_OPEN_PORT(INFO, PORT_NAME, IERROR) 10 INTEGER INFO, IERROR 11 CHARACTER*(*) PORT_NAME 1213 MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR) 14CHARACTER*(*) SERVICE_NAME, PORT_NAME 15INTEGER INFO, IERROR 16MPI QUERY THREAD (PROVIDED, IERROR) 17 INTEGER PROVIDED, IERROR 18 19 MPI_SESSION_FINALIZE(SESSION, IERROR) 20INTEGER SESSION, IERROR 21MPI_SESSION_GET_INFO(SESSION, INFO_USED, IERROR) 22 INTEGER SESSION, INFO_USED, IERROR 23 24 MPI_SESSION_GET_NTH_PSET(SESSION, INFO, N, PSET_LEN, PSET_NAME, IERROR) 25INTEGER SESSION, INFO, N, PSET_LEN, IERROR 26CHARACTER*(*) PSET_NAME 27MPI_SESSION_GET_NUM_PSETS(SESSION, INFO, NPSET_NAMES, IERROR) 28INTEGER SESSION, INFO, NPSET_NAMES, IERROR 29 30 MPI_SESSION_GET_PSET_INFO(SESSION, PSET_NAME, INFO, IERROR) 31INTEGER SESSION, INFO, IERROR 32 CHARACTER*(*) PSET_NAME 33 MPI_SESSION_INIT(INFO, ERRHANDLER, SESSION, IERROR) 34 INTEGER INFO, ERRHANDLER, SESSION, IERROR 3536 MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR) 37 CHARACTER*(*) SERVICE_NAME, PORT_NAME 38 INTEGER INFO, IERROR 39 40 41 A.5.10 One-Sided Communications Fortran Bindings 42MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, 43 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR) 44 <type> ORIGIN_ADDR(*) 45INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT, 46TARGET_DATATYPE, OP, WIN, IERROR 47INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP 48

```
1
 MPI_COMPARE_AND_SWAP(ORIGIN_ADDR, COMPARE_ADDR, RESULT_ADDR, DATATYPE,
\mathbf{2}
 TARGET_RANK, TARGET_DISP, WIN, IERROR)
3
 <type> ORIGIN_ADDR(*), COMPARE_ADDR(*), RESULT_ADDR(*)
4
 INTEGER DATATYPE, TARGET_RANK, WIN, IERROR
5
 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
6
 MPI FETCH AND OP(ORIGIN ADDR, RESULT ADDR, DATATYPE, TARGET RANK,
7
 TARGET_DISP, OP, WIN, IERROR)
8
 <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
9
 INTEGER DATATYPE, TARGET_RANK, OP, WIN, IERROR
10
 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
11
12
 MPI_GET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
13
 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
14
 <type> ORIGIN_ADDR(*)
15
 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
16
 TARGET_DATATYPE, WIN, IERROR
17
 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
18
 MPI_GET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_ADDR,
19
 RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, TARGET_DISP,
20
 TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
21
 <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
22
 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
23
 TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR
24
 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
25
26
 MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
27
 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
28
 <type> ORIGIN_ADDR(*)
29
 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
30
 TARGET_DATATYPE, WIN, IERROR
31
 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
32
 MPI_RACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
33
 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
34
 IERROR)
35
 <type> ORIGIN_ADDR(*)
36
 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
37
 TARGET_DATATYPE, OP, WIN, REQUEST, IERROR
38
 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
39
40
 MPI_RGET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
41
 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST,
42
 IERROR)
43
 <type> ORIGIN_ADDR(*)
44
 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
45
 TARGET_DATATYPE, WIN, REQUEST, IERROR
46
 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
47
 MPI_RGET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE,
48
```

1 RESULT_ADDR, RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,  $\mathbf{2}$ IERROR) <type> ORIGIN_ADDR(*), RESULT_ADDR(*) 4 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE, 5TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST, 6 IERROR 7 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP 9 MPI_RPUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, 10 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST, 11 IERROR) 12<type> ORIGIN_ADDR(*) 13 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT, 14TARGET_DATATYPE, WIN, REQUEST, IERROR 15INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP 1617MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR) 18 INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR 19 20If the Fortran compiler provides TYPE(C_PTR), then overloaded by: 21INTERFACE MPI_WIN_ALLOCATE 22 SUBROUTINE MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, & 23WIN, IERROR) 24IMPORT :: MPI_ADDRESS_KIND 25INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR 26INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR 27END SUBROUTINE 28 SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, & 29 WIN, IERROR) 30 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR 31IMPORT :: MPI_ADDRESS_KIND 32 INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR 33 INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE 34 TYPE(C_PTR) :: BASEPTR 35 END SUBROUTINE 36 END INTERFACE 37 38 MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR) 39 INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR 40 INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR 41 If the Fortran compiler provides TYPE(C_PTR), then overloaded by: 42INTERFACE MPI_WIN_ALLOCATE_SHARED 43 SUBROUTINE MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, & 44BASEPTR, WIN, IERROR) 45IMPORT :: MPI_ADDRESS_KIND 46INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR 47INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR 48

```
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
 MPI_WIN_ATTACH(WIN, BASE, SIZE, IERROR)
12
 INTEGER WIN, IERROR
13
 <type> BASE(*)
14
 INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
15
16
 MPI_WIN_COMPLETE(WIN, IERROR)
17
 INTEGER WIN, IERROR
18
 MPI_WIN_CREATE(BASE, SIZE, DISP_UNIT, INFO, COMM, WIN, IERROR)
19
 <type> BASE(*)
20
 INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
21
 INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
22
23
 MPI_WIN_CREATE_DYNAMIC(INFO, COMM, WIN, IERROR)
^{24}
 INTEGER INFO, COMM, WIN, IERROR
25
 MPI_WIN_DETACH(WIN, BASE, IERROR)
26
 INTEGER WIN, IERROR
27
 <type> BASE(*)
28
29
 MPI_WIN_FENCE(ASSERT, WIN, IERROR)
30
 INTEGER ASSERT, WIN, IERROR
^{31}
32
 MPI_WIN_FLUSH(RANK, WIN, IERROR)
 INTEGER RANK, WIN, IERROR
33
34
 MPI_WIN_FLUSH_ALL(WIN, IERROR)
35
 INTEGER WIN, IERROR
36
37
 MPI_WIN_FLUSH_LOCAL(RANK, WIN, IERROR)
 INTEGER RANK, WIN, IERROR
38
39
 MPI_WIN_FLUSH_LOCAL_ALL(WIN, IERROR)
40
 INTEGER WIN, IERROR
41
42
 MPI_WIN_FREE(WIN, IERROR)
 INTEGER WIN, IERROR
43
44
 MPI_WIN_GET_GROUP(WIN, GROUP, IERROR)
45
 INTEGER WIN, GROUP, IERROR
46
47
 MPI_WIN_GET_INFO(WIN, INFO_USED, IERROR)
48
 INTEGER WIN, INFO_USED, IERROR
```

A.5. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE	1027
MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR) INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR	1 2
MPI_WIN_LOCK_ALL(ASSERT, WIN, IERROR) INTEGER ASSERT, WIN, IERROR	3 4 5
MPI_WIN_POST(GROUP, ASSERT, WIN, IERROR) INTEGER GROUP, ASSERT, WIN, IERROR	6 7 8
MPI_WIN_SET_INFO(WIN, INFO, IERROR) INTEGER WIN, INFO, IERROR	9 10
MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, BASEPTR, IERROR) INTEGER WIN, RANK, DISP_UNIT, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR	11 12 13
INTEGER(KIND-MPI_ADDRESS_KIND) SIZE, BASEPIR If the Fortran compiler provides TYPE(C_PTR), then overloaded by: INTERFACE MPI_WIN_SHARED_QUERY	14 15
SUBROUTINE MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, & BASEPTR, IERROR)	16 17 18
IMPORT :: MPI_ADDRESS_KIND INTEGER :: WIN, RANK, DISP_UNIT, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR	19 20
END SUBROUTINE SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, &	21 22 23
BASEPTR, IERROR) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR IMPORT :: MPI_ADDRESS_KIND	24 25
INTEGER :: WIN, RANK, DISP_UNIT, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE	26 27 28
TYPE(C_PTR) :: BASEPTR END SUBROUTINE END INTERFACE	29 30
MPI_WIN_START(GROUP, ASSERT, WIN, IERROR)	31 32 33
INTEGER GROUP, ASSERT, WIN, IERROR MPI_WIN_SYNC(WIN, IERROR)	34 35
INTEGER WIN, IERROR MPI_WIN_TEST(WIN, FLAG, IERROR)	36 37 38
INTEGER WIN, IERROR LOGICAL FLAG	39 40
MPI_WIN_UNLOCK(RANK, WIN, IERROR) INTEGER RANK, WIN, IERROR	41 42 43
MPI_WIN_UNLOCK_ALL(WIN, IERROR) INTEGER WIN, IERROR	44 45
MPI_WIN_WAIT(WIN, IERROR) INTEGER WIN, IERROR	46 47 48

```
A.5.11 External Interfaces Fortran Bindings
1
\mathbf{2}
 MPI_GREQUEST_COMPLETE(REQUEST, IERROR)
3
 INTEGER REQUEST, IERROR
4
\mathbf{5}
 MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST,
6
 IERROR)
7
 EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN
8
 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
9
 INTEGER REQUEST, IERROR
10
 MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR)
11
 INTEGER STATUS(MPI_STATUS_SIZE), IERROR
12
 LOGICAL FLAG
13
14
 MPI_STATUS_SET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
15
 INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
16
 MPI_STATUS_SET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)
17
 INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR
18
 INTEGER(KIND=MPI_COUNT_KIND) COUNT
19
20
21
 A.5.12 I/O Fortran Bindings
22
 MPI_CONVERSION_FN_NULL(USERBUF, DATATYPE, COUNT, FILEBUF, POSITION,
23
 EXTRA_STATE, IERROR)
24
 <TYPE> USERBUF(*), FILEBUF(*)
25
26
 INTEGER DATATYPE, COUNT, IERROR
 INTEGER(KIND=MPI_OFFSET_KIND) POSITION
27
 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
28
29
 MPI_FILE_CLOSE(FH, IERROR)
30
 INTEGER FH, IERROR
^{31}
 MPI_FILE_DELETE(FILENAME, INFO, IERROR)
32
33
 CHARACTER*(*) FILENAME
34
 INTEGER INFO, IERROR
35
 MPI_FILE_GET_AMODE(FH, AMODE, IERROR)
36
 INTEGER FH, AMODE, IERROR
37
38
 MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR)
39
 INTEGER FH, IERROR
40
 LOGICAL FLAG
41
 MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR)
42
 INTEGER FH, IERROR
43
 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP
44
45
 MPI_FILE_GET_GROUP(FH, GROUP, IERROR)
46
 INTEGER FH, GROUP, IERROR
47
 MPI_FILE_GET_INFO(FH, INFO_USED, IERROR)
48
```

1 INTEGER FH, INFO_USED, IERROR 2 MPI_FILE_GET_POSITION(FH, OFFSET, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET MPI_FILE_GET_SIZE(FH, SIZE, IERROR) 10 INTEGER FH, IERROR 11 INTEGER(KIND=MPI_OFFSET_KIND) SIZE 1213 MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERROR) 14INTEGER FH, DATATYPE, IERROR 15INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT 16MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR) 17 INTEGER FH, ETYPE, FILETYPE, IERROR 18 INTEGER(KIND=MPI_OFFSET_KIND) DISP 19 CHARACTER*(*) DATAREP 2021MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) 22 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 23<type> BUF(*) 24MPI_FILE_IREAD_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) 25INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 26<type> BUF(*) 2728 MPI_FILE_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR) 29 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 30 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET 31<type> BUF(*) 32 MPI_FILE_IREAD_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR) 33 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 34 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET 35 <type> BUF(*) 36 37 MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) 38 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 39 <type> BUF(*) 40 MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) 41 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 42<type> BUF(*) 43 44MPI_FILE_IWRITE_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) 45INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 46<type> BUF(*) 47MPI_FILE_IWRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR) 48

```
1
 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
\mathbf{2}
 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
3
 <type> BUF(*)
4
 MPI_FILE_IWRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
5
 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
6
 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
7
 <type> BUF(*)
8
9
 MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
10
 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
11
 <type> BUF(*)
12
 MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR)
13
 INTEGER COMM, AMODE, INFO, FH, IERROR
14
 CHARACTER*(*) FILENAME
15
16
 MPI_FILE_PREALLOCATE(FH, SIZE, IERROR)
17
 INTEGER FH, IERROR
18
 INTEGER(KIND=MPI_OFFSET_KIND) SIZE
19
 MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
20
 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
21
 <type> BUF(*)
22
23
 MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
24
 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
25
 <type> BUF(*)
26
 MPI_FILE_READ_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
27
 INTEGER FH, COUNT, DATATYPE, IERROR
28
 <type> BUF(*)
29
30
 MPI_FILE_READ_ALL_END(FH, BUF, STATUS, IERROR)
31
 INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
32
 <type> BUF(*)
33
 MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
34
 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
35
 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
36
 <type> BUF(*)
37
38
 MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
39
 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
40
 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
41
 <type> BUF(*)
42
 MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
43
 INTEGER FH, COUNT, DATATYPE, IERROR
44
 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
45
 <type> BUF(*)
46
47
 MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)
48
```

INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR <type> BUF(*)</type>	1 2
MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR <type> BUF(*)</type>	3 4 5 6
MPI_FILE_READ_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR) INTEGER FH, COUNT, DATATYPE, IERROR <type> BUF(*)</type>	7 8 9
<pre>MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR)     INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR     <type> BUF(*)</type></pre>	10 11 12 13
<pre>MPI_FILE_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR <type> BUF(*)</type></pre>	14 15 16
MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR) INTEGER FH, WHENCE, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	17 18 19 20
MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR) INTEGER FH, WHENCE, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	21 22 23 24
MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR) INTEGER FH, IERROR LOGICAL FLAG	24 25 26 27
MPI_FILE_SET_INFO(FH, INFO, IERROR) INTEGER FH, INFO, IERROR	28 29 30
MPI_FILE_SET_SIZE(FH, SIZE, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) SIZE	31 32 33
<pre>MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR) INTEGER FH, ETYPE, FILETYPE, INFO, IERROR INTEGER(KIND=MPI_OFFSET_KIND) DISP CHARACTER*(*) DATAREP</pre>	34 35 36 37 38
MPI_FILE_SYNC(FH, IERROR) INTEGER FH, IERROR	39 40
MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR <type> BUF(*)</type>	41 42 43 44
<pre>MPI_FILE_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)     INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR     <type> BUF(*)</type></pre>	45 46 47 48

1 2 3	MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR) INTEGER FH, COUNT, DATATYPE, IERROR <type> BUF(*)</type>
4 5 6 7	<pre>MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)     INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR     <type> BUF(*)</type></pre>
8 9 10 11	<pre>MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET <type> BUF(*)</type></pre>
12 13 14 15 16	<pre>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(*)</type></pre>
17 18 19 20	<pre>MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR) INTEGER FH, COUNT, DATATYPE, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET <type> BUF(*)</type></pre>
21 22 23 24	<pre>MPI_FILE_WRITE_AT_ALL_END(FH, BUF, STATUS, IERROR)     INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR     <type> BUF(*)</type></pre>
25 26 27 28	<pre>MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)     INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR     <type> BUF(*)</type></pre>
29 30 31	<pre>MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)     INTEGER FH, COUNT, DATATYPE, IERROR     <type> BUF(*)</type></pre>
32 33 34 35	<pre>MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)     INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR     <type> BUF(*)</type></pre>
36 37 38	<pre>MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)     INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR     <type> BUF(*)</type></pre>
<ol> <li>39</li> <li>40</li> <li>41</li> <li>42</li> <li>43</li> <li>44</li> <li>45</li> <li>46</li> <li>47</li> <li>48</li> </ol>	<pre>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</pre>

A.5.13 Language Bindings Fortran Bindings MPI_F_SYNC_REG(BUF) <type> BUF(*) MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR) INTEGER P, R, NEWTYPE, IERROR MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR) INTEGER R, NEWTYPE, IERROR MPI_TYPE_CREATE_F90_REAL(P, R, NEWTYPE, IERROR) INTEGER P, R, NEWTYPE, IERROR MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, DATATYPE, IERROR) INTEGER TYPECLASS, SIZE, DATATYPE, IERROR A.5.14 Tools / Profiling Interface Fortran Bindings MPI_PCONTROL(LEVEL) INTEGER LEVEL 21A.5.15 Deprecated Fortran Bindings 22 23MPI_ATTR_DELETE(COMM, KEYVAL, IERROR) INTEGER COMM, KEYVAL, IERROR MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR) INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR 27LOGICAL FLAG 2829 MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR) 30 INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR MPI_DUP_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR) 33 INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN. 34 ATTRIBUTE_VAL_OUT, IERR 35LOGICAL FLAG 36 37 MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR) INTEGER INFO, VALUELEN, IERROR CHARACTER*(*) KEY, VALUE LOGICAL FLAG MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR) 42INTEGER INFO, VALUELEN, IERROR CHARACTER*(*) KEY LOGICAL FLAG MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)

EXTERNAL COPY_FN, DELETE_FN

1 2

5

6

10

11

12

13

141516

17

18

19 20

 24 

25

26

31

32

38

39

40

41

43

44

4546

47

```
1
 INTEGER KEYVAL, EXTRA_STATE, IERROR
\mathbf{2}
 MPI_KEYVAL_FREE(KEYVAL, IERROR)
3
 INTEGER KEYVAL, IERROR
4
\mathbf{5}
 MPI_NULL_COPY_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
6
 ATTRIBUTE_VAL_OUT, FLAG, IERR)
\overline{7}
 INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
8
 ATTRIBUTE_VAL_OUT, IERR
9
 LOGICAL FLAG
10
 MPI_NULL_DELETE_FN(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)
11
 INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR
12
13
 MPI_SIZEOF(X, SIZE, IERROR)
14
 <type> X
15
 INTEGER SIZE, IERROR
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
```

## 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.5 were already introduced in the corresponding sections in previous versions of this standard.

B.1	Changes from Version 3.1 to Version 4.0	24
B.1.1	Fixes to Errata in Previous Versions of MPI	25
D.1.1	Fixes to Errata III Frevious versions of WFT	26
1.	Sections 8.6.1, 8.6.2 and 8.9 on pages 413, 417 and 440, and MPI-3.1 Sections 7.6.1,	27
	7.6.2 and 7.8 on pages 315, 318 and 329.	28
	$MPI_NEIGHBOR_ALLTOALL\{ V W\} \text{ and } MPI_NEIGHBOR_ALLGATHER\{ V\} \text{ for Car-}$	29
	tesian virtual grids were clarified. An advice to implementors was added to illustrate	30
	a correct implementation for the case of $periods[d] == 1$ or .TRUE. and $dims[d] == 1$ or	31
	2 in a direction d.	32
0		33
Ζ.	Section 19.3.5 on page 836, and MPI-3.1 Section 17.2.5 on page 657 line 11.	34
	Clarified that the MPI_STATUS_F2F08 and MPI_STATUS_F082F routines and the declaration for TVPE(MPI_Status) are not supposed to appear with maif h	35
	declaration for TYPE(MPI_Status) are not supposed to appear with mpif.h.	36
3.	Sections 2.5.4, 19.3.5, and A.1.1 on pages 18, 836, and 852, and MPI-3.1 Sections	37
	2.5.4, 17.2.5, and A.1.1 on pages 15, 656, and 669.	38
	Define the C constants MPI_F_STATUS_SIZE, MPI_F_SOURCE, MPI_F_TAG, and	39
	MPI_F_ERROR.	40 41
		41 42
4.	Section 19.3.5 on page 837, and MPI-3.1 Section 17.2.5 on page 658.	42
	Added missing const to IN parameters for MPI_STATUS_F2F08 and	43 44
	MPI_STATUS_F082F.	44 45
		40
		40
		48

1	B.1.2	2 Changes in MPI-4.0
2 3 4 5	1.	Sections 2.2, 18.1, and 19.1.5 on pages 9, 781, and 790. The limit for the maximum length of MPI identifiers was removed. This change is not backward compatible.
6 7 8 9	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, 47, 61, 68, 82, 85, 274, 680, and 873. The semantic terms were updated.
10 11 12	3.	Throughout the entire document. New large count functions and callbacks were introduced to accomodate large buffers and/or datatypes.
13 14 15 16		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.
17		A new error class MPI_ERR_VALUE_TOO_LARGE was introduced.
<ol> <li>18</li> <li>19</li> <li>20</li> <li>21</li> <li>22</li> <li>23</li> </ol>	4.	Sections 2.8, 9.3, 9.5, and 11.2.1 on pages 24, 454, 469, and 484. 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.
24 25 26	5.	Section 3.7 on page 59. The introduction of MPI nonblocking communication was changed to describe correctness and performance reasons for the use of nonblocking communication.
27 28 29	6.	Sections 3.7 and 3.9 on pages 61 and 92. Addition of MPI_ISENDRECV and MPI_ISENDRECV_REPLACE.
30 31 32	7.	Sections 3.7.3, 3.9, 6.13, 8.8, and 8.9 on pages 68, 92, 274, 432, and 440. Persistent collective communication and persistent neighborhood communication were added to the standard.
33 34 35 36	8.	Sections 3.8.4 and 16.3 on pages 90 and 776. Cancelling a send request by calling MPI_CANCEL has been deprecated and may be removed in a future version of the MPI specification.
37 38 39	9.	Chapter 4 on page 101. A new chapter on partitioned communication was added.
40 41 42 43	10.	Section 7.4.2 on page 323. MPI_COMM_TYPE_HW_UNGUIDED was added as a new possible value for the split_type parameter of the MPI_COMM_SPLIT_TYPE function.
44 45 46 47 48	11.	Section 7.4.2 on page 323. 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 323. The functions MPI_COMM_DUP and MPI_COMM_IDUP were updated to no longer propagate info hints. This change may affect backward compatibility.	1 2 3 4
13.	Section 7.4.2 on page 323. The MPI_COMM_IDUP_WITH_INFO function was added.	5 6 7
14.	Sections 7.4.4, 12.2.7, and 14.2.8 on pages 341, 561, and 645. The definition of info hints was updated to allow applications to provide assertions regarding their usage of MPI objects and operations.	8 9 10 11
15.	Section 7.4.4 on page 341. The new info hints "mpi_assert_no_any_tag", "mpi_assert_no_any_source", "mpi_assert_exact_length", and "mpi_assert_allow_overtaking" were added for use with communicators.	12 13 14 15
16.	Sections 7.4.4, 12.2.7, and 14.2.8 on pages 341, 561, and 645. The semantics of the MPI_COMM_SET_INFO, MPI_COMM_GET_INFO, MPI_WIN_SET_INFO, MPI_WIN_GET_INFO, MPI_FILE_SET_INFO, and MPI_FILE_GET_INFO were clarified.	16 17 18 19 20
17.	Section 8.5 on page 388. MPI_DIMS_CREATE is now guaranteed to return MPI_SUCCESS if the number of di- mensions passed to the routine is set to 0 and the number of nodes is set to 1.	21 22 23 24
18.	Sections 9.2, 12.2.2, and 12.2.3 on pages 451, 550, and 552. Introduced alignment requirements for memory allocated through MPI_ALLOC_MEM, MPI_WIN_ALLOCATE, and MPI_WIN_ALLOCATE_SHARED and added a new info key "mpi_minimum_memory_alignment" to specify a desired alternative minimum alignment.	25 26 27 28
19.	Sections 9.3 and 9.4 on pages 454 and 466. Clarified definition of errors to say that MPI should continue whenever possible and allow the user to recover from errors.	29 30 31 32
20.	Section 9.4 on page 466. Added text to clarify what is implied about the status of MPI and user visible buffers when MPI functions return MPI_SUCCESS or other error codes.	33 34 35 36
21.	Section 9.4 on page 467. The error class MPI_ERR_PROC_ABORTED has been added.	37 38
22.	Section 10 on page 475. Added a new function MPI_INFO_GET_STRING that takes a buffer length argument for returning info value strings. This function returns the required buffer length for the requested string and guarantees null termination for C strings where buffer size is greater than 0.	<ul> <li>39</li> <li>40</li> <li>41</li> <li>42</li> <li>43</li> <li>44</li> </ul>
23.	Section 10 on page 475 and Section 16.3 on page 776. MPI_INFO_GET and MPI_INFO_GET_VALUELEN were deprecated.	45 46 47 48

1 2 3 4	24.	Chapter 11, Sections 2.8, $3.2.3$ , $7.2.4$ , $7.3.2$ , $7.4.2$ , $7.6.2$ , $9.1.1$ , $9.1.2$ , $9.3$ , $9.3.4$ , $9.5$ , $11.6$ , $14.2.1$ , $14.2.7$ , $14.7$ , $15.3.4$ , $19.3.4$ , $19.3.6$ , and Annex A on pages $483$ , $24$ , $33$ , $311$ , $314$ , $323$ , $354$ , $447$ , $449$ , $454$ , $463$ , $469$ , $512$ , $637$ , $643$ , $711$ , $726$ , $833$ , $838$ , and $849$ The Sessions Model was added to the standard.
5 6 7	25.	Section 11.2.1 on page 484. A new function MPI_INFO_CREATE_ENV was added.
8 9 10 11	26.	Sections 11.2.1 and 11.10.4 on pages 484 and 541. Clarified the semantic of failure and error reporting before (and during) MPI_INIT and after MPI_FINALIZE.
12 13 14 15 16	27.	Section 11.8.4 on page 525. 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
17 18 19	28.	Section 12.5.3 on page 596. RMA passive target synchronization using locks can now be used portably in memory allocated via MPI_WIN_ALLOCATE_SHARED.
20 21 22 23 24	29.	Section 13.3 on page 632 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).
25 26 27	30.	Section 15.3.8 on page 749. A callback-driven event interface was added to the MPI tool information interface.
28 29 30	31.	Section 15.3.10, Table 15.7, and Section 16.3 on pages 769, 771, and 776. MPI_T_ERR_INVALID_ITEM is deprecated. MPI routines should return MPI_T_ERR_INVALID_INDEX instead of MPI_T_ERR_INVALID_ITEM.
31 32 33	32.	Section 16.3 on page 777. MP1_SIZEOF was deprecated.
34 35 36 37 38	33.	Section 19.1.5 on page 790. An exception was added for the specific Fortran names in the case of TS 29113 interface specifications in mpif.h for MPI_NEIGHBOR_ALLTOALLW_INIT, MPI_NEIGHBOR_ALLTOALLV_INIT, and MPI_NEIGHBOR_ALLGATHERV_INIT.
39 40	B.2	Changes from Version 3.0 to Version 3.1
41 42	B.2.1	Fixes to Errata in Previous Versions of MPI
43 44 45 46 47 48	1.	Chapters 3–19, Annex A.4 on page 911, and Example 6.21 on page 236, and MPI-3.0 Chapters 3–17, Annex A.3 on page 707, and Example 5.21 on page 187. Within the mpi_f08 Fortran support method, BIND(C) was removed from all SUBROUTINE, FUNCTION, and ABSTRACT INTERFACE definitions.

2.	Section 3.2.5 on page 36, and MPI-3.0 Section 3.2.5 on page 30. The three public fields MPI_SOURCE, MPI_TAG, and MPI_ERROR of the Fortran derived type TYPE(MPI_Status) must be of type INTEGER.	1 2 3
3.	Section 3.8.2 on page 85, and MPI-3.0 Section 3.8.2 on page 67. The flag arguments of the Fortran interfaces of MPI_IMPROBE were originally incorrectly defined as INTEGER (instead as LOGICAL).	4 5 6 7
4.	Section 7.4.2 on page 323, and MPI-3.0 Section 6.4.2 on page 237. In the mpi_f08 binding of MPI_COMM_IDUP, the output argument newcomm is declared as ASYNCHRONOUS.	8 9 10 11
5.	Section 7.4.4 on page 341, and MPI-3.0 Section 6.4.4 on page 248. In the mpi_f08 binding of MPI_COMM_SET_INFO, the intent of comm is IN, and the optional output argument ierror was missing.	12 13 14
6.	Section 8.6 on page 412, and MPI-3.0 Sections 7.6, on pages 314. In the case of virtual general graph topolgies (created with MPI_CART_CREATE), the use of neighborhood collective communication is restricted to adjacency matrices with the number of edges between any two processes is defined to be the same for both processes (i.e., with a symmetric adjacency matrix).	15 16 17 18 19 20
7.	Section 9.1.1 on page 447, and MPI-3.0 Section 8.1.1 on page 335. In the mpi_f08 binding of MPI_GET_LIBRARY_VERSION, a typo in the resultlen argument was corrected.	21 22 23 24
8.	Sections 9.2 (MPI_ALLOC_MEM and MPI_ALLOC_MEM_CPTR), 12.2.2 (MPI_WIN_ALLOCATE and MPI_WIN_ALLOCATE_CPTR), 12.2.3 (MPI_WIN_ALLOCATE_SHARED and MPI_WIN_ALLOCATE_SHARED_CPTR), 12.2.3 (MPI_WIN_SHARED_QUERY and MPI_WIN_SHARED_QUERY_CPTR), 15.2.1 and 15.2.6 (Profiling interface), and corresponding sections in MPI-3.0. The linker name concept was substituted by defining specific procedure names.	25 26 27 28 29 30
9.	Section 12.2.1 on page 547, and MPI-3.0 Section 11.2.2 on page 407. The "same_size" info key can be used with all window flavors, and requires that all processes in the process group of the communicator have provided this info key with the same value.	31 32 33 34 35
10.	Section 12.3.4 on page 570, 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.	36 37 38
11.	Section 12.3.4 on page 570, 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.	39 40 41 42
12.	Section 15.3 on page 723, 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.	43 44 45 46
13.	Section 15.3.3 on page 725, and MPI-3.0 Section 14.3.3 on page 563. New paragraph clarifying variable name equivalence in the tools information interface.	47 48

1 2 3 4 5 6	14.	Sections 15.3.6, 15.3.7, and 15.3.9 on pages 730, 736, and 764, 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.
7 8 9	15.	Section 15.3.7 on page 736, and MPI-3.0 Section 14.3.7 on page 573. Clarify return code of MPI_T_PVAR_{START,STOP,RESET} routines.
10 11 12	16.	Section 15.3.7 on page 736, 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.
13 14 15	17.	Section 19.1.4 on page 789, 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.
16 17 18 19	18.	Section 19.1.5 on page 790, 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.
20 21 22	19.	Section 19.1.6 on page 795, and MPI-3.0 Section 17.1.6 on page 611. The requirements on BIND(C) procedure interfaces were removed.
23 24 25 26	20.	Annexes A.3, A.4, and A.5 on pages 874, 911, and 999, and MPI-3.0 Annexes A.2, A.3, and A.4 on pages 685, 707, and 756. The predefined callback MPI_CONVERSION_FN_NULL was added to all three an- nexes.
27 28 29 30 31	21.	Annex A.4.5 on page 952, 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.
32 33	B.2.2	2 Changes in MPI-3.1
34 35 36 37	1.	Sections 2.6.4 and 5.1.5 on pages 24 and 139. 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.
38 39 40 41 42 43	2.	Sections 9.1.1, 11.2.1, and 11.6 on pages 447, 484, and 512. 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 im- plementation supports threads.
44 45 46 47 48	3.	Section 12.2.1 on page 547. 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 654 and 661. Added MPI_FILE_IREAD_AT_ALL, MPI_FILE_IWRITE_AT_ALL, MPI_FILE_IREAD_ALL, and MPI_FILE_IWRITE_ALL	1 2 3
5.	Sections 15.3.6, 15.3.7, and 15.3.9 on pages 730, 736, and 764. Clarified that NULL parameters can be provided in MPI_T_{CVAR PVAR CATEGORY}_GET_INFO routines.	4 5 6 7
6.	Sections 15.3.6, 15.3.7, 15.3.9, and 15.3.10 on pages 730, 736, 764, and 769. New routines MPI_T_CVAR_GET_INDEX, MPI_T_PVAR_GET_INDEX, MPI_T_CATEGORY_GET_INDEX, were added to support retrieving indices of vari- ables and categories. The error codes MPI_T_ERR_INVALID and	8 9 10 11
B.3	MPI_T_ERR_INVALID_NAME were added to indicate invalid uses of the interface. Changes from Version 2.2 to Version 3.0	12 13 14 15
B.3.1	Fixes to Errata in Previous Versions of MPI	16 17
1.	Sections 2.6.2 and 2.6.3 on pages 22 and 22, 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.	18 19 20 21 22
	Sections 3.2.2, 6.9.2, 14.5.2 Table 14.2, and Annex A.1.1 on pages 31, 224, 694, and 849, and MPI-2.2 Sections 3.2.2, 5.9.2, 13.5.2 Table 13.2, 16.1.16 Table 16.1, and Annex A.1.1 on pages 27, 164, 433, 472 and 513 This is an MPI-2.2 erratum: New named predefined datatypes MPI_CXX_BOOL, MPI_CXX_FLOAT_COMPLEX, MPI_CXX_DOUBLE_COMPLEX, and MPI_CXX_LONG_DOUBLE_COMPLEX were added in C and Fortran corresponding to the C++ types bool, std::complex <float>, std::complex<double>, and std::complex<long double="">. These datatypes also correspond to the deprecated C++ predefined datatypes MPI::BOOL, MPI::COMPLEX, MPI::DOUBLE_COMPLEX, and MPI::LONG_DOUBLE_COMPLEX, which were removed in MPI-3.0. The non-standard C++ types Complex&lt;&gt; were substituted by the standard types std::complex&lt;&gt;</long></double></float>	23 24 25 26 27 28 29 30 31 32 33 34 35 36
3.	Sections 6.9.2 on pages 224 and MPI-2.2 Section 5.9.2, page 165, line 47. This is an MPI-2.2 erratum: MPI_C_COMPLEX was added to the "Complex" reduction group.	37 38 39
4.	Section 8.5.5 on page 399, and MPI-2.2, Section 7.5.5 on page 257, C++ interface on page 264, line 3. This is an MPI-2.2 erratum: The argument rank was removed and in/outdegree are now defined as int& indegree and int& outdegree in the C++ interface of MPI_DIST_GRAPH_NEIGHBORS_COUNT.	40 41 42 43 44 45
5.	Section 14.5.2, Table 14.2 on page 694, and MPI-2.2, Section 13.5.3, Table 13.2 on page 433.	43 46 47 48

1 2	This was an MPI-2.2 erratum: The MPI_C_BOOL "external 32" representation is corrected to a 1-byte size.
3 4 5 6	<ol> <li>MPI-2.2 Section 16.1.16 on page 471, line 45. This is an MPI-2.2 erratum: The constant MPI::_LONG_LONG should be MPI::LONG_LONG.</li> </ol>
7 8 9 10 11 12	<ol> <li>Annex A.1.1 on page 849, Table "Optional datatypes (Fortran)," and MPI-2.2, Annex A.1.1, Table on page 517, lines 34, and 37–41. This is an MPI-2.2 erratum: The C++ datatype handles MPI::INTEGER16, MPI::REAL16, MPI::F_COMPLEX4, MPI::F_COMPLEX8, MPI::F_COMPLEX16, MPI::F_COMPLEX32 were added to the table.</li> </ol>
13 14	B.3.2 Changes in MPI-3.0
15 16 17 18	<ol> <li>Section 2.6.1 on page 21, Section 17.2 on page 780 and all other chapters. The C++ bindings were removed from the standard. See errata in Section B.3.1 on page 1041 for the latest changes to the MPI C++ binding defined in MPI-2.2. This change may affect backward compatibility.</li> </ol>
19 20 21 22 23 24 25 26 27 28	<ol> <li>Section 2.6.1 on page 21, Section 16.1 on page 773 and Section 17.1 on page 779. The deprecated functions MPI_TYPE_HVECTOR, MPI_TYPE_HINDEXED, MPI_TYPE_STRUCT, MPI_ADDRESS, MPI_TYPE_EXTENT, MPI_TYPE_LB, MPI_TYPE_UB, MPI_ERRHANDLER_CREATE (and its callback function prototype MPI_Handler_function), MPI_ERRHANDLER_SET, MPI_ERRHANDLER_GET, the dep- recated special datatype handles MPI_LB, MPI_UB, and the constants MPI_COMBINER_HINDEXED_INTEGER, MPI_COMBINER_HVECTOR_INTEGER, MPI_COMBINER_STRUCT_INTEGER were removed from the standard. This change may affect backward compatibility.</li> </ol>
29 30 31 32	<ol> <li>Section 2.3 on page 10.</li> <li>Clarified parameter usage for IN parameters. C bindings are now const-correct where backward compatibility is preserved.</li> </ol>
33 34 35 36	4. Section 2.5.4 on page 18 and Section 8.5.4 on page 392. The recommended C implementation value for MPI_UNWEIGHTED changed from NULL to non-NULL. An additional weight array constant (MPI_WEIGHTS_EMPTY) was in- troduced.
37 38 39 40	<ol> <li>Section 2.5.4 on page 18 and Section 9.1.1 on page 447. Added the new routine MPI_GET_LIBRARY_VERSION to query library specific ver- sions, and the new constant MPI_MAX_LIBRARY_VERSION_STRING.</li> </ol>
41 42 43 44 45 46 47 48	<ul> <li>6. Sections 2.5.8, 3.2.2, 3.3, 6.9.2, on pages 20, 31, 33, 224, Sections 5.1, 5.1.7, 5.1.8, 5.1.11, 13.3 on pages 117, 145, 147, 151, 632, and Annex A.1.1 on page 849. New inquiry functions, MPI_TYPE_SIZE_X, MPI_TYPE_GET_EXTENT_X, MPI_TYPE_GET_TRUE_EXTENT_X, and MPI_GET_ELEMENTS_X, return their results as an MPI_Count value, which is a new type large enough to represent element counts in memory, file views, etc. A new function, MPI_STATUS_SET_ELEMENTS_X, modifies the opaque part of an MPI_Status object so that a call to MPI_GET_ELEMENTS_X returns the provided MPI_Count value (in</li> </ul>

	Fortran, INTEGER(KIND=MPI_COUNT_KIND)). The corresponding predefined datatype is MPI_COUNT.	$\frac{1}{2}$
-		3
7.	Chapter 3 on page 29 through Chapter 19 on page 783.	4
	In the C language bindings, the array-arguments' interfaces were modified to consistently use use [] instead of *.	5 6
	Exceptions are MPI_INIT, which continues to use char <b>***argv</b> (correct because of	7
	subtle rules regarding the use of the & operator with char *argv[]), and	8
	MPI_INIT_THREAD, which is changed to be consistent with MPI_INIT.	9
		10
8.	Sections 3.2.5, 5.1.5, 5.1.11, 5.2 on pages 36, 139, 151, 172.	11
	The functions MPI_GET_COUNT and MPI_GET_ELEMENTS were defined to set the	12
	count argument to MPI_UNDEFINED when that argument would overflow. The func-	13
	tions MPI_PACK_SIZE and MPI_TYPE_SIZE were defined to set the size argument	14
	to MPI_UNDEFINED when that argument would overflow. In all other MPI-2.2 rou-	15
	tines, the type and semantics of the count arguments remain unchanged, i.e., int or	16
	INTEGER.	17
		18
9.	Section 3.2.6 on page 39, and Section 3.8 on page 82.	19
	MPI_STATUS_IGNORE can be also used in MPI_IPROBE, MPI_PROBE, MPI_IMPROBE,	20
	and MPI_MPROBE.	21
10	Section 3.8 on page 82 and Section 3.10 on page 98.	22
10.	The use of MPI_PROC_NULL in probe operations was clarified. A special predefined	23
	message MPI_MESSAGE_NO_PROC was defined for the use of matching probe (i.e., the	24
	new MPI_MPROBE and MPI_IMPROBE) with MPI_PROC_NULL.	25
		26
11.	Sections 3.8.2, 3.8.3, 19.3.4, A.1.1 on pages 85, 87, 833, 849.	27
	Like MPI_PROBE and MPI_IPROBE, the new MPI_MPROBE and MPI_IMPROBE	28
	operations allow incoming messages to be queried without actually receiving them,	29
	except that MPI_MPROBE and MPI_IMPROBE provide a mechanism to receive the	30
	specific message with the new routines MPI_MRECV and MPI_IMRECV regardless of	31
	other intervening probe or receive operations. The opaque object MPI_Message, the	32
	null handle MPI_MESSAGE_NULL, and the conversion functions MPI_Message_c2f and	33
	MPI_Message_f2c were defined.	34
10	Section 5.1.2 on name 110 and Section 5.1.12 on name 155	35
12.	Section 5.1.2 on page 119 and Section 5.1.13 on page 155. The routine MPI_TYPE_CREATE_HINDEXED_BLOCK and constant	36
		37
	MPI_COMBINER_HINDEXED_BLOCK were added.	38
13.	Chapter 6 on page $185$ and Section $6.12$ on page $248$ .	39
	Added nonblocking interfaces to all collective operations.	40
		41
14.	Sections 7.4.2, 7.4.4, 12.2.7, on pages 323, 341, 561.	42
	The new routines MPI_COMM_DUP_WITH_INFO, MPI_COMM_SET_INFO,	43
	MPI_COMM_GET_INFO, MPI_WIN_SET_INFO, and MPI_WIN_GET_INFO were	44
	added. The routine MPI_COMM_DUP must also duplicate info hints.	45
15	Section 7.4.2 on page 323.	46
10.	Added MPI_COMM_IDUP.	47
		48

1 2 3 4	16.	Section 7.4.2 on page 323. Added the new communicator construction routine MPI_COMM_CREATE_GROUP, which is invoked only by the processes in the group of the new communicator being constructed.
5 6 7 8	17.	Section 7.4.2 on page 323. Added the MPI_COMM_SPLIT_TYPE routine and the communicator split type con- stant MPI_COMM_TYPE_SHARED.
9 10 11 12 13	18.	Section 7.6.2 on page 354. In MPI-2.2, communication involved in an MPI_INTERCOMM_CREATE operation could interfere with point-to-point communication on the parent communicator with the same tag or MPI_ANY_TAG. This interference has been removed in MPI-3.0.
14 15 16	19.	Section 7.8 on page 377. Section 6.8 on page 238. The constant MPI_MAX_OBJECT_NAME also applies for type and window names.
17 18 19	20.	Section 8.5.8 on page 410. MPI_CART_MAP can also be used for a zero-dimensional topologies.
20 21 22 23 24 25 26 27 28 29 30 31		Section 8.6 on page 412 and Section 8.7 on page 425. The following neighborhood collective communication routines were added to support sparse communication on virtual topology grids: MPI_NEIGHBOR_ALLGATHER, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL, MPI_NEIGHBOR_ALLTOALLV, MPI_NEIGHBOR_ALLTOALLW and the nonblocking variants MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLTOALLW. The displacement arguments in MPI_NEIGHBOR_ALLTOALLW and MPI_INEIGHBOR_ALLTOALLW were defined as address size integers. In MPI_DIST_GRAPH_NEIGHBORS, an ordering rule was added for communicators created with MPI_DIST_GRAPH_CREATE_ADJACENT.
32 33 34 35	22.	Section 11.2.1 on page 484 and Section 11.2.1 on page 487. The use of MPI_INIT, MPI_INIT_THREAD and MPI_FINALIZE was clarified. After MPI is initialized, the application can access information about the execution envi- ronment by querying the new predefined info object MPI_INFO_ENV.
36 37 38	23.	Section 11.2.1 on page 484. Allow calls to MPI_T routines before MPI_INIT and after MPI_FINALIZE.
<ul> <li>39</li> <li>40</li> <li>41</li> <li>42</li> </ul>	24.	Chapter 12 on page 545. Substantial revision of the entire One-sided chapter, with new routines for window creation, additional synchronization methods in passive target communication, new one-sided communication routines, a new memory model, and other changes.
43 44	25.	Section 15.3 on page 723. A new MPI Tool Information Interface was added.
45 46		The following changes are related to the Fortran language support.
47 48	26.	Section 2.3 on page 10, and Sections 19.1.1, 19.1.2, 19.1.7 on pages 783, 784, and 799. The new mpi_08 Fortran module was introduced.

- 27. Section 2.5.1 on page 16, and Sections 19.1.2, 19.1.3, 19.1.7 on pages 784, 787, and 799. Handles to opaque objects were defined as named types within the mpi_08 Fortran module. The operators .EQ., .NE., ==, and /= were overloaded to allow the comparison of these handles. The handle types and the overloaded operators are also available through the mpi Fortran module.
- 28. Sections 2.5.4, 2.5.5 on pages 18, 19, Sections 19.1.1, 19.1.10, 19.1.11, 19.1.12, 19.1.13 on pages 783, 809, 811, 811, 814, and Sections 19.1.2, 19.1.3, 19.1.7 on pages 784, 787, 799.

Within the mpi_08 Fortran module, choice buffers were defined as assumed-type and assumed-rank according to Fortran 2008 TS 29113 [46], and the compile-time constant MPI_SUBARRAYS_SUPPORTED was set to .TRUE.. With this, Fortran subscript triplets can be used in nonblocking MPI operations; vector subscripts are not supported in nonblocking operations. If the compiler does not support this Fortran TS 29113 feature, the constant is set to .FALSE..

- 29. Section 2.6.2 on page 22, Section 19.1.2 on page 784, and Section 19.1.7 on page 799. The ierror dummy arguments are OPTIONAL within the mpi_08 Fortran module.
- 30. Section 3.2.5 on page 36, Sections 19.1.2, 19.1.3, 19.1.7, on pages 784, 787, 799, and Section 19.3.5 on page 835.
  Within the mpi_08 Fortran module, the status was defined as TYPE(MPI_Status). Additionally, within both the mpi and the mpi_f08 modules, the constants
  MPI_STATUS_SIZE, MPI_SOURCE, MPI_TAG, MPI_ERROR, and TYPE(MPI_Status) are defined. New conversion routines were added: MPI_STATUS_F2F08, MPI_STATUS_F082F, MPI_Status_c2f08, and MPI_Status_f082c, In mpi.h, the new type MPI_F08_status, and the external variables MPI_F08_STATUS_IGNORE and MPI_F08_STATUSES_IGNORE were added.
- 31. Section 3.6 on page 56.

In Fortran with the mpi module or mpif.h, the type of the buffer_addr argument of MPI_BUFFER_DETACH is incorrectly defined and the argument is therefore unused.

- 32. Section 5.1 on page 117, Section 5.1.6 on page 142, and Section 19.1.15 on page 815. The Fortran alignments of basic datatypes within Fortran derived types are implementation dependent; therefore it is recommended to use the BIND(C) attribute for derived types in MPI communication buffers. If an array of structures (in C/C++) or derived types (in Fortran) is to be used in MPI communication buffers, it is recommended that the user creates a portable datatype handle and additionally applies MPI_TYPE_CREATE_RESIZED to this datatype handle.
- 33. Sections 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 150, 231, 238, 372, 377, 457, 459, 461, 773, and 801. 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_SET_NAME,

 $\mathbf{2}$ 

 $\overline{7}$ 

 $45 \\ 46$ 

1 2 3 4 5 6 7 8	MPI_TYPE_GET_NAME, MPI_TYPE_MATCH_SIZE, the callback prototype defini- tion 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 consis- tency reasons, INOUBUF was changed to INOUTBUF in MPI_REDUCE_LOCAL, and intracomm to newintracomm in MPI_INTERCOMM_MERGE.
9 34. 10 11 12	Section 7.7.2 on page 362. 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.
¹³ 35. ¹⁴ ¹⁵ ¹⁶ ¹⁷	Section 9.2 on page 451. With the mpi and mpi_f08 Fortran modules, MPI_ALLOC_MEM now also supports TYPE(C_PTR) C-pointers instead of only returning an address-sized integer that may be usable together with a non-standard Cray-pointer.
19 20	Section 19.1.15 on page 815, and Section 19.1.7 on page 799. Fortran SEQUENCE and BIND(C) derived application types can now be used as buffers in MPI operations.
21 22 37. 23 24 25 26 27 28 29	Section 19.1.16 on page 817 to Section 19.1.19 on page 826, Section 19.1.7 on page 799, and Section 19.1.8 on page 800. 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.
$30 \\ 31 \\ 32 \\ 33$	Section 19.1.2 on page 784. Within the mpi_08 Fortran module, dummy arguments are now declared with INTENT=IN, OUT, or INOUT as defined in the mpi_08 interfaces.
³⁴ 39. ³⁵ ³⁶	Section 19.1.3 on page 787, and Section 19.1.7 on page 799. The existing mpi Fortran module must implement compile-time argument checking.
³⁷ 40.	Section 19.1.4 on page 789. The use of the mpif.h Fortran include file is now strongly discouraged.
<ul> <li>39</li> <li>40</li> <li>41</li> <li>42</li> <li>43</li> <li>44</li> </ul>	Section A.1.1, Table " <i>Predefined functions</i> " on page 857, Section A.1.3 on page 864, and Section A.4.5 on page 952. Within the new mpi_f08 module, all callback prototype definitions are now defined with explicit interfaces PROCEDURE(MPI) that have the BIND(C) attribute; user-written callbacks must be modified if the mpi_f08 module is used.
<ol> <li>45</li> <li>42.</li> <li>46</li> <li>47</li> <li>48</li> </ol>	Section A.1.3 on page 864. In some routines, the Fortran callback prototype names were changed from $\dots$ _FN to $\dots$ _FUNCTION to be consistent with the other language bindings.

B.4	Changes from Version 2.1 to Version 2.2	$\frac{1}{2}$
1.	Section 2.5.4 on page 18. It is now guaranteed that predefined named constant handles (as other constants) can be used in initialization expressions or assignments, i.e., also before the call to MPI_INIT.	2 3 4 5 6
2.	Section 2.6 on page 21, and Section 17.2 on page 780. The C++ language bindings have been deprecated and may be removed in a future version of the MPI specification.	7 8 9 10
3.	Section 3.2.2 on page 31. MPI_CHAR for printable characters is now defined for C type char (instead of signed char). This change should not have any impact on applications nor on MPI libraries (except some comment lines), because printable characters could and can be stored in any of the C types char, signed char, and unsigned char, and MPI_CHAR is not allowed for predefined reduction operations.	11 12 13 14 15 16 17
4.	Section 3.2.2 on page 31. MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_BOOL, MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, and MPI_C_LONG_DOUBLE_COMPLEX are now valid predefined MPI datatypes.	18 19 20 21
5.	Section 3.4 on page 47, Section 3.7.2 on page 61, Section 3.9 on page 92, and Section 6.1 on page 185. The read access restriction on the send buffer for blocking, non blocking and collective API has been lifted. It is permitted to access for read the send buffer while the operation is in progress.	22 23 24 25 26 27
6.	Section 3.7 on page 59. The Advice to users for IBSEND and IRSEND was slightly changed.	28 29
7.	Section 3.7.3 on page 68. The advice to free an active request was removed in the Advice to users for MPI_REQUEST_FREE.	30 31 32 33
8.	Section 3.7.6 on page 81. MPI_REQUEST_GET_STATUS changed to permit inactive or null requests as input.	34 35 36
9.	Section 6.8 on page 215. "In place" option is added to MPI_ALLTOALL, MPI_ALLTOALLV, and MPI_ALLTOALLW for intra-communicators.	37 38 39
10.	Section 6.9.2 on page 224. 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.	40 41 42 43 44
11.	Section 6.9.2 on page 224. 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	45 46 47 48

1 2 3		integer types. MPI_C_BOOL is considered a Logical type. MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, and MPI_C_LONG_DOUBLE_COMPLEX are considered Complex types.
4 5 6 7	12.	Section 6.9.7 on page 238. The local routines MPI_REDUCE_LOCAL and MPI_OP_COMMUTATIVE have been added.
8 9 10 11	13.	Section 6.10.1 on page 240. The collective function MPI_REDUCE_SCATTER_BLOCK is added to the MPI stan- dard.
11 12 13	14.	Section 6.11.2 on page 245. Added in place argument to MPI_EXSCAN.
14 15 16 17 18 19 20	15.	Section 7.4.2 on page 323, and Section 7.6 on page 351. 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.
20 21 22 23 24	16.	Section 7.4.2 on page 323. 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.
25 26 27 28 29 30	17.	Section 8.5.4 on page 392. 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.
31 32 33 34	18.	Section 8.5.5 on page 399. 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.
35 36 37 38	19.	Section 8.5.5 on page 399. Remove ambiguity regarding duplicated neighbors with MPI_GRAPH_NEIGHBORS and MPI_GRAPH_NEIGHBORS_COUNT.
39 40	20.	Section 9.1.1 on page 447. The subversion number changed from 1 to 2.
41 42 43 44	21.	Section 9.3 on page 454, Section 16.2 on page 776, and Annex A.1.3 on page 864. Changed function pointer typedef names MPI_{Comm,File,Win}_errhandler_fn to MPI_{Comm,File,Win}_errhandler_function. Deprecated old "_fn" names.
45 46 47 48	22.	Section 11.2.4 on page 494. Attribute deletion callbacks on MPI_COMM_SELF are now called in LIFO order. Imple- mentors must now also register all implementation-internal attribute deletion callbacks on MPI_COMM_SELF before returning from MPI_INIT_MPI_INIT_THREAD.

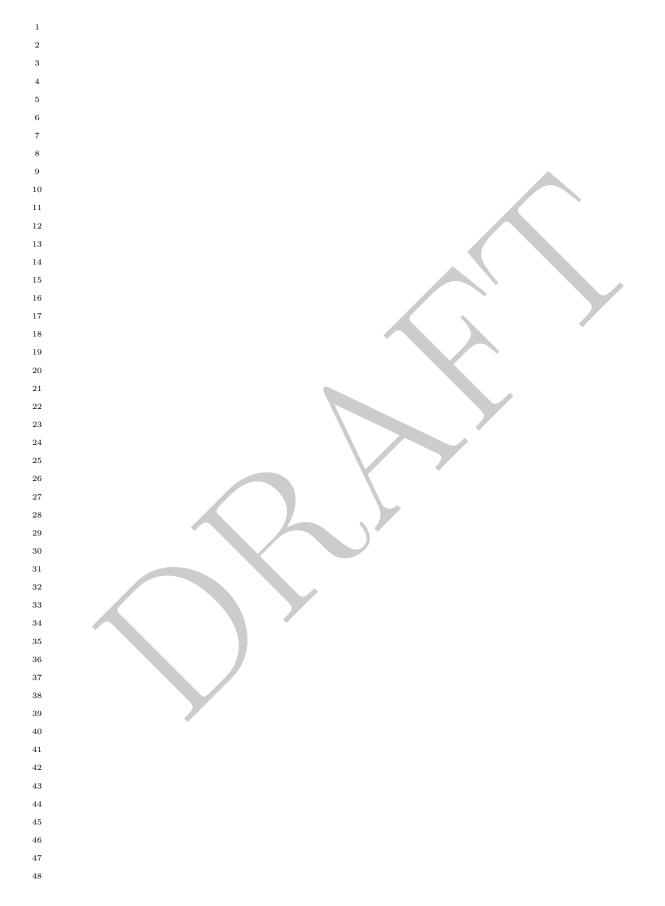
23.	Section 12.3.4 on page 570. The restriction added in MPI 2.1 that the operation MPI_REPLACE in MPI_ACCUMULATE can be used only with predefined datatypes has been removed. MPI_REPLACE can now be used even with derived datatypes, as it was in MPI 2.0. Also, a clarification has been made that MPI_REPLACE can be used only in MPI_ACCUMULATE, not in collective operations that do reductions, such as MPI_REDUCE and others.	1 2 3 4 5 6 7 8
24.	Section 13.2 on page 625. Add "*" to the query_fn, free_fn, and cancel_fn arguments to the C++ binding for MPI::Grequest::Start() for consistency with the rest of MPI functions that take function pointer arguments.	9 10 11 12
25.	Section 14.5.2 on page 693, and Table 14.2 on page 694. MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, MPI_C_LONG_DOUBLE_COMPLEX, and MPI_C_BOOL are added as predefined datatypes in the "external32" representation.	13 14 15 16 17 18
26.	Section 19.3.7 on page 841. The description was modified that it only describes how an MPI implementation behaves, but not how MPI stores attributes internally. The erroneous MPI-2.1 Example 16.17 was replaced with three new examples 19.13, 19.14, and 19.15 on pages 841–843 explicitly detailing cross-language attribute behavior. Implementations that matched the behavior of the old example will need to be updated.	19 20 21 22 23 24 25
27.	Annex A.1.1 on page 849. Removed type MPI::Fint (compare MPI_Fint in Section A.1.2 on page 863).	25 26 27
28.	Annex A.1.1 on page 849. Table <i>Named Predefined Datatypes</i> . Added MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_BOOL, MPI_C_FLOAT_COMPLEX, MPI_C_COMPLEX, MPI_C_DOUBLE_COMPLEX, and MPI_C_LONG_DOUBLE_COMPLEX are added as predefined datatypes.	28 29 30 31 32
B.5	Changes from Version 2.0 to Version 2.1	33 34
1.	Section 3.2.2 on page 31, and Annex A.1 on page 849. In addition, the MPI_LONG_LONG should be added as an optional type; it is a synonym for MPI_LONG_LONG_INT.	35 36 37 38
2.	Section 3.2.2 on page 31, and Annex A.1 on page 849. 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.	39 40 41 42 43
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.	44 45 46 47 48

1	4. Section 5.1 on page 117.
2	General rule about derived datatypes: Most datatype constructors have replication
3	count or block length arguments. Allowed values are non-negative integers. If the
4	value is zero, no elements are generated in the type map and there is no effect on
5	datatype bounds or extent.
6	datatype bounds of extent.
	5. Section 5.3 on page 180.
7	MPI_BYTE should be used to send and receive data that is packed using
8	MPI_PACK_EXTERNAL.
9	MIFI_FACK_LATERNAL.
10	6. Section 6.9.6 on page 236.
11	If comm is an inter-communicator in MPI_ALLREDUCE, then both groups should
12	provide count and datatype arguments that specify the same type signature (i.e., it is
13	not necessary that both groups provide the same count value).
14	not necessary that both groups provide the same count value).
15	7. Section 7.3.1 on page 312.
16	MPI_GROUP_TRANSLATE_RANKS and MPI_PROC_NULL: MPI_PROC_NULL is a valid
17	rank for input to MPI_GROUP_TRANSLATE_RANKS, which returns MPI_PROC_NULL
18	as the translated rank.
19	
20	8. Section 7.7 on page 361.
21	About the attribute caching functions:
22	J. J
23	Advice to implementors. High-quality implementations should raise an er-
	ror when a keyval that was created by a call to MPI_XXX_CREATE_KEYVAL
24	is used with an object of the wrong type with a call to
25	MPI_YYY_GET_ATTR, MPI_YYY_SET_ATTR, MPI_YYY_DELETE_ATTR, or
26	MPI_YYY_FREE_KEYVAL. To do so, it is necessary to maintain, with each key-
27	val, information on the type of the associated user function. (End of advice to
28	implementors.)
29	
30	9. Section 7.8 on page 377.
31	In MPI_COMM_GET_NAME: In C, a null character is additionally stored at
32	name[resultlen]. resultlen cannot be larger then MPI_MAX_OBJECT_NAME-1. In For-
33	tran, name is padded on the right with blank characters. resultlen cannot be larger
34	then MPI_MAX_OBJECT_NAME.
35	
36	10. Section 8.4 on page 387.
37	About MPI_GRAPH_CREATE and MPI_CART_CREATE: All input arguments must
38	have identical values on all processes of the group of comm_old.
39	
40	11. Section 8.5.1 on page 388.
41	In MPI_CART_CREATE: If ndims is zero then a zero-dimensional Cartesian topology
42	is created. The call is erroneous if it specifies a grid that is larger than the group size
42	or if <b>ndims</b> is negative.
44	12. Section 8.5.3 on page 390.
45	In MPI_GRAPH_CREATE: If the graph is empty, i.e., nnodes $== 0$ , then
46	MPI_COMM_NULL is returned in all processes.
47	
48	

13.	Section 8.5.3 on page 390. In MPI_GRAPH_CREATE: A single process is allowed to be defined multiple times in the list of neighbors of a process (i.e., there may be multiple edges between two processes). A process is also allowed to be a neighbor to itself (i.e., a self loop in the graph). The adjacency matrix is allowed to be non-symmetric.	1 2 3 4 5 6
	Advice to users. Performance implications of using multiple edges or a non- symmetric adjacency matrix are not defined. The definition of a node-neighbor edge does not imply a direction of the communication. ( <i>End of advice to users.</i> )	7 8 9
14.	Section 8.5.5 on page 399. In MPI_CARTDIM_GET and MPI_CART_GET: If comm is associated with a zero- dimensional Cartesian topology, MPI_CARTDIM_GET returns ndims=0 and MPI_CART_GET will keep all output arguments unchanged.	10 11 12 13 14
15.	Section 8.5.5 on page 399. In MPI_CART_RANK: If comm is associated with a zero-dimensional Cartesian topol- ogy, coord is not significant and 0 is returned in rank.	15 16 17
16.	Section 8.5.5 on page 399. In MPI_CART_COORDS: If comm is associated with a zero-dimensional Cartesian topology, coords will be unchanged.	18 19 20 21
17.	Section 8.5.6 on page 407. In MPI_CART_SHIFT: It is erroneous to call MPI_CART_SHIFT with a direction that is either negative or greater than or equal to the number of dimensions in the Cartesian communicator. This implies that it is erroneous to call MPI_CART_SHIFT with a comm that is associated with a zero-dimensional Cartesian topology.	22 23 24 25 26 27
18.	Section 8.5.7 on page 409. In MPI_CART_SUB: If all entries in remain_dims are false or comm is already associ- ated with a zero-dimensional Cartesian topology then newcomm is associated with a zero-dimensional Cartesian topology.	28 29 30 31
18.1.	Section 9.1.1 on page 447. The subversion number changed from 0 to 1.	32 33 34
19.	Section 9.1.2 on page 449. In MPI_GET_PROCESSOR_NAME: In C, a null character is additionally stored at name[resultlen]. resultlen cannot be larger then MPI_MAX_PROCESSOR_NAME-1. In Fortran, name is padded on the right with blank characters. resultlen cannot be larger then MPI_MAX_PROCESSOR_NAME.	35 36 37 38 39 40
20.	Section 9.3 on page 454. MPI_{COMM,WIN,FILE}_GET_ERRHANDLER behave as if a new error handler object is created. That is, once the error handler is no longer needed, MPI_ERRHANDLER_FREE should be called with the error handler returned from MPI_ERRHANDLER_GET or MPI_{COMM,WIN,FILE}_GET_ERRHANDLER to mark the error handler for deallocation. This provides behavior similar to that of MPI_COMM_GROUP and MPI_GROUP_FREE.	41 42 43 44 45 46 47

1	21	Section $11.2.1$ on page $484$ , see explanations to MPI_FINALIZE.
2		MPI_FINALIZE is collective over all connected processes. If no processes were spawned,
3		accepted or connected then this means over MPI_COMM_WORLD; otherwise it is col-
4		lective over the union of all processes that have been and continue to be connected,
5		as explained in Section 11.10.4 on page 541.
6		as explained in Section 11.10.4 on page 041.
7	22.	Section 11.2.1 on page 484.
8		About MPI_ABORT:
9		
10		Advice to users. Whether the errorcode is returned from the executable or from
11		the MPI process startup mechanism (e.g., mpiexec), is an aspect of quality of the
12		MPI library but not mandatory. (End of advice to users.)
13		
14		Advice to implementors. Where possible, a high-quality implementation will try
15		to return the errorcode from the MPI process startup mechanism (e.g. mpiexec
16		or singleton init). (End of advice to implementors.)
17	0.0	
18	23.	Section 10 on page 475.
19		An implementation must support info objects as caches for arbitrary (key, value)
20		pairs, regardless of whether it recognizes the key. Each function that takes hints in
21		the form of an MPI_Info must be prepared to ignore any key it does not recognize. This
22		description of info objects does not attempt to define how a particular function should
23		react if it recognizes a key but not the associated value. MPI_INFO_GET_NKEYS,
24		MPI_INFO_GET_NTHKEY, MPI_INFO_GET_VALUELEN, and MPI_INFO_GET must
25		retain all (key,value) pairs so that layered functionality can also use the lnfo object.
26	24.	Section 12.3 on page 563.
27		MPI_PROC_NULL is a valid target rank in the MPI RMA calls MPI_ACCUMULATE,
28		MPI_GET, and MPI_PUT. The effect is the same as for MPI_PROC_NULL in MPI point-
29		to-point communication. See also item 25 in this list.
30		
31	25.	Section $12.3$ on page 563.
32		After any RMA operation with rank MPI_PROC_NULL, it is still necessary to finish the
33		RMA epoch with the synchronization method that started the epoch. See also item 24
34		in this list.
35	20	
36	26.	Section 12.3.4 on page 570.
37		MPI_REPLACE in MPI_ACCUMULATE, like the other predefined operations, is defined
38		only for the predefined MPI datatypes.
39	27	Section $14.2.8$ on page $645$ .
	21.	About MPI_FILE_SET_VIEW and MPI_FILE_SET_INFO: When an info object that
40 41		specifies a subset of valid hints is passed to MPI_FILE_SET_VIEW or
41		MPI_FILE_SET_INFO, there will be no effect on previously set or defaulted hints that
42		the info does not specify.
		the fine does not speenly.
44	28.	Section 14.2.8 on page 645.
45 46		About MPI_FILE_GET_INFO: If no hint exists for the file associated with fh, a handle
		to a newly created info object is returned that contains no key/value pair.
47		

29. Section 14.3 on page 648.  $\mathbf{2}$ If a file does not have the mode MPI_MODE_SEQUENTIAL, then MPI_DISPLACEMENT_CURRENT is invalid as disp in MPI_FILE_SET_VIEW. 30. Section 14.5.2 on page 693. The bias of 16 byte doubles was defined with 10383. The correct value is 16383. 31. MPI-2.2, Section 16.1.4 (Section was removed in MPI-3.0). In the example in this section, the buffer should be declared as const void* buf. 32. Section 19.1.9 on page 801. About MPI_TYPE_CREATE_F90_XXX: Advice to implementors. An application may often repeat a call to MPI_TYPE_CREATE_F90_XXX with the same combination of (XXX,p,r). The application is not allowed to free the returned predefined, unnamed datatype handles. To prevent the creation of a potentially huge amount of handles, the MPI implementation should return the same datatype handle for the same ( REAL/COMPLEX/INTEGER, p, r) combination. Checking for the combination ( p,r) in the preceding call to MPI_TYPE_CREATE_F90_XXX and using a hash-table to find formerly generated handles should limit the overhead of finding a previously generated datatype with same combination of (XXX,p,r). (End of advice to implementors.) 33. Section A.1.1 on page 849.  24 MPI_BOTTOM is defined as void * const MPI::BOTTOM. 



# Bibliography

- [1] Reverse domain name notation convention. https://docs.oracle.com/javase/ tutorial/java/package/namingpkgs.html. 340
- [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. 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. 2
- [4] Purushotham V. Bangalore, Nathan E. Doss, and Anthony Skjellum. MPI++: Issues and Features. In *OON-SKI '94*, page in press, 1994. 307
- [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.
   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. 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. 605
- [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. 635
- [9] R. Butler and E. Lusk. User's guide to the p4 programming system. Technical Report TM-ANL-92/17, Argonne National Laboratory, 1992.
- [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.
   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. 2

47 48

1 2

6

9 10

11

12 13

14

15

16 17

18

19

20

21

22 23

 24 

25

26 27

28

29

30

 31 

32 33

34

35

36 37

38

39

40

41

42

43 44

45

46

1 2 3	[12]	S. Chittor and R. J. Enbody. Performance evaluation of mesh-connected wormhole- routed networks for interprocessor communication in multicomputers. In <i>Proceedings</i> of the 1990 Supercomputing Conference, pages 647–656, 1990. 385
4 5 6 7	[13]	S. Chittor and R. J. Enbody. Predicting the effect of mapping on the communica- tion performance of large multicomputers. In <i>Proceedings of the 1991 International</i> <i>Conference on Parallel Processing, vol. II (Software)</i> , pages II–1–II–4, 1991. 385
8 9 10	[14]	Parasoft Corporation. Express version 1.0: A communication environment for parallel computers, 1988. 2
11 12 13 14	[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. 1056, 1058
15 16 17 18 19	[16]	Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In <i>IPPS '93 Workshop on Input/Output in Parallel Computer Systems</i> , pages 56–70, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38. 635
20 21 22	[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. 330
23 24 25 26	[18]	J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Integrated PVM framework supports heterogeneous network computing. <i>Computers in Physics</i> , 7(2):166–75, April 1993. 2
27 28 29 30	[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. 2
31 32	[20]	Edinburgh Parallel Computing Centre, University of Edinburgh. CHIMP Concepts, June 1991. 2
33 34 35	[21]	Edinburgh Parallel Computing Centre, University of Edinburgh. CHIMP Version 1.0 Interface, May 1992. 2
36 37 38 39	[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. 308
40 41 42	[23]	Message Passing Interface Forum. MPI: A Message-Passing Interface standard. The In- ternational Journal of Supercomputer Applications and High Performance Computing, 8, 1994. 2
43 44 45	[24]	Message Passing Interface Forum. MPI: A Message-Passing Interface standard (version 1.1). Technical report, 1995. http://www.mpi-forum.org. 3
46 47 48	[25]	Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Bob Manchek, and Vaidy Sunderam. <i>PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Network Parallel Computing.</i> MIT Press, 1994. 483

- [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. 2
- [27] Brice Goglin, Emmanuel Jeannot, Farouk Mansouri, and Guillaume Mercier. Hardware topology management in MPI applications through hierarchical communicators. *Parallel Computing*, 76:70–90, 2018. 339
- [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. 101
- [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. 101
- [30] D. Gregor, T. Hoefler, B. Barrett, and A. Lumsdaine. Fixing probe for multi-threaded MPI applications. Technical Report 674, Indiana University, Jan. 2009. 85
- [31] William D. Gropp and Barry Smith. Chameleon parallel programming tools users manual. Technical Report ANL-93/23, Argonne National Laboratory, March 1993. 2
- [32] Michael Hennecke. A Fortran 90 interface to MPI version 1.1. Technical Report Internal Report 63/96, Rechenzentrum, Universität Karlsruhe, D-76128 Karlsruhe, Germany, June 1996. 788
- [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. 85
- [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. 248
- [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. 412
- [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. 248
- [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. 249

 $46 \\ 47$ 

1 2

3

4

5

6

7 8

9

10

11 12

13

14

15 16

17

18 19

20

21

22

23

24 25

26

27

28

29 30

 31 

32

33

34

35

36

37 38

39

40

41 42

43

44

45

1 2 3 4	[38]	T. Hoefler, M. Schellmann, S. Gorlatch, and A. Lumsdaine. Communication optimization for medical image reconstruction algorithms. In <i>Recent Advances in Parallel Virtual Machine and Message Passing Interface</i> , 15th European PVM/MPI Users' Group Meeting, volume LNCS 5205, pages 75–83. Springer, Sep. 2008. 248
5 6 7 8	[39]	T. Hoefler and J. L. Träff. Sparse collective operations for MPI. In <i>Proceedings of</i> the 23rd IEEE International Parallel & Distributed Processing Symposium, HIPS'09 Workshop, May 2009. 412
9 10 11	[40]	Torsten Hoefler and Marc Snir. Writing parallel libraries with MPI — common practice, issues, and extensions. In Cotronis et al. [15], pages 345–355. 326
12 13 14 15	[41]	Daniel J. Holmes, Bradley Morgan, Anthony Skjellum, Purushotham V. Bangalore, and Srinivas Sridharan. Planning for performance: Enhancing achievable performance for MPI through persistent collective operations. <i>Parallel Computing</i> , 81:32 – 57, 2019. 274
16 17 18	[42]	Institute of Electrical and Electronics Engineers, New York. <i>IEEE Standard for Binary Floating-Point Arithmetic, IEEE Standard 754-2008</i> , 2008. 693
19 20 21 22	[43]	International Organization for Standardization, Geneva. ISO 8859-1:1987: Informa- tion processing — 8-bit single-byte coded graphic character sets — Part 1: Latin al- phabet No. 1, February 1987. 693
23 24 25	[44]	International Organization for Standardization, Geneva. ISO/IEC 9945-1:1996: Infor- mation technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language], December 1996. 512, 639
26 27 28 29	[45]	International Organization for Standardization, Geneva. ISO/IEC 1539-1:2010: In- formation technology — Programming languages — Fortran — Part 1: Base language, November 2010. 783, 786
30 31 32 33	[46]	International Organization for Standardization, Geneva. ISO/IEC TS 29113:2012: Information technology — Further interoperability of Fortran with C, December 2012. 783, 784, 786, 799, 800, 1045
34 35	[47]	Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. <i>The High Performance Fortran Handbook</i> . MIT Press, 1993. 133
36 37 38 39	[48]	David Kotz. Disk-directed I/O for MIMD multiprocessors. In <i>Proceedings of the 1994</i> Symposium on Operating Systems Design and Implementation, pages 61–74, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994. 635
40 41 42	[49]	O. Krämer and H. Mühlenbein. Mapping strategies in message-based multiprocessor systems. <i>Parallel Computing</i> , 9:213–225, 1989. 385
43 44 45 46 47 48	[50]	S. J. Lefflet, R. S. Fabry, W. N. Joy, P. Lapsley, S. Miller, and C. Torek. An advanced 4.4BSD interprocess communication tutorial, Unix programmer's supplementary documents (PSD) 21. Technical report, Computer Systems Research Group, Depertment of Electrical Engineering and Computer Science, University of California, Berkeley, 1993. Available online: https://docs.freebsd.org/44doc/psd/21.ipc/paper.pdf. 543

[51]	Message Passing Interface Forum. Summary of the semantics of all operation-related MPI procedures, 2020. Available online: https://www.mpi-forum.org/docs. 873	1 2 3
[52]	Bradley Morgan, Daniel J. Holmes, Anthony Skjellum, Purushotham Bangalore, and Srinivas Sridharan. Planning for performance: Persistent collective operations for MPI. In <i>Proceedings of the 24th European MPI Users' Group Meeting</i> , EuroMPI '17, pages 4:1–4:11, New York, NY, USA, 2017. ACM. 274	3 4 5 6 7
[53]	nCUBE Corporation. nCUBE 2 Programmers Guide, r2.0, December 1990. 2	8 9
[54]	Bill Nitzberg. Performance of the iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, December 1992. 635	10 11
[55]	William J. Nitzberg. <i>Collective Parallel I/O</i> . PhD thesis, Department of Computer and Information Science, University of Oregon, December 1995. 635	12 13 14
[56]	4.4BSD Programmer's Supplementary Documents (PSD). O'Reilly and Associates, 1994. 543	15 16 17
[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.	18 19 20
[58]	Martin Schulz and Bronis R. de Supinski. $P^N$ MPI tools: A whole lot greater than the sum of their parts. In ACM/IEEE Supercomputing Conference (SC), pages 1–10. ACM, 2007. 722	21 22 23 24
[59]	K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In <i>Proceedings of Supercomputing '95</i> , December 1995. 635	25 26 27
[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, <i>Proceedings of the Fifth Distributed Memory Concurrent Computing Conference</i> , pages 767–776. IEEE Press, 1990. 2, 308	28 29 30 31
[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. 2	32 33 34 35
[62]	Anthony Skjellum, Nathan E. Doss, and Purushotham V. Bangalore. Writing Libraries in MPI. In Anthony Skjellum and Donna S. Reese, editors, <i>Proceedings of the Scalable Parallel Libraries Conference</i> , pages 166–173. IEEE Computer Society Press, October 1993. 307	36 37 38 39 40
[63]	Anthony Skjellum, Nathan E. Doss, and Kishore Viswanathan. Inter-communicator ex- tensions to MPI in the MPIX (MPI eXtension) Library. Technical Report MSU-940722, Mississippi State University — Dept. of Computer Science, August 1994. Archived at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.6283. 189	41 42 43 44
[64]	Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P. Leung, and Manfred Morari. The Design and Evolution of Zipcode. <i>Parallel Computing</i> , 20(4):565–596, April 1994. 308, 350	45 46 47 48

#### 

[65] The Internet Society. XDR: External Data Representation Standard, May 2006. http:  $\mathbf{2}$ //www.rfc-editor.org/pdfrfc/rfc4506.txt.pdf. 693 [66] Rajeev Thakur and Alok Choudhary. An extended two-phase method for accessing sections of out-of-core arrays. Scientific Programming, 5(4):301–317, Winter 1996. 635 [67] Jesper Larsson Träff. SMP-aware message passing programming. In Eighth Inter-national Workshop on High-level Parallel Programming Models and Supportive Envi-ronments (HIPS), 17th International Parallel and Distributed Processing Symposium (*IPDPS*), pages 56–65, 2003. 339 [68] The Unicode Standard, Version 2.0. Addison-Wesley, 1996. ISBN 0-201-48345-9. 693 [69] D. Walker. Standards for message passing in a distributed memory environment. Tech-nical Report TM-12147, Oak Ridge National Laboratory, August 1992. 2  24  31 

## General Index

This index lists mainly terms of the MPI specification. The underlined page numbers refer to the definitions or parts of the definition of the terms. Bold face numbers mark section titles.

absolute addresses, 20, 139, 819 access epoch, 587action in function names,  $\underline{10}$ active, 69, 383 active target communication, 587 addresses, 154 absolute, 20, 21, 139, 819 correct use, 154 relative displacement, 20, 139 alignment, 452, 551, 554, 1037 all-reduce, 236 nonblocking, 268 persistent, 293 all-to-all, 215 nonblocking, 261 persistent, 287 array arguments, 18 assertions, 601 ASYNCHRONOUS Fortran attribute, 821 attribute, 309, 361, 841 caching, 308 backward incompatibilies, 781 barrier synchronization, 191 nonblocking, 250 persistent, 275 blocking, <u>47</u>, 51, 651 I/O, 652 Blocking operation, 12 Blocking procedure,  $\underline{14}$ bounds of datatypes, 145 broadcast, 192 nonblocking, 251 persistent, 276 buffer allocation, 56 buffered, <u>48</u>, <u>60</u>, <u>61</u> nonblocking, 60buffered send, 52

language binding, <b>22</b>	12
caching, <b>307</b> , <u>308</u> , <b>361</b>	13
callback functions	14
language interoperability, 840	
	15
prototype definitions, <b>864</b> deprecated, <b>871</b>	16
cancel, 23, 70, <u>82</u> , <b>90</b> , 492, 776, 1036	17
canonical pack and unpack, 180	18
Cartesian	19
topology, <u>387</u> , <b>388</b>	20
Chameleon, 2	21
change-log, <b>1035</b>	22
Chimp, 2	23
choice, 19	24
class	25
in function names, $\underline{10}$	
clock synchronization, <b>450</b>	26
collective, 651	27
split, <b>701</b>	28
collective communication, 185	29
correctness, 299	30
file data access operations, 676	31
neighborhood, 412	32
nonblocking, 248	33
Collective operation, $\underline{13}$	
Collective procedure, $\underline{14}$	34
commit, <u>148</u>	35
COMMON blocks, 824	36
communication, 545	37
collective, 185	38
modes, <b>47</b> one-sided, <b>545</b>	39
overlap with computation, 59	40
partitioned point-to-point, <b>101</b>	41
point-to-point, <b>29</b>	42
$RMA, \underline{545}$	43
communicator, $34$ , <b>307</b> , $308$	
hidden, <b>188</b> , <b>302</b> , <b>595</b>	44
complete operation, 11	45
Completing procedure, $\underline{14}$	46
completion, $68$	47
	48

#### General Index

1 multiple, 73  $\mathbf{2}$ completion stage, 11connected, 5413 constants, 18, 845, 849 4 context, **307**, 308, 310 5control variables 6 tools interface, 730 7 conversion, 46 8 counts, 20 9 create 10 in function names, 10 11 data, 31 12data conversion, 46 13 datatype 14derived, 15 15equivalent, 1516named, 15 portable, 1517predefined, 15 18 unnamed,  $\underline{15}$ 19datatypes, 117, 838 20deadlock avoidance 21via nonblocking communication, 59 22 default file error, 712 23delete in function names, 10 24deprecated interfaces, 21, 773 25derived datatype, 15, 117, 815 26disconnected, 54227displacement, <u>635</u>, 649 28distributed graph 29 topology, 387, 392 30 Dynamic Process Model, 483 dynamically attached memory, 556  31 32 elementary datatype, <u>635</u>, 650 33 empty, 69 34 end of file, 63735 envelope, <u>30</u>, <u>33</u> 36 environmental inquiries, 449 37 equivalent datatypes, 15error handling, 24, 454 38 default file error handler, 638, 641, 712 39 error codes and classes, 466, 469 40 error handlers, 457, 469, 840 41 finalize, 25, 493, 542 42I/O, 711, 712 43 initial error handler, 25, 455, 485, 493, 526 44 one-sided communication, 603 process failure, 25, 542 45program error, 25 46 resource error, 25 47startup, 25, 485, 526 48

transmission failure,  $\underline{24}$ establishing communication, 528etype,  $\underline{635}$ , 650events tools interface, 749exclusive scan nonblocking, 273persistent, 298explicit offsets,  $\underline{652}$ , 654exposure epoch,  $\underline{587}$ extent of datatypes,  $\underline{118}$ ,  $\underline{143}$ , 145true extent, 147external32file data representation,  $\underline{691}$ extra-state, 845

fairness, 53, 79 file, 635 data access, 651 collective operations, 676 explicit offsets, 654 individual file pointers, 661 seek, 679 shared file pointers, 672 split collective, 680 end of file, 637filetype, 636 handle, 637interoperability, 689 manipulation, 637 offset, 20, 636 pointer, 637 size, 637 view, 635, 636, 648 file size, 707 finalize, 490 finished, 493 Fortran language binding, 22, 783 Fortran support, 783 Freeing procedure, 14 freeing stage, 12

gather, **194** nonblocking, **252** persistent, **277** gather-to-all, **211** nonblocking, **258** persistent, **284** general datatype, <u>117</u> generalized requests, **625**, <u>625</u> get in function names, <u>10</u> graph

topology, <u>387</u>, **390** group, 307, <u>308</u>, <u>310</u>, 352 group objects, 310handles, <u>16</u>, **833** hardware resource type, 335, 336 host rank, 449 immediate, <u>14</u>, <u>61</u> inactive, 68 inclusive scan, 243 nonblocking, 272 persistent, 297 incomplete,  $\underline{14}$ ,  $\underline{61}$ Incomplete procedure, 14 independent,  $\underline{542}$ individual file pointers, 652, 661 info object, 475 file info, 645 keys, 871 values, 872 initial error handler, 25Initialization procedure,  $\underline{13}$ initialization stage, 11initiation, <u>11</u>, **61** Initiation procedure, 14inter-communication, <u>309</u>, <u>351</u> inter-communicator, 309, 351 collective operations, 189, 190 interlanguage communication, 846 internal file data representation, 690 interoperability, 689 intra-communication, 309, 351 intra-communicator, <u>308</u>, <u>351</u> collective operations, 188 intra-communicator objects, <u>311</u> I/O, 635 IO rank, **450** is in function names, 10language binding, 21, 783 interoperability, 832summary, 849 large count, 21, 831 displacement, 831 large count, 21lb_marker, 133, 137, <u>142</u>, 143, 148 erased, 146 local, 48local group, 324 Local procedure, 13 loosely synchronous model, 382

lower bound, $\underline{143}$	1
lower-bound markers, <b>142</b>	2
	3
macros, 24	4
main thread, $514$	5
matched receives, 87	6
matching	7
type, <b>43</b> , <u>151</u> , <b>706</b>	8
matching probe, 85	
nemory	9
alignment, 452, 551, 554, 1037 allocation, <b>451</b> , 550, 552	10
system, $\frac{16}{16}$	11
memory model, 546, <u>586</u>	12
separate, 546, 554	13
unified, 546, 554	14
message, 29	15
data, <b>31</b>	16
envelope, 33	10
modes, 47	
module variables, 824	18
mpi module	19
Fortran support, <b>787</b>	20
MPI operation, $\underline{11}$	21
MPI process initialization, $483$	22
Dynamic Process Model, 483	23
Sessions Model, <b>311</b> , <b>314</b> , <b>449</b> , <b>483</b> , <b>5</b> 1	<b>14</b> . 24
638	25
World Model, <b>311</b> , <b>314</b> , <b>449</b> , <b>483</b> , <b>514</b>	
<b>638</b> , <b>712</b>	. 20
mpi_f08 module	27
Fortran support, <b>784</b>	28
MPI_SIZEOF and storage_size(), 23, 778,	29
807-809	30
mpiexec, <u>486</u> , <u>510</u>	31
mpif.h include file	32
Fortran support, <b>789</b>	33
mpirun, 510	34
multiple completions, 73	
	35
named datatype, $\underline{15}$	36
names, <b>529</b>	37
name publishing, <b>534</b>	38
naming objects, <b>377</b>	39
native	40
file data representation, $\underline{690}$	41
neighborhood collective communication, 412	42
nonblocking, <b>425</b>	
periodic and dims==1 or 2, $1035$	43
non-local, 47, 48	44
Non-local procedure, $\underline{13}$	45
nonblocking, <b>59</b> , 563, 651	46
communication, $\underline{59}$	47
completion, 68	48

#### 1064

1 Fortran problems, 818  $\mathbf{2}$ I/O, 652 initiation, 61 3 persistent 4 partitioned completion, 108 5request objects, 61 6 Nonblocking operation, 12 7 Nonblocking procedure, 148 Noncollective operation, 13 9 null handle, 68 10 null processes, 98 11 offset, 20, 636 12one-sided communication, 545 13 Fortran problems, 819 14opaque objects, 16, 838 15operation 16blocking, 12 collective, 1317complete, 11 18 nonblocking,  $\underline{12}$ 19 noncollective, 13 20operation-related, 13 21partitioned receive, 12 22 partitioned send, 12 23persistent, 12origin, 546 24 25pack, 172 26canonical, 180 27packing unit, 175 28parallel procedure, 382 29partitioned completion, **108** partitioned point-to-point communication, 101 30 passive target communication, 587  31 performance variables 32 tools interface, 736 33 persistent communication requests, 92 34 collective persistent, 274, 432 35 Fortran problems, 819 36 Persistent operation, 1237 PICL, 2 PMPI_, 717 38 point-to-point communication, 29 39 portable datatype, 1540 ports, **529** 41 POSIX 42environment, 474 43 FORTRAN, 17 44 I/O, 638, 639, 652, 690, 702 model, **635** 45 predefined datatype, 15 46 predefined reduction operations, 224 47private window copy, 585 48

probe, 82 matching, 85 procedure blocking, 14 collective, 14 completing, 14freeing, 14 incomplete,  $\underline{14}$ initialization, 13 initiation, 14 local, 13non-local, 13nonblocking, 14 operation-related,  $\underline{13}$ starting,  $\underline{13}$ process creation, 483 process failures, 25 process group, 34processes, 24 processor name, 450 profiling interface, 717 program error, 25 progress, 249 prototype definitions, 864 deprecated, 871 public window copy, 585 PVM, 2 rank, 310 ready, 48, 60, 61 nonblocking, 60 ready send, 52receive, 29, 30, 34 buffer, 30complete, 59 context, 352 start call, 59 reduce, **222** nonblocking, 267 persistent, 292 reduce-scatter, 240 nonblocking, 269, 271 persistent, 294, 296 reduction operations, 221, 840 predefined, 224 process-local, 238 scan, **243** user-defined, 231 related, 175 relative displacement, 20, 139 remote group, 324 Remote Memory Access see RMA, 545 removed interfaces, 21, 779

request complete I/O, 652 request objects, 61 resource error, 25RMA, 545 communication calls, 563 request-based, 578 memory model, 585 synchronization calls, 587 scan, **243** inclusive, **243** scatter, 204 nonblocking, 255 persistent, 281 seek, 679 semantics, 1036 collective communications, 186 file collective, 705 collective access, 676 conflicting access, 702 consistency, 701 explicit offsets, 654 nonblocking collective, 705 shared file pointer, 672 split collective, 701 inter-communicator, 324MPI_COMM_IDUP, 326 nonblocking communications, 72 nonblocking partitioned communications, **109** partitioned point-to-point communication, 102 point-to-point communication, 52 process failure, 542 semantics and correctness one-sided communication, 604 send, <u>29</u>, **30** buffer, 29complete, 59 context, 352start, 59send-receive, 40nonblocking, 67, 68 separate memory model, 546, 554, 586 sequential storage,  $\underline{155}$ serialization, 59 Sessions Model, 311, 314, 449, 483, 514, 638 set in function names,  $\underline{10}$ shared file pointers, 652, 672 shared memory allocation, 552 signals, 27

		-
	singleton init, <b>540</b>	1
	size changing	2
	I/O, 707	3
	source, $352$	4
	split collective, 651, 680	5
	stage	6
	completion, $\underline{11}$	7
	freeing, $\underline{12}$	
	initialization, $\underline{11}$	8
	starting, $\underline{11}$	9
	standard, $\underline{47}$ , 60	10
	nonblocking, $\underline{60}$	11
	standard send, 52	12
	started, <b>493</b>	13
	Starting procedure, $\underline{13}$	14
	starting processes, <b>516</b> , <b>517</b>	
	starting stage, <u>11</u>	15
	startup, 484	16
	portable, <b>510</b>	17
	state, 18	18
	status, <b>36</b> , <b>835</b>	19
	associating information, 632	20
	ignore, <b>39</b>	21
	test, <b>81</b>	22
	strong synchronization, $588$	
	synchronization, $545$ , $563$	23
	synchronization calls	24
Ĺ	RMA, <b>587</b>	25
1	synchronous, $48, 60, 61, 68$	26
	nonblocking, $\underline{60}$	27
	synchronous send, 52	28
	system memory, $\underline{16}$	29
		30
	tag values, 449	
	target, <u>546</u>	31
	thread compliant, $\underline{488}$ , $\underline{512}$	32
	threads, <b>512</b>	33
	thread-safe, 2, 448, 465, 469, 752	34
	timers and synchronization, 473	35
	tool information interface, <b>723</b>	36
	tool support, 717	37
	topologies, 385	
	topology	38
	Cartesian, $387$ , $388$	39
	distributed graph, <u>387</u> , <b>392</b>	40
	$\operatorname{graph}, \frac{387}{282}, 390$	41
	virtual, <b>386</b>	42
	transmission failures, $\underline{24}$	43
	true extent of datatypes, 147	44
	TS 29113, 19, 22, 783–788, 790–793, 796–800,	45
	809–811, 816, 821, 824, 826, 1045	
	type map, $\underline{118}$ type metabing <b>42</b> , 151	46
	type matching, $43$ , $151$ type signature, $118$	47
	type signature, <u>118</u>	48

1	types, <b>863</b>
2	$m_{\rm c} = 122 - 127 - 120 - 149 - 149 - 140$
3	ub_marker, 133, 137, 138, <u>142</u> , 143, 148 erased, <u>146</u>
4	unified memory model, $546$ , $554$ , $586$
5	united memory model, 540, 554, <u>580</u> universe size, <b>539</b>
6	unnamed datatype, <u>15</u>
7	unpack, 172
8	canonical, 180
9	upper bound, <u>143</u>
10	upper bound markers, 142
11	user functions at process termination, 494
	user-defined data representations, 696
12	user-defined reduction operations, 231
13	
14	verbosity levels
15	tools interface, 724
16	version inquiries, 447
17	view, 635, <u>636</u> , <b>648</b>
18	virtual topology, <u>308</u> , <u>309</u> , <b>386</b>
19	weak synchronization, 589
20	window
21	allocation, 550
22	creation, 547
23	dynamically attached memory, 556
24	shared memory allocation, 552
25	World Model, <b>311</b> , <b>314</b> , <b>449</b> , <b>483</b> , <b>514</b> , <b>638</b> ,
	712
26	VDD (7
27	XDR, 47
28	Zipcode, 2
29	
30	
31	
32	
33	
34	
35	
36	
37	
38	
39	
40	
41	
42	
42	
44	
45	
46	
47	
48	

## **Examples Index**

This index lists code examples throughout the text. Some examples are referred to by content; others are listed by the major MPI function that they are demonstrating. MPI functions listed in all capital letter are Fortran examples; MPI functions listed in mixed case are C examples.

ASYNCHRONOUS, 620, 827, 829 Attributes between languages, 841	register optimization, 818, 820
	Independence of nonblocking operations, 306
Blocking/Nonblocking collectives do not	Inter-communicator, 328, 334
match, $304$	Interlanguage communication, 846
,	Intertwined matching pairs, 53
C/Fortran handle conversion, 835	or a state of the
Cartesian virtual topologies, 440	Message exchange (ping-pong), 54
Client-server code, 80	Mixing blocking and nonblocking collective
with probe, 84	operations, $302$
with probe (wrong), 84	Mixing collective and point-to-point requests, 304
Datatype	
3D array, 164	MPI_ACCUMULATE, 572
absolute addresses, 169	MPI_Accumulate, 618, 619, 622
array of structures, 166	MPI_Aint, 166
elaborate example, 177, 178	MPI_Aint_add, 622
matching type, 151	MPI_Allgather, 214
matrix transpose, 165	MPI_ALLOC_MEM, 453, 454
	MPI_Alloc_mem, 454, 622
union, 170	MPI_ALLREDUCE, 237
Datatypes	MPI_Alltoall, 304
matching, 44	MPI_ASYNC_PROTECTS_NONBLOCKING,
not matching, 44	620
untyped, 44	MPI_BARRIER, 506
Deadlock	MPI_Barrier, 492, 608–611, 617–619
if not buffered, 55	MPI_Bcast, 193, 299–303
with MPI_Bcast, 299, 300	MPI_BSEND, $53$
wrong message exchange, 54	MPI_Buffer_attach, 57, 492
False matching of collective exercising 202	MPI_Buffer_detach, 57
False matching of collective operations, 303	MPI_BYTE, 44
Fortran 90	MPI_Cancel, 492
copying and sequence problem, 811, 813,	MPI_CART_COORDS, 408
814	$MPI_CART_GET, 440$
derived types, 815	MPI_CART_RANK, 408
heterogeneous communication (unsafe),	MPI_CART_SHIFT, 408, 419, 440
808, 809	MPI_CART_SUB, 410
invalid KIND, 804	MPI_CARTDIM_GET, 419
MPI_TYPE_MATCH_SIZE	MPI_CHARACTER, 45
implementation, 807	MPI_Comm_accept, 537, 538
overlapping communication and	MPI_Comm_connect, 537, 538
computation, 827, 828, 830	MPI_Comm_create, 328, 345, 346, 348

#### 1068

1	MPI_COMM_CREATE_FROM_GROUP, 502,
2	506
3	MPI_Comm_create_keyval, 375
4	MPI_Comm_dup, 348
5	MPI_Comm_get_attr, 375
6	MPI_Comm_get_parent, 526
7	MPI_Comm_group, 328, 348, 375
8	MPI_Comm_rank, 29
	MPI_Comm_remote_size, 334
9	MPI_Comm_set_attr, 375
10	MPI_COMM_SPAWN, 520
11	MPI_Comm_spawn, 520, 526
12	MPI_COMM_SPAWN_MULTIPLE, 525
13	MPI_Comm_spawn_multiple, 525 MPI_Comm_split, 334, 358, 359
14	MPI_Comm_split_type, 337, 339
15	MPI_Compare_and_swap, 619, 622
16	MPI_DIMS_CREATE, 390, 440
17	MPI_DIST_GRAPH_CREATE, 397
18	MPI_Dist_graph_create, 398
	MPI_DIST_GRAPH_CREATE_ADJACENT,
19	397
20	MPI_F_sync_reg, 620
21	MPI_FILE_CLOSE, 663, 667
22	MPI_FILE_GET_AMODE, 644
23	MPI_FILE_IREAD, 667
24	MPI_FILE_OPEN, 663, 667
25	MPI_FILE_READ, 663
26	MPI_FILE_SET_ATOMICITY, 707
27	MPI_FILE_SET_VIEW, 663, 667
28	MPI_FILE_SYNC, 708
29	MPI_Finalize, 491–493
30	MPI_FREE_MEM, 453, 454
31	MPI_Free_mem, 622
32	MPI_Gather, 178, 198, 199, 203 MPI_Gathery, 178, 199–203
	MPI_GET, 568, 570
33	MPI_Get, 608–611, 616, 617
34	MPI_Get_accumulate, 618, 619, 622
35	MPI_GET_ADDRESS, 140, 815, 816, 838
36	MPI_Get_address, 166, 169, 170, 177
37	MPI_GET_COUNT, 153
38	MPI_GET_ELEMENTS, 153
39	MPI_GRAPH_CREATE, 391, 404
40	MPI_GRAPH_NEIGHBORS, 404
41	MPI_GRAPH_NEIGHBORS_COUNT, 404
42	MPI_Grequest_complete, 630
43	MPI_Grequest_start, 630
	MPI_Group_excl, 345
44	MPI_Group_free, 328, 345, 346
45	MDI ODOLID EDOM GEGGION DOET FOO
10	MPI_GROUP_FROM_SESSION_PSET, 502,
46	505, 506
46 47	

MPI_Ialltoall, 304 MPI_Ibarrier, 302–304 MPI_Ibcast, 252, 305, 306 MPI_INFO_CREATE, 502 MPI_INFO_ENV, 486 MPI_INFO_SET, 502 MPI_Init, 29 MPI_Intercomm_create, 358, 359 MPI_Iprobe, 492 MPI_IRECV, 71-73, 80 MPI_Irecv, 304 MPI_ISEND, 71, 72, 80 MPI_NEIGHBOR_ALLGATHER, 415 MPI_NEIGHBOR_ALLTOALL, 419 MPI_Op_create, 235, 236, 246 MPI_Open_port, 537, 538 MPI_Pack, 177, 178 MPI_Pack_size, 178 MPI_Parrived, 113 MPI_Pready, 102, 110, 111, 113 MPI_Precv_init, 102, 110, 111, 113 MPI_PROBE, 84 MPI_Psend_init, 102, 110, 111, 113 MPI_Publish_name, 537 MPI_Put, 593, 599, 609-611, 616, 617 MPI_RECV, 44, 45, 53-55, 73, 84, 151 MPI_Recv, 29, 303 MPI_REDUCE, 226, 229 MPI_Reduce, 229, 230, 235, 236 MPI_REQUEST_FREE, 72 MPI_Request_free, 492 MPI_Rget, 621 MPI_Rput, 621 MPI_Scan, 246 MPI_Scatter, 208 MPI_Scatterv, 208, 209 MPI_SEND, 44, 45, 54, 55, 73, 84, 151 MPI_Send, 29, 166, 169, 170, 177, 303, 304 MPI_SENDRECV, 164, 165, 419 MPI_SENDRECV_REPLACE, 408 MPI_SESSION_FINALIZE, 502, 505, 506 MPI_SESSION_GET_NTH_PSET, 505, 506 MPI_SESSION_GET_NUM_PSETS, 505, 506 MPI_SESSION_INIT, 502, 505, 506 MPI_SSEND, 53, 73 MPI_Test_cancelled, 492 MPI_TYPE_COMMIT, 149, 164, 165, 568, 815, 816 MPI_Type_commit, 166, 169, 170, 177, 198-203, 209, 246 MPI_TYPE_CONTIGUOUS, 120, 143, 151, 153MPI_Type_contiguous, 198 MPI_TYPE_CREATE_DARRAY, 138

MPI_TYPE_CREATE_HVECTOR, 164, MPI_Type_create_hvector, 166, 169 MPI_TYPE_CREATE_INDEXED_BLO 568MPI_TYPE_CREATE_RESIZED, 815, 8 MPI_TYPE_CREATE_STRUCT, 130, 1 165, 815, 816 MPI_Type_create_struct, 166, 169, 170, 201, 203, 246 MPI_TYPE_CREATE_SUBARRAY, 716 MPI_TYPE_EXTENT, 568 MPI_TYPE_FREE, 568 MPI_Type_get_contents, 171 MPI_Type_get_envelope, 171 MPI_TYPE_GET_EXTENT, 164, 165, 5 572 $MPI_Type_get_extent$ , 166 MPI_TYPE_INDEXED, 125, 164 MPI_Type_indexed, 166, 169 MPI_TYPE_VECTOR, 121, 164, 165 MPI_Type_vector, 199, 200, 202, 209 MPI_Unpack, 177, 178 MPI_Unpublish_name, 537 MPI_User_function, 236 MPI_WAIT, 71-73, 80, 667 MPI_Wait, 302–304 MPI_Waitall, 304, 305, 621 MPI_WAITANY, 80 MPI_Waitany, 621 MPI_WAITSOME, 80 MPI_Win_attach, 622 MPI_Win_complete, 593, 611, 617 MPI_WIN_CREATE, 568, 570, 572 MPI_Win_create_dynamic, 622 MPI_Win_detach, 622 MPI_WIN_FENCE, 568, 570, 572 MPI_Win_fence, 616 MPI_Win_flush, 610, 618, 619, 622 MPI_Win_flush_all, 619 MPI_Win_flush_local, 609 MPI_WIN_FREE, 570, 572 MPI_Win_lock, 599, 608-611 MPI_Win_lock_all, 620-622 MPI_Win_post, 611, 617 MPI_Win_start, 593, 611, 617 MPI_Win_sync, 609, 610, 618, 619 shared memory windows, 620 MPI_Win_unlock, 599, 608-611 MPI_Win_unlock_all, 621, 622 MPI_Win_wait, 611, 617 mpiexec, 486, 511, 512

Neighborhood collective communication,

165	Non-deterministic program with MPI_Bcast,	1
att	301	2
CK,	Non-overtaking messages, 53	3
10	Nonblocking operations, 71, 72	4
816 42	message ordering, 72	5
43,	progress, 73	6
177,	Overlapping Communicators, 305	7
	Partitioned Communication	8
3	Equal send/recv partitioning, 110	9
	Partial completion notification, 113	10
	Send with tasks, 111	11
	Simple example, 102	12
570,	Pipelining nonblocking collective operations, $\frac{305}{2}$	13 14
	Point-to-point	15
	Hello world, 29	
	Profiling interface	16
	implementation using the C macro	17
	preprocessor, 721	18
	implementation using weak symbols, 720	19
	measurement wrapper, 720	20
	Progression of nonblocking collective	21
	operations, 303	22
		23
	Shared memory windows	24
	MPI_Win_sync, 620	25
	Threads and MPI, 513	26
	Tool information interface	27
	basic usage of performance variables, 747	28
	listing names of all control variables, 733	29
	reading the value of a control variable, 736	30
	Topologies, 440	31
	Typemap, 119–121, 125, 130, 138	32
	Virtual topologies, 440	33
	1 0 /	34
		35
		36
		37
		38
		39
		40
		40
		41
		43
		44
		45
4.40		46
440		47
		48

# MPI Constant and Predefined Handle Index

This index lists predefined MPI constants and handles, including info keys and values. Underlined page numbers give the location of the primary definition or use of the indexed term.

14"access_style", 647 15"accumulate_ops", 548 16"accumulate_ordering", 548, 612, 613 "alloc_shared_noncontig", 553, 554 17"appnum", 541 18 "arch", 486, 525 19 "argv", 486 2021"cb_block_size", 647 22"cb_buffer_size", 647 "cb_nodes", 647 23"chunked", 647 24"chunked_item", 647 25"chunked_size", 647 26"collective_buffering", 647 27"command", 486 28"external32", 180, 690, 691, 693-696, 701, 29 803-806, 809, 1042, 1049 30  31 "false", 342, 647 32 "file", 486, 526 33 "file_perm", 647 34 "filename", 647 35 "host", 486, 525 36 "hwloc://L3Cache", 498 37 38 "internal", 690, 691, 701 39 "io_node_list", 648 "ip_address", 537 40 "ip_port", 536 41 42"maxprocs", 486, 487 43 "mpi://", 499 44"mpi://SELF", 498, 499, 638 "mpi://WORLD", 498, 499, 505 45MPI::_LONG_LONG, 1042 46 MPI::BOOL, 1041 47MPI::COMPLEX, 1041 48

5 6

12

13

MPI::DOUBLE_COMPLEX, 1041 MPI::F_COMPLEX16, 1042 MPI::F_COMPLEX32, 1042 MPI::F_COMPLEX4, 1042 MPI::F_COMPLEX8, 1042 MPI::INTEGER16, 1042 MPI::LONG_DOUBLE_COMPLEX, 1041 MPI::LONG_LONG, 1042 MPI::REAL16, 1042 MPI_2DOUBLE_PRECISION, 228, 855 MPI_2INT, 228, 855 MPI_2INTEGER, 228, 855 MPI_2REAL, 228, 855 MPI_ADDRESS_KIND, 19, 20, 33, 361, 810, 841.852 MPL_AINT, 33, 119, 225, 557, 694, 831, 853, 854, 1047, 1049 MPI_ANY_SOURCE, 35, 36, 40, 42, 52, 65-67, 69, 82-87, 96, 342, 383, 450, 851 MPI_ANY_TAG, 18, 35, 36, 38, 40, 42, 65-67, 69, 82, 83, 85-87, 89, 90, 96, 99, 342, 851.1044 MPI_APPNUM, 541, 858 MPI_ARGV_NULL, 19, 519, 520, 524, 810, 860 MPI_ARGVS_NULL, 19, 524, 810, 860 "mpi_assert_allow_overtaking", 342, 1037 "mpi_assert_exact_length", 342, 1037 "mpi_assert_no_any_source", 342, 1037 "mpi_assert_no_any_tag", 342, 1037 MPI_ASYNC_PROTECTS_NONBLOCKING, 19, 620, 784, 785, 787, 789, 792, 800, 801, 821, 852, 1046 MPI_BAND, 224, 226, 856 MPI_BOR, 224, 226, 856 MPI_BOTTOM, 10, 19, 20, 39, 109, 139, 154, 155, 188, 394, 396, 522, 557, 561, 786, 788, 796, 810, 815, 817, 819, 820, 822-824, 826, 838-840, 845, 851, 1053 MPI_BSEND_OVERHEAD, 59, 851

MPI_BXOR, 224, 226, 856
MPI_BYTE, <u>31</u> , 32, 43–46, 180, 225, 636, 690,
691, 694, 706, 846, 853, 854, 1050
MPI_C_BOOL, <u>32</u> , 225, 694, 853, 1042,
1047-1049
MPI_C_COMPLEX, <u>32</u> , 225, 694, 853, 1041,
1047-1049
MPI_C_DOUBLE_COMPLEX, <u>32</u> , 225, 694,
853, 1047-1049 MDL C FLOAT COMPLEX 22, 225, 604, 852
MPI_C_FLOAT_COMPLEX, <u>32</u> , 225, 694, 853, 1047–1049
MPI_C_LONG_DOUBLE_COMPLEX, <u>32</u> , 225,
694, 853, 1047–1049
MPI_CART, <u>399</u> , 857
MPI_CHAR, <u>32</u> , 46, 130, 227, 694, 728, 853,
1047
MPI_CHARACTER, <u>31</u> , 45, 46, 227, 694, 854
MPI_COMBINER_CONTIGUOUS, 157, <u>161</u> , 859
MPI_COMBINER_DARRAY, 157, <u>163</u> , 859
MPI_COMBINER_DUP, 157, <u>161</u> , 859
MPI_COMBINER_F90_COMPLEX, 157, <u>163</u> ,
859
MPI_COMBINER_F90_INTEGER, 157, <u>163</u> , 859
MPI_COMBINER_F90_REAL, 157, <u>163</u> , 859
MPI_COMBINER_HINDEXED, 23, 157, <u>162</u> ,
859
MPI_COMBINER_HINDEXED_BLOCK, 157,
<u>162</u> , 859, 1043
MPI_COMBINER_HINDEXED_INTEGER, 23,
780, 1042
MPI_COMBINER_HVECTOR, 23, 157, <u>162</u> , 859
MPI_COMBINER_HVECTOR_INTEGER, 23,
780, 1042
MPI_COMBINER_INDEXED, 157, <u>162</u> , 859
MPI_COMBINER_INDEXED_BLOCK, 157, 162,
859
MPI_COMBINER_NAMED, 157, <u>161</u> , 859
MPI_COMBINER_RESIZED, 157, <u>163</u> , 859
MPI_COMBINER_STRUCT, 23, 157, 162, 859
MPI_COMBINER_STRUCT_INTEGER, 23, 780,
1042
MPI_COMBINER_SUBARRAY, 157, <u>163</u> , 859
MPI_COMBINER_VECTOR, 157, <u>161</u> , 859
MPI_COMM_DUP_FN, 23, <u>364</u> , 857, 1046
MPI_COMM_NULL, <u>311</u> , 327, 328, 330–332,
335-337, 340, 341, 357, 379, 389, 391,
522, 543, 544, 834, 856, 1050
MPI_COMM_NULL_COPY_FN, 23, <u>364</u> , 786,
840, 857, 1046
MPI_COMM_NULL_DELETE_FN, 23, <u>365</u> , 857
MPI_COMM_PARENT, <u>379</u>
MPI_COMM_SELF, 25, <u>311</u> , 330, 343, 361, 379,
455, 485, 490, <del>493</del> , 495, 543, 638, 781,
855, 1036, 1048

MPI_COMM_TYPE_HW_GUIDED, <u>335</u> , 336, 855, 1036	$\frac{1}{2}$
MPI_COMM_TYPE_HW_UNGUIDED, <u>336</u> ,	3
337, 338, 855, 1036	4
MPI_COMM_TYPE_SHARED, <u>335</u> , <u>336</u> , <u>855</u> ,	5
1044	6
MPI_COMM_WORLD, 18, 26, 34, 309,	7
311-314, 322, 323, 337-339, 343, 346,	8
355, 379, 389, 449, 450, 458, 470,	
483-486, 488-491, 493, 495, 509, 510,	9
516-519, 522, 524, 539-543, 689, 711,	10
712, 734, 743, 781, 833, 845, 855,	11
1036, 1052	12
MPI_COMPLEX, <u>31</u> , 225, 693, 694, 802, 854	13
MPI_COMPLEX16, <u>33</u> , 225, 695, 854	14
MPI_COMPLEX32, <u>33</u> , 225, 695, 854	
MPI_COMPLEX4, <u>33</u> , 225, 695, 854	15
MPI_COMPLEX8, <u>33</u> , 225, 695, 854	16
MPI_CONGRUENT, <u>323</u> , 353, 855	17
MPI_CONVERSION_FN_NULL, 700, 857	18
MPI_COUNT, <u>33</u> , 119, 225, 233, 694, 728, 831,	19
853,854,1043	20
MPI_COUNT_KIND, 19, <u>20</u> , 33, 852	21
MPI_CXX_BOOL, <u>33</u> , 225, 694, 695, 854, 1041	
MPI_CXX_DOUBLE_COMPLEX, <u>33</u> , 225, 694,	22
695,854,1041	23
MPI_CXX_FLOAT_COMPLEX, <u>33</u> , 225, 694,	24
695, 854, 1041	25
MPI_CXX_LONG_DOUBLE_COMPLEX, <u>33</u> ,	26
225,694,695,854,1041	27
MPI_DATATYPE_NULL, <u>150</u> , 856	28
MPI_DISPLACEMENT_CURRENT, 649, 860,	29
1053	
MPI_DIST_GRAPH, <u>399</u> , 857, 1048	30
MPI_DISTRIBUTE_BLOCK, <u>135</u> , 136, 860	31
MPI_DISTRIBUTE_CYCLIC, <u>135</u> , 136, 860	32
MPI_DISTRIBUTE_DFLT_DARG, 135, 136, 860	33
MPI_DISTRIBUTE_NONE, <u>135</u> , 136, 860	34
MPI_DOUBLE, <u>32</u> , 225, 694, 728, 737–739, 801,	35
853	36
MPI_DOUBLE_COMPLEX, <u>32</u> , 225, 693, 694,	
802,854	37
MPI_DOUBLE_INT, 228, 229, 855	38
MPI_DOUBLE_PRECISION, <u>31</u> , 225, 694, 802,	39
854	40
MPI_DUP_FN, 23, 364, <u>774</u> , 858	41
MPI_ERR_ACCESS, 467, 641, 713, 850	42
MPI_ERR_AMODE, 467, 639, 713, 850	43
MPI_ERR_ARG, 467, 849	
MPI_ERR_ASSERT, 467, 604, 850	44
MPI_ERR_BAD_FILE, 467, 713, 850	45
MPI_ERR_BASE, 453, 467, 604, 850	46
MPI_ERR_BUFFER, 467, 849	47
MPI_ERR_COMM, 467, 849	48

1	MPI_ERR_CONVERSION, 467, 700, 713, 850
2	MPI_ERR_COUNT, 467, 849
3	MPI_ERR_DIMS, 467, 849
	MPI_ERR_DISP, 467, 604, 850
4	MPI_ERR_DUP_DATAREP, 467, 697, 713, 850
5	MPI_ERR_FILE, 467, 713, 850
6	MPI_ERR_FILE_EXISTS, 467, 713, 850
7	MPI_ERR_FILE_IN_USE, 467, 641, 713, 850
8	MPI_ERR_GROUP, 467, 849
9	
	MPI_ERR_IN_STATUS, 37, 39, 69, 76, 78, 457,
10	466, 467, 629, 654, 850
11	MPI_ERR_INFO, 467, 850
12	MPI_ERR_INFO_KEY, 467, 477, 850
13	MPI_ERR_INFO_NOKEY, 467, 477, 850
14	MPI_ERR_INFO_VALUE, 467, 477, 850
15	MPI_ERR_INTERN, 456, 467, 849
	MPI_ERR_IO, 467, 713, 850
16	MPI_ERR_KEYVAL, 375, 467, 850
17	MPI_ERR_LASTCODE, 466, 468, <u>470</u> , 471, 769,
18	851
19	MPI_ERR_LOCKTYPE, 467, 604, 850
20	MPI_ERR_NAME, 467, 536, 850
21	MPI_ERR_NO_MEM, 452, 467, 850
	MPI_ERR_NO_SPACE, 467, 713, 850
22	MPI_ERR_NO_SUCH_FILE, 467, 641, 713, 850
23	MPI_ERR_NOT_SAME, 467, 713, 850
24	MPI_ERR_OP, 468, 603, 849
25	MPI_ERR_OTHER, 466, 468, 849
26	MPI_ERR_PENDING, 76, 468, 849
27	MPI_ERR_PORT, 468, 533, 850
28	MPI_ERR_PROC_ABORTED, 468, 510, 850,
	1037
29	MPI_ERR_QUOTA, 468, 713, 850
30	MPI_ERR_RANK, 468, 603, 849
31	MPI_ERR_READ_ONLY, 468, 713, 850
32	MPI_ERR_REQUEST, 468, 849
33	MPI_ERR_RMA_ATTACH, 468, 604, 850
34	MPI_ERR_RMA_CONFLICT, 468, 604, 850
	MPI_ERR_RMA_FLAVOR, 468, 555, 604, 850
35	MPI_ERR_RMA_RANGE, 468, 604, 850
36	MPI_ERR_RMA_SHARED, 468, 604, 850
37	MPI_ERR_RMA_SYNC, 468, 604, 850
38	MPI_ERR_ROOT, 468, 849
39	MPI_ERR_SERVICE, 468, 536, 850
40	MPI_ERR_SESSION, 468
41	MPI_ERR_SIZE, 468, 604, 850
	MPI_ERR_SPAWN, 468, 520, 521, 850
42	MPI_ERR_TAG, 468, 849
43	MPI_ERR_TOPOLOGY, 468, 849
44	MPI_ERR_TRUNCATE, 468, 849
45	MPI_ERR_TYPE, 157, 159, 468, 712, 849
46	MPI_ERR_UNKNOWN, 466, 468, 849
47	MPI_ERR_UNSUPPORTED_DATAREP, 468,
	713, 850
48	(13, 000

MPI_ERR_UNSUPPORTED_OPERATION, 468, 713,850 MPI_ERR_VALUE_TOO_LARGE, 468, 698, 1036 MPI_ERR_WIN, 468, 604, 850 MPI_ERRCODES_IGNORE, 19, 521, 522, 810, 860 MPI_ERRHANDLER_NULL, 465, 856 MPI_ERROR, 37, 69, 248, 578, 836, 852, 1039, 1045 MPI_ERRORS_ABORT, 455, 485, 526, 1036 "mpi_errors_abort", 526, 1038 MPI_ERRORS_ARE_FATAL, 455, 456, 472, 526, 603, 711, 852, 1036 "mpi_errors_are_fatal", 526, 1038 MPI_ERRORS_RETURN, 455, 456, 473, 509, 526, 712, 845, 852 "mpi_errors_return", 526, 1038 MPI_F08_STATUS_IGNORE, 837, 861, 1045 MPI_F08_STATUSES_IGNORE, 837, 861, 1045 MPI_F_ERROR, 836, 852, 1035 MPI_F_SOURCE, 836, 852, 1035 MPI_F_STATUS_IGNORE, 836, 861 MPI_F_STATUS_SIZE, 19, 836, 852, 1035 MPI_F_STATUSES_IGNORE, 836, 861 MPI_F_TAG, 836, 852, 1035 MPI_FILE_NULL, 640, 712, 856 MPI_FLOAT, 32, 130, 223, 225, 692, 694, 853 MPI_FLOAT_INT, 15, 228, 229, 855 MPI_GRAPH, 399, 857 MPI_GROUP_EMPTY, 310, 316, 317, 327, 328, 330, 340, 357, 856 MPI_GROUP_NULL, 310, 320, 321, 856 MPI_HOST, <u>449</u>, 855 "mpi_hw_resource_type", 336, 337, 1036 MPI_IDENT, <u>314</u>, 323, 855 MPI_IN_PLACE, 19, 188, 218, 789, 810, 851 MPI_INFO_ENV, 482, 486, 487, 495, 855, 1044 MPI_INFO_NULL, 397, 452, 481, 521, 531, 639, 641, 650, 856 "mpi_initial_errhandler", 485, 486, 526, 1038 MPI_INT, 15, <u>32</u>, 118, 224, 692–694, 727, 728, 731, 737, 741, 801, 845, 847, 853 MPI_INT16_T, 32, 225, 694, 853, 1047, 1049 MPI_INT32_T, 32, 225, 694, 728, 853, 1047, 1049 MPI_INT64_T, <u>32</u>, 225, 694, 728, 853, 1047, 1049 MPI_INT8_T, <u>32</u>, 225, 694, 853, 1047, 1049 MPI_INTEGER, 31, 43, 225, 694, 801, 802, 847, 854 MPI_INTEGER1, 33, 225, 695, 854 MPI_INTEGER16, 225, 695, 854 MPI_INTEGER2, 33, 225, 693, 695, 854

MPI_INTEGER4, 33, 225, 695, 854 MPI_INTEGER8, 33, 225, 695, 806, 854 MPI_INTEGER_KIND, 19, 852 MPI_IO, 449, 450, 855 MPI_KEYVAL_INVALID, 365, 366, 367, 851 MPI_LAND, 224, 225, 856 MPI_LASTUSEDCODE, 470, 858 MPI_LB, 23, 780, 1042 MPI_LOCK_EXCLUSIVE, 596, 851 MPI_LOCK_SHARED, 596, 597, 851 MPI_LOGICAL, 31, 225, 694, 854 MPI_LONG, 32, 224, 694, 853 MPI_LONG_DOUBLE, 32, 225, 694, 853 MPI_LONG_DOUBLE_INT, 228, 855 MPI_LONG_INT, 228, 229, 855 MPI_LONG_LONG, 32, 225, 853, 1049 MPI_LONG_LONG_INT, 32, 225, 694, 853, 1049 MPI_LOR, 224, 225, 856 MPI_LXOR, 224, 225, 856 MPI_MAX, 223-225, 246, 856 MPI_MAX_DATAREP_STRING, 18, 651, 697, 852 MPI_MAX_ERROR_STRING, 18, 465, 471, 852 MPI_MAX_FROM_GROUP_STRINGTAG, 357 MPI_MAX_FROM_GROUP_TAG, 340, 852 MPI_MAX_INFO_KEY, 18, 467, 475, 478, 479, 852 MPI_MAX_INFO_VAL, 18, 467, 475, 852 MPI_MAX_LIBRARY_VERSION_STRING, 18, <u>448</u>, 852, 1042 MPI_MAX_OBJECT_NAME, 18, 378-381, 852, 1044, 1050 MPI_MAX_PORT_NAME, 19, 530, 852 MPI_MAX_PROCESSOR_NAME, 18, 451, 852, 1051MPI_MAX_PSET_NAME_LEN, 501, 852 MPI_MAXLOC, 224, 227, 228, 231, 856 MPI_MESSAGE_NO_PROC, 86, 89, 90, 99, 851, 1043 MPI_MESSAGE_NULL, 86, 88-90, 856, 1043 MPI_MESSAGE_PROC_NULL, 87 MPI_MIN, 224, 225, 856 "mpi_minimum_memory_alignment", 452, 551, 554.1037 MPI_MINLOC, 224, 227, 228, 231, 856 MPI_MODE_APPEND, 638, 639, 859 MPI_MODE_CREATE, 638, 639, 648, 859 MPI_MODE_DELETE_ON_CLOSE, 638-640, 859 MPI_MODE_EXCL, 638, 639, 859 MPI_MODE_NOCHECK, 597, 602, 603, 859 MPI_MODE_NOPRECEDE, 592, 602, 603, 859 MPI_MODE_NOPUT, 602, 603, 859

1 MPI_MODE_NOSTORE, 602, 603, 859 MPI_MODE_NOSUCCEED, 602, 603, 859  $\mathbf{2}$ MPI_MODE_RDONLY, 638, 639, 644, 859 3 MPI MODE RDWR, 638, 639, 859 4 MPI_MODE_SEQUENTIAL, 638, 639, 642, 649, 5654, 662, 679, 704, 859, 1053 6 MPI_MODE_UNIQUE_OPEN, 638, 639, 859 7 MPI_MODE_WRONLY, 638, 639, 859 MPI_NO_OP, 548, 575-577, 856, 1039 MPI_NULL_COPY_FN, 23, 364, 774, 858 9 MPI_NULL_DELETE_FN, 23, 365, 774, 858 10 MPI_OFFSET, 33, 225, 694, 853, 854, 1047, 11 104912MPI_OFFSET_KIND, 19, 20, 33, 706, 810, 852 13 MPI_OP_NULL, 235, 856 14MPI_ORDER_C, 18, 132, 133, 136, 860 15MPI_ORDER_FORTRAN, 18, 132, 133, 136, 16860 MPI_PACKED, 15, 31, 32, 43, 44, 173, 175, 17180, 693, 694, 846, 853, 854 18 MPI_PROC_NULL, 30, 34, 36, 83, 86, 89, 90, 1998, 99, 190, 193, 195, 197, 206, 208, 20224, 313, 408, 412, 449, 450, 554, 555, 21564, 851, 1043, 1050, 1052 22MPI_PROD, 224, 225, 856 23MPI_REAL, 31, 43, 225, 693, 694, 801, 802,  24 808, 854 MPI_REAL16, 33, 225, 695, 854 25MPI_REAL2, <u>33</u>, 225, 695, 854 26MPI_REAL4, <u>33</u>, 225, 695, 801, 806, 854 27MPI_REAL8, 33, 225, 695, 801, 854, 1047 28 MPI_REPLACE, 572, 573, 575, 577, 620, 856, 291049, 1052 30 MPI_REQUEST_NULL, 68-71, 74-78, 628, 856 31MPI_ROOT, 190, 851 MPI_SEEK_CUR, 671, 679, 860 32 MPI_SEEK_END, 671, 679, 860 33 MPI_SEEK_SET, 671, 672, 679, 860 34 MPI_SESSION_NULL, 498, 856 35 "mpi_shared_memory", 336, 1036 36 MPI_SHORT, 32, 224, 694, 853 37 MPI_SHORT_INT, 228, 855 38 MPI_SIGNED_CHAR, 32, 225, 227, 694, 853, 39 1049 MPI_SIMILAR, 314, 323, 353, 855 40 "mpi_size", 499, 502, 506 41 MPI_SOURCE, 37, 248, 836, 852, 1039, 1045 42MPI_STATUS_IGNORE, 10, 19, 39, 40, 627, 43 654, 788, 810, 836, 837, 845, 860, 861, 441043 45MPI_STATUS_SIZE, 19, 37, 790, 852, 1045 46MPI_STATUSES_IGNORE, 18, 19, 39, 40, 627, 629, 810, 836, 837, 860, 861 47MPI_SUBARRAYS_SUPPORTED, 19, 784, 785, 48

1	788-790, 792, 796-799, 812, 813, 852,
2	1045
3	MPI_SUBVERSION, 19, <u>448</u> , 861
4	MPI_SUCCESS, 22, 23, 69, 76, 78, 364–366,
4	368–370, 372, 373, 390, 466, 467,
5	471-473, 485, 521, 700, 723, 733, 741,
6	
7	744–746, 751, 755, 756, 760, 767, 769,
8	771, 774, 849, 1037
	MPI_SUM, 224, 225, 572, 845, 856
9	MPI_T_BIND_MPI_COMM, <u>725</u> , 862
10	MPI_T_BIND_MPI_DATATYPE, <u>725</u> , 862
11	MPI_T_BIND_MPI_ERRHANDLER, <u>725</u> , 862
12	MPI_T_BIND_MPI_FILE, <u>725</u> , 862
13	MPI_T_BIND_MPI_GROUP, <u>725</u> , 862
	MPI_T_BIND_MPI_INFO, <u>725</u> , 862
14	MPI_T_BIND_MPI_MESSAGE, 725, 862
15	MPI_T_BIND_MPI_OP, <u>725</u> , <u>862</u>
16	MPI_T_BIND_MPI_REQUEST, <u>725</u> , 862
17	MPI_T_BIND_MPI_SESSION, <u>725</u>
18	MPI_T_BIND_MPI_WIN, <u>725</u> , 862
	MPI_T_BIND_NO_OBJECT, <u>725</u> , 732, 734,
19	741, 743, 755, 757, 862
20	MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE,
21	
22	$\frac{752}{100}$ , 863
23	MPI_T_CB_REQUIRE_MPI_RESTRICTED,
	<u>752</u> , 863
24	MPI_T_CB_REQUIRE_NONE, <u>752</u> , 863
25	MPI_T_CB_REQUIRE_THREAD_SAFE, <u>752</u> ,
26	863
27	MPI_T_CVAR_HANDLE_NULL, <u>735</u> , 861
28	MPI_T_ENUM_NULL, 731, 741, 755, 861
	MPI_T_ERR_CANNOT_INIT, <u>771</u> , 851
29	MPI_T_ERR_CVAR_SET_NEVER, <u>736</u> , 771,
30	851
31	MPI_T_ERR_CVAR_SET_NOT_NOW, <u>736</u> ,
32	771, 851
33	MPI_T_ERR_INVALID, <u>771</u> , 851, 1041
34	MPI_T_ERR_INVALID_HANDLE, <u>743</u> , 760,
	771, 851
35	MPI_T_ERR_INVALID_INDEX, 23, 751, 755,
36	<u>771</u> , 777, 851, 1038
37	MPI_T_ERR_INVALID_ITEM, 23, 777, 851,
38	1038
39	MPI_T_ERR_INVALID_NAME, 733, 741, 756,
40	767, <u>771</u> , 851, 1041
41	MPI_T_ERR_INVALID_SESSION, 771, 851
42	MPI_T_ERR_MEMORY, <u>771</u> , 851
43	MPI_T_ERR_NOT_ACCESSIBLE, <u>753</u> , 771, 851
44	MPI_T_ERR_NOT_INITIALIZED, 771, 851
	MPI_T_ERR_NOT_SUPPORTED, <u>751</u> , 771,
45	851
46	MPI_T_ERR_OUT_OF_HANDLES, <u>771</u> , 851
47	MPI_T_ERR_OUT_OF_SESSIONS, <u>771</u> , 851
48	

<ul> <li>851</li> <li>MPI_T_ERR_PVAR_NO_STARTSTOP, 744, 745, 771, 851</li> <li>MPI_T_ERR_PVAR_NO_WRITE, 746, 771, 851</li> <li>MPI_T_PVAR_ALL_HANDLES, 744, 745-747, 862</li> <li>MPI_T_PVAR_CLASS_AGGREGATE, 738, 739, 862</li> <li>MPI_T_PVAR_CLASS_COUNTER, 738, 862</li> <li>MPI_T_PVAR_CLASS_GENERIC, 739, 862</li> <li>MPI_T_PVAR_CLASS_HIGHWATERMARK, 738, 862</li> <li>MPI_T_PVAR_CLASS_LEVEL, 737, 862</li> <li>MPI_T_PVAR_CLASS_LEVEL, 737, 862</li> <li>MPI_T_PVAR_CLASS_PERCENTAGE, 738, 862</li> <li>MPI_T_PVAR_CLASS_SIZE, 737, 862</li> <li>MPI_T_SCOPE_ALL, 732, 862</li> <li>MPI_T_SCOPE_ALL_EQ, 732, 736, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, 732, 736, 862</li> <li>MPI_T_SCOPE_READONLY, 732, 862</li> <li>MPI_T_SCOPE_READONLY, 732, 862</li> <li>MPI_T_SCOPE_READONLY, 732, 863</li> <li>MPI_T_SOURCE_UNORDERED, 751, 863</li> <li>MPI_T_SOURCE_UNORDERED, 751, 863</li> <li>MPI_T_VERBOSITY_MPIDEV_BASIC, 724, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_BASIC, 724, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, 724, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, 724, 861</li> </ul>
<ul> <li>745, 771, 851</li> <li>MPI_T_ERR_PVAR_NO_WRITE, 746, 771, 851</li> <li>MPI_T_PVAR_ALL_HANDLES, 744, 745–747, 862</li> <li>MPI_T_PVAR_CLASS_AGGREGATE, 738, 739, 862</li> <li>MPI_T_PVAR_CLASS_COUNTER, 738, 862</li> <li>MPI_T_PVAR_CLASS_GENERIC, 739, 862</li> <li>MPI_T_PVAR_CLASS_HIGHWATERMARK, 738, 862</li> <li>MPI_T_PVAR_CLASS_LEVEL, 737, 862</li> <li>MPI_T_PVAR_CLASS_LOWWATERMARK, 738, 862</li> <li>MPI_T_PVAR_CLASS_LOWWATERMARK, 738, 862</li> <li>MPI_T_PVAR_CLASS_SIZE, 737, 862</li> <li>MPI_T_PVAR_CLASS_TIMER, 738, 862</li> <li>MPI_T_PVAR_CLASS_TIMER, 738, 862</li> <li>MPI_T_PVAR_CLASS_TIMER, 738, 862</li> <li>MPI_T_PVAR_BESSION_NULL, 742, 861</li> <li>MPI_T_SCOPE_ALL_EQ, 732, 736, 862</li> <li>MPI_T_SCOPE_GROUP, 732, 862</li> <li>MPI_T_SCOPE_CONSTANT, 732, 862</li> <li>MPI_T_SCOPE_READONLY, 732, 862</li> <li>MPI_T_SCOPE_CONSTANT, 732, 862</li> <li>MPI_T_SCOPE_READONLY, 732, 862</li> <li>MPI_T_SOURCE_UNORDERED, 751, 863</li> <li>MPI_T_SOURCE_UNORDERED, 751, 863</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, 724, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, 724, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, 724, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, 724, 861</li> </ul>
<ul> <li>MPI_T_ERR_PVAR_NO_WRITE, <u>746</u>, 771, 851</li> <li>MPI_T_PVAR_ALL_HANDLES, <u>744</u>, 745–747, 862</li> <li>MPI_T_PVAR_CLASS_AGGREGATE, <u>738</u>, 739, 862</li> <li>MPI_T_PVAR_CLASS_COUNTER, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_GENERIC, <u>739</u>, 862</li> <li>MPI_T_PVAR_CLASS_HIGHWATERMARK, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_LEVEL, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_LEVEL, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_PERCENTAGE, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_STATE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_STATE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>738</u>, 862</li> <li>MPI_T_PVAR_SESSION_NULL, <u>744</u>, 861</li> <li>MPI_T_SCOPE_ALL, <u>732</u>, 862</li> <li>MPI_T_SCOPE_CONSTANT, <u>732</u>, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_CONSTANT, <u>732</u>, 862</li> <li>MPI_T_SCOPE_CONSTANT, <u>732</u>, 862</li> <li>MPI_T_SCOPE_CONSTANT, <u>732</u>, 862</li> <li>MPI_T_SCOPE_READONLY, <u>732</u>, 862</li> <li>MPI_T_SOURCE_ORDERED, <u>751</u>, 863</li> <li>MPI_T_SOURCE_ORDERED, <u>751</u>, 863</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> </ul>
<ul> <li>MPI_T_PVAR_ALL_HANDLES, <u>744</u>, 745–747, 862</li> <li>MPI_T_PVAR_CLASS_AGGREGATE, <u>738</u>, 739, 862</li> <li>MPI_T_PVAR_CLASS_COUNTER, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_GENERIC, <u>739</u>, 862</li> <li>MPI_T_PVAR_CLASS_HIGHWATERMARK, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_LEVEL, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_LEVEL, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_PERCENTAGE, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_STATE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_STATE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_STATE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>738</u>, 862</li> <li>MPI_T_SCOPE_ALL, <u>732</u>, 862</li> <li>MPI_T_SCOPE_ALL, <u>732</u>, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_LOCAL, <u>732</u>, 862</li> <li>MPI_T_SCOPE_CONDERED, <u>751</u>, 863</li> <li>MPI_T_SOURCE_UNORDERED, <u>751</u>, 863</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> </ul>
862 MPI_T_PVAR_CLASS_AGGREGATE, <u>738</u> , 739, 862 MPI_T_PVAR_CLASS_COUNTER, <u>738</u> , 862 MPI_T_PVAR_CLASS_GENERIC, <u>739</u> , 862 MPI_T_PVAR_CLASS_HIGHWATERMARK, <u>738</u> , 862 MPI_T_PVAR_CLASS_LEVEL, <u>737</u> , 862 MPI_T_PVAR_CLASS_LEVEL, <u>737</u> , 862 MPI_T_PVAR_CLASS_PERCENTAGE, <u>738</u> , 862 MPI_T_PVAR_CLASS_SIZE, <u>737</u> , 862 MPI_T_PVAR_CLASS_SIZE, <u>737</u> , 862 MPI_T_PVAR_CLASS_STATE, <u>737</u> , 862 MPI_T_PVAR_CLASS_STATE, <u>737</u> , 862 MPI_T_PVAR_CLASS_TIMER, <u>738</u> , 862 MPI_T_PVAR_CLASS_TIMER, <u>738</u> , 862 MPI_T_PVAR_SESSION_NULL, <u>744</u> , 861 MPI_T_PVAR_SESSION_NULL, <u>742</u> , 861 MPI_T_SCOPE_ALL, <u>732</u> , 862 MPI_T_SCOPE_CONSTANT, <u>732</u> , 862 MPI_T_SCOPE_GROUP_EQ, <u>732</u> , 736, 862 MPI_T_SCOPE_LOCAL, <u>732</u> , 862 MPI_T_SCOPE_LOCAL, <u>732</u> , 862 MPI_T_SCOPE_RADONLY, <u>732</u> , 862 MPI_T_SOURCE_ORDERED, <u>751</u> , 863 MPI_T_SOURCE_ORDERED, <u>751</u> , 863 MPI_T_VERBOSITY_MPIDEV_ALL, <u>724</u> , 861 MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u> , 861 MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u> , 861 MPI_T_VERBOSITY_TUNER_ALL, <u>724</u> , 861
<ul> <li>MPI_T_PVAR_CLASS_AGGREGATE, <u>738</u>, 739, <u>862</u></li> <li>MPI_T_PVAR_CLASS_COUNTER, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_GENERIC, <u>739</u>, 862</li> <li>MPI_T_PVAR_CLASS_HIGHWATERMARK, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_LEVEL, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_LOWWATERMARK, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_LOWWATERMARK, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_STATE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>738</u>, 862</li> <li>MPI_T_PVAR_BESSION_NULL, <u>744</u>, 861</li> <li>MPI_T_SCOPE_ALL_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_CONSTANT, <u>732</u>, 862</li> <li>MPI_T_SCOPE_GROUP, <u>732</u>, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_CONSTANT, <u>732</u>, 862</li> <li>MPI_T_SCOPE_CONDERED, <u>751</u>, 863</li> <li>MPI_T_SOURCE_ORDERED, <u>751</u>, 863</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> </ul>
862 MPI_T_PVAR_CLASS_COUNTER, <u>738</u> , 862 MPI_T_PVAR_CLASS_GENERIC, <u>739</u> , 862 MPI_T_PVAR_CLASS_HIGHWATERMARK, <u>738</u> , 862 MPI_T_PVAR_CLASS_LEVEL, <u>737</u> , 862 MPI_T_PVAR_CLASS_LOWWATERMARK, <u>738</u> , 862 MPI_T_PVAR_CLASS_DERCENTAGE, <u>738</u> , 862 MPI_T_PVAR_CLASS_SIZE, <u>737</u> , 862 MPI_T_PVAR_CLASS_SIZE, <u>737</u> , 862 MPI_T_PVAR_CLASS_STATE, <u>737</u> , 862 MPI_T_PVAR_CLASS_TIMER, <u>738</u> , 862 MPI_T_PVAR_LASS_TIMER, <u>738</u> , 862 MPI_T_PVAR_LASS_TIMER, <u>738</u> , 862 MPI_T_PVAR_HANDLE_NULL, <u>744</u> , 861 MPI_T_SCOPE_ALL, <u>732</u> , 862 MPI_T_SCOPE_ALL_EQ, <u>732</u> , 736, 862 MPI_T_SCOPE_GROUP, <u>732</u> , 862 MPI_T_SCOPE_GROUP_EQ, <u>732</u> , 736, 862 MPI_T_SCOPE_LOCAL, <u>732</u> , 862 MPI_T_SCOPE_LOCAL, <u>732</u> , 862 MPI_T_SOURCE_ORDERED, <u>751</u> , 863 MPI_T_SOURCE_UNORDERED, <u>751</u> , 863 MPI_T_VERBOSITY_MPIDEV_ALL, <u>724</u> , 861 MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u> , 861 MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u> , 861 MPI_T_VERBOSITY_TUNER_ALL, <u>724</u> , 861
<ul> <li>MPI_T_PVAR_CLASS_COUNTER, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_GENERIC, <u>739</u>, 862</li> <li>MPI_T_PVAR_CLASS_HIGHWATERMARK, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_LEVEL, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_LOWWATERMARK, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_STATE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_STATE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_STATE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_STATE, <u>737</u>, 862</li> <li>MPI_T_PVAR_LANDLE_NULL, <u>744</u>, 861</li> <li>MPI_T_PVAR_SESSION_NULL, <u>742</u>, 861</li> <li>MPI_T_SCOPE_ALL_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_GROUP, <u>732</u>, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_LOCAL, <u>732</u>, 862</li> <li>MPI_T_SCOPE_READONLY, <u>732</u>, 862</li> <li>MPI_T_SOURCE_ORDERED, <u>751</u>, 863</li> <li>MPI_T_VERBOSITY_MPIDEV_BASIC, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> </ul>
<ul> <li>MPI_T_PVAR_CLASS_GENERIC, <u>739</u>, 862</li> <li>MPI_T_PVAR_CLASS_HIGHWATERMARK, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_LEVEL, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_LOWWATERMARK, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_PERCENTAGE, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_STATE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>738</u>, 862</li> <li>MPI_T_PVAR_BESSION_NULL, <u>744</u>, 861</li> <li>MPI_T_SCOPE_ALL_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_CONSTANT, <u>732</u>, 862</li> <li>MPI_T_SCOPE_CONDERED, <u>751</u>, 863</li> <li>MPI_T_SOURCE_ORDERED, <u>751</u>, 863</li> <li>MPI_T_VERBOSITY_MPIDEV_BASIC, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> </ul>
<ul> <li>MPI_T_PVAR_CLASS_HIGHWATERMARK, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_LEVEL, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_LOWWATERMARK, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_PERCENTAGE, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_STATE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_STATE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>738</u>, 862</li> <li>MPI_T_PVAR_SESSION_NULL, <u>744</u>, 861</li> <li>MPI_T_SCOPE_ALL_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_GROUP, <u>732</u>, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_CONSTANT, <u>732</u>, 862</li> <li>MPI_T_SCOPE_CONDERED, <u>751</u>, 863</li> <li>MPI_T_SOURCE_ORDERED, <u>751</u>, 863</li> <li>MPI_T_VERBOSITY_MPIDEV_BASIC, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> </ul>
738, 862         MPI_T_PVAR_CLASS_LEVEL, 737, 862         MPI_T_PVAR_CLASS_LOWWATERMARK,         738, 862         MPI_T_PVAR_CLASS_PERCENTAGE, 738, 862         MPI_T_PVAR_CLASS_SIZE, 737, 862         MPI_T_PVAR_CLASS_STATE, 738, 862         MPI_T_PVAR_CLASS_STATE, 738, 862         MPI_T_SCOPE_ALL_RQ, 732, 736, 862         MPI_T_SCOPE_GROUP, 732, 862         MPI_T_SCOPE_GROUP_EQ, 732, 736, 862         MPI_T_SCOPE_GROUP_EQ, 732, 736, 862         MPI_T_SCOPE_CONSTANT, 732, 862         MPI_T_SOURCE_ORDERED, 751, 863         MPI_T_SOURCE_UNORDERED, 751, 863         MPI_T_VERBOSITY_MPIDEV_ALL, 724, 861         MPI_T_VERBOSITY_MPIDEV_DETAIL, 724, 861         MPI_T_VERBOSITY_MPIDEV_DETAIL, 724, 861         MPI_T_VERBOSITY_TUNER_ALL, 724, 861
<ul> <li>MPI_T_PVAR_CLASS_LEVEL, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_LOWWATERMARK, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_PERCENTAGE, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_STATE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>738</u>, 862</li> <li>MPI_T_PVAR_ANDLE_NULL, <u>744</u>, 861</li> <li>MPI_T_SCOPE_ALL, <u>732</u>, 862</li> <li>MPI_T_SCOPE_ALL_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_GROUP, <u>732</u>, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_CONSTANT, <u>732</u>, 862</li> <li>MPI_T_SCOPE_READONLY, <u>732</u>, 862</li> <li>MPI_T_SOURCE_ORDERED, <u>751</u>, 863</li> <li>MPI_T_SOURCE_UNORDERED, <u>751</u>, 863</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> </ul>
<ul> <li>MPI_T_PVAR_CLASS_LOWWATERMARK, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_PERCENTAGE, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_STATE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>738</u>, 862</li> <li>MPI_T_PVAR_BANDLE_NULL, <u>744</u>, 861</li> <li>MPI_T_SCOPE_ALL, <u>732</u>, 862</li> <li>MPI_T_SCOPE_ALL_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_CONSTANT, <u>732</u>, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_LOCAL, <u>732</u>, 862</li> <li>MPI_T_SOURCE_ORDERED, <u>751</u>, 863</li> <li>MPI_T_SOURCE_UNORDERED, <u>751</u>, 863</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> </ul>
738, 862         MPI_T_PVAR_CLASS_PERCENTAGE, 738, 862         MPI_T_PVAR_CLASS_SIZE, 737, 862         MPI_T_PVAR_CLASS_STATE, 737, 862         MPI_T_PVAR_CLASS_STATE, 737, 862         MPI_T_PVAR_CLASS_TIMER, 738, 862         MPI_T_PVAR_CLASS_TIMER, 738, 862         MPI_T_PVAR_SESSION_NULL, 744, 861         MPI_T_SCOPE_ALL, 732, 862         MPI_T_SCOPE_CONSTANT, 732, 862         MPI_T_SCOPE_GROUP_EQ, 732, 736, 862         MPI_T_SCOPE_GROUP_EQ, 732, 862         MPI_T_SCOPE_CONSTANT, 732, 862         MPI_T_SCOPE_GROUP_EQ, 732, 736, 862         MPI_T_SCOPE_CONSTANT, 732, 862         MPI_T_SCOPE_READONLY, 732, 862         MPI_T_SOURCE_ORDERED, 751, 863         MPI_T_SOURCE_UNORDERED, 751, 863         MPI_T_VERBOSITY_MPIDEV_ALL, 724, 861         MPI_T_VERBOSITY_MPIDEV_DETAIL, 724, 861         MPI_T_VERBOSITY_TUNER_ALL, 724, 861         MPI_T_VERBOSITY_TUNER_ALL, 724, 861
<ul> <li>MPI_T_PVAR_CLASS_PERCENTAGE, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_STATE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>738</u>, 862</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>738</u>, 862</li> <li>MPI_T_PVAR_HANDLE_NULL, <u>744</u>, 861</li> <li>MPI_T_SCOPE_ALL, <u>732</u>, 862</li> <li>MPI_T_SCOPE_ALL_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 862</li> <li>MPI_T_SCOPE_LOCAL, <u>732</u>, 862</li> <li>MPI_T_SCOPE_READONLY, <u>732</u>, 862</li> <li>MPI_T_SOURCE_ORDERED, <u>751</u>, 863</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> </ul>
<ul> <li>MPI_T_PVAR_CLASS_SIZE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_STATE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>738</u>, 862</li> <li>MPI_T_PVAR_HANDLE_NULL, <u>744</u>, 861</li> <li>MPI_T_PVAR_SESSION_NULL, <u>742</u>, 861</li> <li>MPI_T_SCOPE_ALL_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_COALL, <u>732</u>, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_COAL, <u>732</u>, 862</li> <li>MPI_T_SCOPE_READONLY, <u>732</u>, 862</li> <li>MPI_T_SOURCE_ORDERED, <u>751</u>, 863</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> </ul>
<ul> <li>MPI_T_PVAR_CLASS_STATE, <u>737</u>, 862</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>738</u>, 862</li> <li>MPI_T_PVAR_HANDLE_NULL, <u>744</u>, 861</li> <li>MPI_T_PVAR_SESSION_NULL, <u>742</u>, 861</li> <li>MPI_T_SCOPE_ALL_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_CONSTANT, <u>732</u>, 862</li> <li>MPI_T_SCOPE_GROUP, <u>732</u>, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_COALL, <u>732</u>, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_READONLY, <u>732</u>, 862</li> <li>MPI_T_SOURCE_ORDERED, <u>751</u>, 863</li> <li>MPI_T_SOURCE_UNORDERED, <u>751</u>, 863</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> </ul>
<ul> <li>MPI_T_PVAR_CLASS_TIMER, <u>738</u>, 862</li> <li>MPI_T_PVAR_HANDLE_NULL, <u>744</u>, 861</li> <li>MPI_T_PVAR_SESSION_NULL, <u>742</u>, 861</li> <li>MPI_T_SCOPE_ALL, <u>732</u>, 862</li> <li>MPI_T_SCOPE_CONSTANT, <u>732</u>, 862</li> <li>MPI_T_SCOPE_GROUP, <u>732</u>, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 862</li> <li>MPI_T_SCOPE_READONLY, <u>732</u>, 862</li> <li>MPI_T_SOURCE_ORDERED, <u>751</u>, 863</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> </ul>
<ul> <li>MPI_T_PVAR_HANDLE_NULL, <u>744</u>, 861</li> <li>MPI_T_PVAR_SESSION_NULL, <u>742</u>, 861</li> <li>MPI_T_SCOPE_ALL_EQ, <u>732</u>, 862</li> <li>MPI_T_SCOPE_CONSTANT, <u>732</u>, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_LOCAL, <u>732</u>, 862</li> <li>MPI_T_SCOPE_READONLY, <u>732</u>, 862</li> <li>MPI_T_SOURCE_ORDERED, <u>751</u>, 863</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> </ul>
<ul> <li>MPI_T_PVAR_SESSION_NULL, <u>742</u>, 861</li> <li>MPI_T_SCOPE_ALL, <u>732</u>, 862</li> <li>MPI_T_SCOPE_ALL_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_CONSTANT, <u>732</u>, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_LOCAL, <u>732</u>, 862</li> <li>MPI_T_SOURCE_ORDERED, <u>751</u>, 863</li> <li>MPI_T_SOURCE_UNORDERED, <u>751</u>, 863</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> </ul>
<ul> <li>MPI_T_SCOPE_ALL, <u>732</u>, 862</li> <li>MPI_T_SCOPE_ALL_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_CONSTANT, <u>732</u>, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_LOCAL, <u>732</u>, 862</li> <li>MPI_T_SOURCE_ORDERED, <u>751</u>, 863</li> <li>MPI_T_SOURCE_UNORDERED, <u>751</u>, 863</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> </ul>
<ul> <li>MPI_T_SCOPE_ALL_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_CONSTANT, <u>732</u>, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_LOCAL, <u>732</u>, 862</li> <li>MPI_T_SOURCE_ORDERED, <u>751</u>, 863</li> <li>MPI_T_SOURCE_UNORDERED, <u>751</u>, 863</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> </ul>
MPI_T_SCOPE_CONSTANT, <u>732</u> , 862 MPI_T_SCOPE_GROUP, <u>732</u> , 862 MPI_T_SCOPE_GROUP_EQ, <u>732</u> , 736, 862 MPI_T_SCOPE_LOCAL, <u>732</u> , 862 MPI_T_SCOPE_READONLY, <u>732</u> , 862 MPI_T_SOURCE_ORDERED, <u>751</u> , 863 MPI_T_SOURCE_UNORDERED, <u>751</u> , 863 MPI_T_VERBOSITY_MPIDEV_ALL, <u>724</u> , 861 MPI_T_VERBOSITY_MPIDEV_BASIC, <u>724</u> , <u>861</u> MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u> , 861 MPI_T_VERBOSITY_TUNER_ALL, <u>724</u> , 861 MPI_T_VERBOSITY_TUNER_ALL, <u>724</u> , 861
<ul> <li>MPI_T_SCOPE_GROUP, <u>732</u>, 862</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>732</u>, 736, 862</li> <li>MPI_T_SCOPE_LOCAL, <u>732</u>, 862</li> <li>MPI_T_SOURCE_ORDERED, <u>751</u>, 863</li> <li>MPI_T_SOURCE_UNORDERED, <u>751</u>, 863</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>724</u>, 861</li> <li>MPI_T_VERBOSITY_TUNER_BASIC, <u>724</u>, 861</li> </ul>
MPI_T_SCOPE_GROUP_EQ, <u>732</u> , 736, 862 MPI_T_SCOPE_LOCAL, <u>732</u> , 862 MPI_T_SCOPE_READONLY, <u>732</u> , 862 MPI_T_SOURCE_ORDERED, <u>751</u> , 863 MPI_T_SOURCE_UNORDERED, <u>751</u> , 863 MPI_T_VERBOSITY_MPIDEV_ALL, <u>724</u> , 861 MPI_T_VERBOSITY_MPIDEV_BASIC, <u>724</u> , 861 MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u> , 861 MPI_T_VERBOSITY_TUNER_ALL, <u>724</u> , 861 MPI_T_VERBOSITY_TUNER_BASIC, <u>724</u> , 861
MPI_T_SCOPE_LOCAL, <u>732</u> , 862 MPI_T_SCOPE_READONLY, <u>732</u> , 862 MPI_T_SOURCE_ORDERED, <u>751</u> , 863 MPI_T_SOURCE_UNORDERED, <u>751</u> , 863 MPI_T_VERBOSITY_MPIDEV_ALL, <u>724</u> , 861 MPI_T_VERBOSITY_MPIDEV_BASIC, <u>724</u> , 861 MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u> , 861 MPI_T_VERBOSITY_TUNER_ALL, <u>724</u> , 861 MPI_T_VERBOSITY_TUNER_BASIC, <u>724</u> , 861
MPI_T_SCOPE_READONLY, <u>732</u> , 862 MPI_T_SOURCE_ORDERED, <u>751</u> , 863 MPI_T_SOURCE_UNORDERED, <u>751</u> , 863 MPI_T_VERBOSITY_MPIDEV_ALL, <u>724</u> , 861 MPI_T_VERBOSITY_MPIDEV_BASIC, <u>724</u> , 861 MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u> , 861 MPI_T_VERBOSITY_TUNER_ALL, <u>724</u> , 861 MPI_T_VERBOSITY_TUNER_BASIC, <u>724</u> , 861
MPI_T_SOURCE_ORDERED, <u>751</u> , 863 MPI_T_SOURCE_UNORDERED, <u>751</u> , 863 MPI_T_VERBOSITY_MPIDEV_ALL, <u>724</u> , 861 MPI_T_VERBOSITY_MPIDEV_BASIC, <u>724</u> , 861 MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u> , 861 MPI_T_VERBOSITY_TUNER_ALL, <u>724</u> , 861 MPI_T_VERBOSITY_TUNER_BASIC, <u>724</u> , 861
MPI_T_SOURCE_UNORDERED, <u>751</u> , 863 MPI_T_VERBOSITY_MPIDEV_ALL, <u>724</u> , 861 MPI_T_VERBOSITY_MPIDEV_BASIC, <u>724</u> , 861 MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u> , 861 MPI_T_VERBOSITY_TUNER_ALL, <u>724</u> , 861 MPI_T_VERBOSITY_TUNER_BASIC, <u>724</u> , 861
MPI_T_VERBOSITY_MPIDEV_BASIC, <u>724</u> , 861 MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u> , 861 MPI_T_VERBOSITY_TUNER_ALL, <u>724</u> , 861 MPI_T_VERBOSITY_TUNER_BASIC, <u>724</u> , 861
861 MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u> , 861 MPI_T_VERBOSITY_TUNER_ALL, <u>724</u> , 861 MPI_T_VERBOSITY_TUNER_BASIC, <u>724</u> , 861
MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>724</u> , 861 MPI_T_VERBOSITY_TUNER_ALL, <u>724</u> , 861 MPI_T_VERBOSITY_TUNER_BASIC, <u>724</u> , 861
861 MPI_T_VERBOSITY_TUNER_ALL, <u>724</u> , 861 MPI_T_VERBOSITY_TUNER_BASIC, <u>724</u> , 861
MPI_T_VERBOSITY_TUNER_ALL, <u>724</u> , 861 MPI_T_VERBOSITY_TUNER_BASIC, <u>724</u> , 861
MPI_T_VERBOSITY_TUNER_BASIC, 724, 861
MPI_I_VERBOSITY_TUNER_DETAIL, $724$ ,
0.01
MPI_T_VERBOSITY_USER_ALL, <u>724</u> , 861
MPI_T_VERBOSITY_USER_BASIC, <u>724</u> , 861 MPI_T_VERBOSITY_USER_DETAIL, <u>724</u> , 861
MPI_TAG, <u>37</u> , 248, 836, 852, 1039, 1045 MPI_TAG_UB, 34, <u>449</u> , 841, 844, 855
MPI_TAG_0B, 54, <u>449</u> , 841, 844, 855 MPI_THREAD_FUNNELED, 487, 488, 859
"MPI_THREAD_FUNNELED", 497
MPI_THREAD_MULTIPLE, 487, 488, 490, 859,
1040
1010
"MPL THREAD MULTIPLE" 497
"MPI_THREAD_MULTIPLE", 497 MPI_THREAD_SERIALIZED, 487, 488, 859
MPI_THREAD_SERIALIZED, 487, 488, 859

MPI_TYPE_DUP_FN, <u>372</u>, 857 MPI_TYPE_NULL_COPY_FN, <u>372</u>, 857 MPI_TYPE_NULL_DELETE_FN, <u>372</u>, 857, 1046 MPI_TYPECLASS_COMPLEX, 807, 860 MPI_TYPECLASS_INTEGER, 807, 860 MPI_TYPECLASS_REAL, 807, 860 MPI_UB, 4, 23, <u>780</u>, 1042 MPI_UINT16_T, 32, 225, 694, 853, 1047, 1049 MPI_UINT32_T, 32, 225, 694, 728, 853, 1047, 1049MPI_UINT64_T, <u>32</u>, 225, 694, 728, 853, 1047, 1049MPI_UINT8_T, 32, 225, 694, 853, 1047, 1049 MPI_UNDEFINED, 21, 38, 39, 57, 74, 75, 78, 79, 142, 146, 148, 153, 176, 312, 313, 331, 332, 335, 399, 410, 411, 693, 698, 803, 804, 851, 1043, 1049 MPI_UNEQUAL, <u>314</u>, 323, 353, 855 MPI_UNIVERSE_SIZE, 517, 539, 540, 858 MPI_UNSIGNED, <u>32</u>, 224, 694, 728, 737-739, 853 MPI_UNSIGNED_CHAR, <u>32</u>, 225, 227, 694, 853 MPI_UNSIGNED_LONG, 32, 225, 694, 728, 737-739, 853 MPI_UNSIGNED_LONG_LONG, 32, 225, 694, 728, 737–739, 853, 1049 MPI_UNSIGNED_SHORT, <u>32</u>, 224, 694, 853 MPI_UNWEIGHTED, 19, <u>394</u>, <u>396–398</u>, 405, 407, 810, 860, 1042, 1048 MPI_VAL, 16, 833 MPI_VERSION, 19, 448, 861 MPI_WCHAR, 32, 227, 381, 693, 694, 853, 1049 MPI_WEIGHTS_EMPTY, 19, <u>394</u>, 396, 810, 860, 1042 MPI_WIN_BASE, <u>560</u>, 844, 858 MPI_WIN_CREATE_FLAVOR, 560, 858 MPI_WIN_DISP_UNIT, 560, 858 MPI_WIN_DUP_FN, 369, 857 MPI_WIN_FLAVOR_ALLOCATE, 561, 858 MPI_WIN_FLAVOR_CREATE, 561, 858 MPI_WIN_FLAVOR_DYNAMIC, 561, 858 MPI_WIN_FLAVOR_SHARED, 555, 561, 858 MPI_WIN_MODEL, 560, 587, 858 MPI_WIN_NULL, <u>559</u>, 856 MPI_WIN_NULL_COPY_FN, <u>369</u>, 857 MPI_WIN_NULL_DELETE_FN, 369, 857 MPI_WIN_SEPARATE, 561, 586, 587, 606, 858 MPI_WIN_SIZE, 560, 858 MPI_WIN_UNIFIED, 561, 587, 606, 615, 858 MPI_WTIME_IS_GLOBAL, 449, 450, 474, 841, 855 "mpix://UNIVERSE", 498

## **MPI** Declarations Index

10

 $11 \\ 12$ 

This index refers to declarations needed in C and Fortran, such as address kind integers, handles, etc. The underlined page numbers is the "main" reference (sometimes there are more than one when key concepts are discussed in multiple areas). Fortran defined types are shown as TYPE(*name*).

13	MPI_Aint, <u>20</u> , 20, 33, <u>119</u> , 122, 125, 129,
14	139-141, 145-147, 157, 180-182, 547,
15	550, 552, 556, 557, 560, 564, 567, 571,
16	573, 576-578, 580, 581, 583, 692, 697,
17	751,810,841,853,863
	MPI_Comm, 16, <u>30</u> , 314, 321–326, <u>329</u> , 331,
18	334, 341 - 343, 353 - 357, 363, 366, 367,
19	500,855,856,863,864
20	TYPE(MPI_Comm), 797, 855, 856, 864
21	MPI_Count, <u>20</u> , 21, 33, 749, 751, 853, 863, 1042
22	MPI_Datatype, <u>119</u> , 372, 373, 819, 853–856,
23	863, 864
24	TYPE(MPI_Datatype), 853-856, 864
	MPI_Errhandler, <u>457</u> , 458–465, 834, 852, 856,
25	863, 864
26	TYPE(MPI_Errhandler), 852, 856, 864
27	MPI_F08_status, <u>836</u> , <u>861</u> , <u>863</u> , <u>1045</u>
28	MPI_File, 461, 462, <u>637</u> , 640–643, 645, 646,
29	648,650,654660,662666,668675,
30	$677-679,\ 681-688,\ 692,\ 702-704,\ 834,$
31	856, 863, 864
32	TYPE(MPI_File), 856, 864
33	MPI_Fint, <u>833</u> , 861, 863, 1049
34	MPI_Group, <u>312</u> , 313–320, 354, 500, 561, 592,
	593, 643, 834, 856, 863, 864
35	TYPE(MPI_Group), 856, 864
36	MPI_Info, 451, <u>475</u> , 476–481, 499, 502, 517,
37	521, 522, 530, 534-536, 543, 562, 637,
38	640, 645, 646, 648, 776, 834, 855, 856,
39	863, 864, 1052
40	TYPE(MPI_Info), 855, 856, 864
41	MPI_Message, <u>85</u> , 834, 851, 856, 863, 864, 1043
42	TYPE(MPI_Message), 851, 856, 864
	MPI_Offset, <u>20</u> , 20, 33, 641–643, 648, 650,
43	654-660, 670-672, 679, 681, 683, 698,
44	<u>706</u> , 751, 833, 853, 863
45	MPI_Op, 222, <u>231</u> , 234, 236, 238–241, 243, 245, 246, 267, 269, 271, 273, 202, 204
46	246, 267-269, 271-273, 292-294, 296-298, 571, 573, 576, 581, 583, 834,
47	856, 863, 864
48	000, 000, 004

TYPE(MPI_Op), 856, 864 MPI_Request, 61-65, 67, 69, 70, 71, 74-77, 79, 81, 90-97, 104-108, 466, 626, 629, 657-660, 666, 668-670, 674, 675, 812, 834, 856, 863, 864 TYPE(MPI_Request), 856, 864 MPI_Session, 463, 464, 834, 856, 863, 864 TYPE(MPI_Session), 864 MPI_Status, 34, 37, 39-41, 69, 70, 74-77, 79, 81-83, 85, 87, 91, 152, 153, 627, 632, 633, 654-657, 662-665, 673, 674, 677, 678, 682, 684-688, 787, 835-838, 860, 863, 864, 1039, 1042, 1045 TYPE(MPI_Status), 37, 797, 836-838, 860, 864, 1035, 1039, 1045 MPI_T_cb_safety, 863 MPI_T_cvar_handle, 734, 735, 861 MPI_T_enum, <u>728</u>, 729, 730, 739, 754, 861 MPI_T_event_instance, 759 MPI_T_event_registration, 756 MPI_T_pvar_handle, <u>742</u>, <u>743–746</u>, <u>861</u>, <u>862</u> MPI_T_pvar_session, <u>742</u>, 743-746, 861 MPI_T_source_order, 863 MPI_Win, 369-371, 459, 460, 547, 550, 552, 556, 559, 561, 562, 564, 567, 571, 573, 576-578, 580, 581, 583, 590, 592-594, 596-601, 834, 856, 863, 864 TYPE(MPI_Win), 856, 864

# 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, 23	MPI_Win_errhandler_fn, 776, 1048	15
COMM_DELETE_ATTR_FUNCTION, 23	MPI_Win_errhandler_function, 459, 776, 865,	16
COPY_FUNCTION, 23	1048	17
		18
DELETE_FUNCTION, 23		19
MDL Comm. comu atta function 22, 22, 262		
MPI_Comm_copy_attr_function, 22, 23, <u>363</u> , 364, 786, 857, 864		20
MPI_Comm_delete_attr_function, 23, <u>363</u> , <u>365</u> ,		21
857, 864		22
MPI_Comm_errhandler_fn, 776, 1048		23
MPI_Comm_errhandler_function, 23, 457, 776,		24
780, 865, 1048		25
MPI_Copy_function, 23, 773, 858, 871		26
MPI_Datarep_conversion_function, 698, 857, 865		27
MPI_Datarep_conversion_function_c, 698, 865		28
MPI_Datarep_extent_function, 697, 865		29
MPI_Delete_function, 23, 774, 858, 871		
MPI_File_errhandler_fn, 776, 1048		30
MPI_File_errhandler_function, 461, 776, 865,		31
1048		32
MPI_Grequest_cancel_function, 628, 865		33
MPI_Grequest_free_function, 627, 865		34
MPI_Grequest_query_function, 627, 865		35
MPI_Handler_function, 23, <u>780</u> , 1042		36
MPI_NULL_COPY_FN, 22		37
MPI_Session_errhandler_function, 463, 865		38
MPI_T_event_cb_function, 758, 865		39
$MPI_T_event_dropped_function, 761, 865$		
MPI_T_event_free_cb_function, 760, 865		40
MPI_Type_copy_attr_function, <u>372</u> , 857, 864		41
MPI_Type_delete_attr_function, <u>372</u> , <u>373</u> , 857,		42
865, 1046		43
MPI_User_function, 232, 236, 864		44
MPI_User_function_c, 232, 864		45
MPI_Win_copy_attr_function, <u>368, 369</u> , 857,		46
864		47
MPI_Win_delete_attr_function, <u>368</u> , <u>369</u> , <u>857</u> ,		48
864		48

# **MPI** Function Index

10

The underlined page numbers refer to the function definitions.

11	MPI_ABORT, 233, 455, 485, 491, <u>508</u> , 509,
12	542, 727, 833, 1052
13	MPI_ACCUMULATE, 545, 563, <u>571</u> , 572, 576,
14	583, 587, 612, 618, 620, 1049, 1052
15	MPI_Accumulate_c, 571
16	MPI_ADD_ERROR_CLASS, <u>469</u> , 470
17	MPI_ADD_ERROR_CODE, <u>470</u> MPI_ADD_ERROR_STRING, <u>471</u>
18	MPI_ADDRESS, 23, <u>779</u> , 796, 1042
19	MPI_AINT_ADD, 24, 139, <u>140</u> , 557, 1040
20	MPI_AINT_DIFF, 24, 139, 140, <u>141</u> , 557, 1040
21	MPI_ALLGATHER, 185, 189, 190, <u>211</u> , 212,
22	214, 216, 259
23	MPI_Allgather_c, 211
24	MPI_ALLGATHER_INIT, <u>284</u>
	MPI_Allgather_init_c, 284
25	MPI_ALLGATHERV, 185, 189, 190, <u>213</u> , 214,
26	261
27	MPI_Allgatherv_c, 213
28	MPI_ALLGATHERV_INIT, <u>285</u> MPI_Allgatherv_init_c, <u>285</u>
29	MPI_ALLOC_MEM, <u>451</u> , 452–454, 467, 549,
30	551, 553-555, 558, 566, 599, 796-798,
31	810, 1037, 1039, 1046
32	MPI_ALLOC_MEM_CPTR, 452, 1039
33	MPI_ALLREDUCE, 185, 188–190, 224, 232,
34	$\underline{236}, 237, 269, 1050$
35	MPI_Allreduce_c, 237
36	MPI_ALLREDUCE_INIT, 293
37	MPI_Allreduce_init_c, 293
38	MPI_ALLTOALL, 185, 189, 190, <u>215</u> , 216, 218, 210, 202, 1047
39	218, 219, 262, 1047 MPI_Alltoall_c, 215
40	MPI_ALLTOALL_INIT, <u>287</u>
41	MPI_Alltoall_init_c, 287
42	MPI_ALLTOALLV, 185, 189, 190, <u>217</u> , 218,
43	219, 221, 264, 1047
44	MPI_Alltoallv_c, 217
45	MPI_ALLTOALLV_INIT, $\underline{288}$
46	MPI_Alltoallv_init_c, 288
47	MPI_ALLTOALLW, 185, 189, 190, <u>219</u> , 220,
48	221, 266, 1047
10	

MPI_Alltoallw_c, 220 MPI_ALLTOALLW_INIT, 290 MPI_Alltoallw_init_c, 290 MPI_ATTR_DELETE, 23, 375, 774, 776 MPI_ATTR_GET, 23, 375, 775, 841 MPI_ATTR_PUT, 23, 375, 775, 841, 844, 845 MPI_BARRIER, 185, 189, 191, 250, 508, 609-611, 708 MPI_BARRIER_INIT, 275 MPI_BCAST, 14, 185, 189, 192, 193, 223, 251, 302, 303 MPI_Bcast_c, 192 MPI_BCAST_INIT, 14, 276 MPI_Bcast_init_c, 276 MPI_BSEND, 14, 48, 49, 59 MPI_Bsend_c, 49 MPI_BSEND_INIT, 93, 97 MPI_Bsend_init_c, 93 MPI_BUFFER_ATTACH, 56, 69 MPI_Buffer_attach_c, 56 MPI_BUFFER_DETACH, <u>57</u>, 1036, 1045 MPI_Buffer_detach_c, 57 MPI_CANCEL, 23, 52, 70, 82, 90, 91, 92, 249, 275, 492, 578, 625, 628, 629, 776, 1036 MPI_CART_COORDS, 387, 402, 403, 1051 MPI_CART_CREATE, 352, 387, 388, 389-391, 411, 412, 811, 1039, 1050 MPI_CART_GET, 387, 401, 1051 MPI_CART_MAP, 387, 410, 411, 1044 MPI_CART_RANK, 387, 402, 1051 MPI_CART_SHIFT, 387, 407, 408, 412, 1051 MPI_CART_SUB, 387, 409, 411, 1051 MPI_CARTDIM_GET, 387, 400, 401, 1051 MPI_CLOSE_PORT, <u>531</u>, 535 MPI_COMM_ACCEPT, 529, 530, 532, 533, 540, 541 MPI_COMM_C2F, 834 MPI_COMM_CALL_ERRHANDLER, 471, 473 MPI_COMM_COMPARE, 323, 353 MPI_COMM_CONNECT, 468, 533, 540, 541 MPI_COMM_CREATE, 321, 323, 327,

328 - 332,  387,  1048	375, 774, 792, 841, 845
MPI_COMM_CREATE_ERRHANDLER, 23,	MPI_COMM_SET_ERRHANDLER, 23, 456,
$456, \underline{457}, 458, 779, 867, 870, 1046$	458,779
MPI_COMM_CREATE_FROM_GROUP, 323,	MPI_COMM_SET_INFO, 275, 341, <u>342</u> , 343,
<u>339</u> , 340, 357, 449	1037, 1039, 1043
MPI_COMM_CREATE_GROUP, 323, <u>330</u> ,	MPI_COMM_SET_NAME, <u>378</u>
331, 332, 340, 1044	MPI_COMM_SIZE, <u>321</u> , 322, 353, 618
MPI_COMM_CREATE_KEYVAL, 23, 361,	MPI_COMM_SPAWN, 485, 486, 511, 516, 517,
$\underline{362}, 364, 365, 375, 773, 840, 841, 866,$	518, 519-522, 524-526, 540, 541, 1038
869,1046,1050	MPI_COMM_SPAWN_MULTIPLE, 485, 486,
MPI_COMM_DELETE_ATTR, 23, 361, 365,	511, 516, 517, 522, 523, 524, 541, 1038
$366, \underline{368}, 375, 775$	MPI_COMM_SPLIT, 323, 327, 328, <u>331</u> , 332,
MPI_COMM_DISCONNECT, 375, 498, 508,	333, 383, 387, 389, 391, 409, 411, 412,
522, 542, 543	1048
MPI_COMM_DUP, 314, 321, 323, <u>324</u> , 325,	MPI_COMM_SPLIT_TYPE, 323, <u>335</u> , 338,
	341, 1036, 1044
326, 328, 355, 358, 361, 364, 368, 375, 368, 375, 368, 375, 368, 375, 368, 375, 368, 375, 368, 375, 368, 375, 368, 375, 368, 375, 368, 375, 375, 375, 375, 375, 375, 375, 375	
383, 773, 781, 1037, 1043	MPI_COMM_TEST_INTER, 352, 353
MPI_COMM_DUP_FN, 23, <u>364</u> , 365, 791,	MPI_COMPARE_AND_SWAP, 545, 563, <u>577</u> ,
857, 1040, 1046	618
MPI_COMM_DUP_WITH_INFO, 323, <u>325</u> ,	MPI_CONVERSION_FN_NULL, 700, 857,
326, 341, 781, 1043	1040
MPI_COMM_F2C, <u>834</u>	MPI_DATAREP_CONVERSION_FUNCTION,
MPI_COMM_FREE, 321, 324, 341, 355, 358,	698, 831
365, 366, 368, 375, 491, 495, 498, 508,	MPI_Datarep_conversion_function_c, 698, 865
522, 542, 543, 774	MPI_DIMS_CREATE, 387, <u>389</u> , 390, 1037
MPI_COMM_FREE_KEYVAL, 23, 361, <u>365</u> ,	MPI_DIST_GRAPH_CREATE, 341, 387, 392,
375, 774	<u>395,</u> 396–398, 407, 412, 1048
MPI_COMM_GET_ATTR, 23, 361, 366, <u>367</u> ,	MPI_DIST_GRAPH_CREATE_ADJACENT,
375, 449, 775, 792, 841, 844	$341, 387, 392, \underline{393}, 394, 398, 407, 412,$
MPI_COMM_GET_ERRHANDLER, 23, 456,	1044, 1048
$\underline{458}, 779, 1051$	MPI_DIST_GRAPH_NEIGHBORS, 387, 405,
MPI_COMM_GET_INFO, 341, <u>343</u> , 1037,	406, 407, 412, 1044, 1048
1043	MPI_DIST_GRAPH_NEIGHBORS_COUNT,
MPI_COMM_GET_NAME, 378, <u>379</u> , 380,	387, <u>405</u> , 407, 1041, 1048
1050	MPI_DUP_FN, 23, 364, <u>774</u> , 858
MPI_COMM_GET_PARENT, 379, 485, 486,	MPI_ERRHANDLER_C2F, 509, <u>834</u>
491, 496, 519, <u>522</u>	MPI_ERRHANDLER_CREATE, 23, <u>779</u> ,
MPI_COMM_GROUP, 17, 312, 314, <u>315</u> , 321,	1042, 1046
322, 353, 456, 1051	MPI_ERRHANDLER_F2C, 509, <u>834</u>
MPI_COMM_IDUP, 321, 323, <u>325</u> , 326, 351,	$MPI_ERRHANDLER_FREE, 456, \underline{465}, 491,$
361,  364,  368,  375,  781,  1037,  1039,	498,509,1051
1043	MPI_ERRHANDLER_GET, 23, <u>779</u> , 1042,
MPI_COMM_IDUP_WITH_INFO, 323, 326,	1051
341, 781, 1037	MPI_ERRHANDLER_SET, 23, 779, 1042
MPI_COMM_JOIN, 496, 543, 544	MPI_ERROR_CLASS, <u>466</u> , 469, 509
MPI_COMM_NULL_COPY_FN, 23, 364, 365,	MPI_ERROR_STRING, <u>465</u> , 466, 469, 471,
· · · · · · · · · · · · · · · · · · ·	
786, 840, 857, 1040, 1046	509 MDI EVENTE CALLDACK CETT INFO 750
MPI_COMM_NULL_DELETE_FN, 23, <u>365</u> ,	MPI_EVENT_CALLBACK_GET_INFO, 759
857, 1040	MPI_EVENT_HANDLE_GET_INFO, 757
MPI_COMM_RANK, 14, <u>322</u> , 353, 794	MPI_EXSCAN, 186, 189, 224, 232, <u>245</u> , 274,
MPI_COMM_RANK_F08, 794	1048
MPI_COMM_REMOTE_GROUP, <u>354</u>	MPI_Exscan_c, 245
MPI_COMM_REMOTE_SIZE, <u>354</u>	MPI_EXSCAN_INIT, <u>298</u>
MPI_COMM_SET_ATTR, 23, 361, 365, <u>366</u> ,	MPI_Exscan_init_c, 298

1

 $\mathbf{2}$ 3

4 $\mathbf{5}$ 

6  $\overline{7}$ 

8

9

10

11

1213

1415

16

17

1819

20

 21 

22

 23 

 24 

25

26

27

 28 

29

30

31

3233

3435

36 37

38

39

40

 41 42

43

44

45

46

47

1 MPI_F_SYNC_REG, 140, 620, 784, 800, 801,  $\mathbf{2}$ 820, 822-825, 1046 MPI_FETCH_AND_OP, 545, 563, 572, 576 3 MPI_FILE_C2F, 834 4 MPI_FILE_CALL_ERRHANDLER, 472, 473 5MPI_FILE_CLOSE, 543, 637, 638, 640 6 MPI_FILE_CREATE_ERRHANDLER, 456, 7 <u>461</u>, 462, 867, 870, 1046 8 MPI_FILE_DELETE, 639, 640, 641, 645, 647, 9 712 MPI_FILE_F2C, <u>834</u> 10 MPI_FILE_GET_AMODE, 644 11 MPI_FILE_GET_ATOMICITY, 703, 704 12MPI_FILE_GET_BYTE_OFFSET, 662, 672, 13680 14MPI_FILE_GET_ERRHANDLER, 456, 462, 15712.1051 16MPI_FILE_GET_GROUP, 643 17MPI_FILE_GET_INFO, 645, 646, 647, 1037, 105218MPI_FILE_GET_POSITION, 671, 672 19MPI_FILE_GET_POSITION_SHARED, 679, 20680, 704 21MPI_FILE_GET_SIZE, 643, 707 22MPI_FILE_GET_TYPE_EXTENT, 691, 692, 23700, 1036  24 MPI_File_get_type_extent_c, 692 MPI_FILE_GET_VIEW, 650, 651 25MPI_FILE_IREAD, 651, 667, 680, 701, 702 26MPI_FILE_IREAD_ALL, 651, 668, 669, 1041 27MPI_File_iread_all_c, 668 28MPI_FILE_IREAD_AT, 651, 658  29 MPI_FILE_IREAD_AT_ALL, 651, 659, 1041 30MPI_File_iread_at_all_c, 659  31 MPI_File_iread_at_c, 658 32MPI_File_iread_c, 667 MPI_FILE_IREAD_SHARED, 651, 675 33 MPI_File_iread_shared_c, 675 34MPI_FILE_IWRITE, 651, 669, 670 35MPI_FILE_IWRITE_ALL, 651, 670, 1041 36 MPI_File_iwrite_all_c, 670 37 MPI_FILE_IWRITE_AT, 651, 660 38MPI_FILE_IWRITE_AT_ALL, 651, 661, 1041 39MPI_File_iwrite_at_all_c, 661 MPI_File_iwrite_at_c, 660 40MPI_File_iwrite_c, 669 41 MPI_FILE_IWRITE_SHARED, 651, 676 42MPI_File_iwrite_shared_c, 676 43MPI_FILE_OPEN, 467, 515, 637, 638, 639, 44645, 647, 649, 672, 706, 707, 712, 713 45MPI_FILE_PREALLOCATE, 641, 642, 702, 46707 47MPI_FILE_READ, 651, 652, 662, 663, 664, 667, 706, 707 48

MPI_FILE_READ_ALL, 651, 664, 669, 681 MPI_FILE_READ_ALL_BEGIN, 651, 680, 681, <u>684</u>, 701, 825 MPI_File_read_all_begin_c, 684 MPI_File_read_all_c, 664 MPI_FILE_READ_ALL_END, 651, 680, 681, 685, 701, 825 MPI_FILE_READ_ALL_FN, 14 MPI_FILE_READ_AT, 651, 654, 655, 656, 658 MPI_FILE_READ_AT_ALL, 651, 655, 656, 659 MPI_FILE_READ_AT_ALL_BEGIN, 651, 682,825 MPI_File_read_at_all_begin_c, 682 MPI_File_read_at_all_c, 655 MPI_FILE_READ_AT_ALL_END, 651, 682, 825 MPI_FILE_READ_AT_ALL_FN, 14 MPI_File_read_at_c, 654 MPI_File_read_c, 662 MPI_FILE_READ_ORDERED, 651, 677, 678 MPI_FILE_READ_ORDERED_BEGIN, 651, 687, 825 MPI_File_read_ordered_begin_c, 687 MPI_File_read_ordered_c, 677 MPI_FILE_READ_ORDERED_END, 651, 687,825 MPI_FILE_READ_ORDERED_FN, 14 MPI_FILE_READ_SHARED, 651, 673, 674, 675,678 MPI_File_read_shared_c, 673 MPI_FILE_SEEK, 671, 672 MPI_FILE_SEEK_SHARED, 679, 680, 704 MPI_FILE_SET_ATOMICITY, 639, 702, 703 MPI_FILE_SET_ERRHANDLER, 456, 462, 712 MPI_FILE_SET_INFO, 645, 646, 647, 1037, 1052 MPI_FILE_SET_SIZE, <u>641</u>, 642, 702, 705, 707 MPI_FILE_SET_VIEW, 134, 468, 638, 645-647, 648, 649, 650, 672, 680, 690, 697, 706, 713, 1052, 1053 MPI_FILE_SYNC, 640, 652, 701, 702, 704, 709 MPI_FILE_WRITE, 651, 652, 665, 666, 670, 706 MPI_FILE_WRITE_ALL, 651, 666, 670 MPI_FILE_WRITE_ALL_BEGIN, 651, 685, 812, 825 MPI_File_write_all_begin_c, 685 MPI_File_write_all_c, 666 MPI_FILE_WRITE_ALL_END, 651, 686, 825 MPI_FILE_WRITE_ALL_FN, 14 MPI_FILE_WRITE_AT, 651, 652, 656, 657, 660

MPI_FILE_WRITE_AT_ALL, 651, 657, 661 MPI_FILE_WRITE_AT_ALL_BEGIN, 651, <u>683</u>, 825 MPI_File_write_at_all_begin_c, 683 MPI_File_write_at_all_c, 657 MPI_FILE_WRITE_AT_ALL_END, 651, 684, 825 MPI_FILE_WRITE_AT_ALL_FN, 14 MPI_File_write_at_c, 656 MPI_File_write_c, 665 MPI_FILE_WRITE_ORDERED, 651, 677, 678 MPI_FILE_WRITE_ORDERED_BEGIN, 651, 688, 825 MPI_File_write_ordered_begin_c, 688 MPI_File_write_ordered_c, 678 MPI_FILE_WRITE_ORDERED_END, 651, 689, 825 MPI_FILE_WRITE_ORDERED_FN, 14 MPI_FILE_WRITE_SHARED, 651, 652, 674, 676-678 MPI_File_write_shared_c, 674 MPI_FINALIZE, 18, 25, 26, 449, 455, 483, 490, 491-495, 514, 542, 543, 638, 723, 734, 747, 749, 833, 836, 837, 1038, 1044, 1052 MPI_FINALIZED, 493, 494, 495, 509, 833, 1040 MPI_FREE_MEM, 453, 467, 551, 553 MPI_GATHER, 185, 188, 189, <u>194</u>, 197, 198, 205, 212, 223, 253 MPI_Gather_c, 194 MPI_GATHER_INIT, 277 MPI_Gather_init_c, 277 MPI_GATHERV, 185, 189, 196, 197-199, 207, 214, 255 MPI_Gatherv_c, 196 MPI_GATHERV_INIT, 279 MPI_Gatherv_init_c, 279 MPI_GET, 545, 563, 567, 568, 576, 581, 587, 609, 611, 612, 620, 823, 1052 MPI_GET_ACCUMULATE, 545, 563, 572, 574, 575, 576, 585, 612, 618, 1039 MPI_Get_accumulate_c, 574 MPI_GET_ADDRESS, 23, 119, 139, 140, 141, 155, 557, 779, 796, 814, 819, 838-840 MPI_Get_c, 567 MPI_GET_COUNT, 37, 38, 39, 69, 153, 154, 578, 633, 654, 1043, 1049 MPI_Get_count_c, 38 MPI_GET_ELEMENTS, 69, 152, 153, 154, 633, 634, 654, 1043 MPI_Get_elements_c, 152 MPI_GET_ELEMENTS_X, 69, 152, 153, 154,

633,654,1042	1
MPI_GET_LIBRARY_VERSION, <u>448</u> , 509,	2
1039, 1040, 1042	3
MPI_GET_PROCESSOR_NAME, <u>450</u> , 451,	4
1051	5
MPI_GET_VERSION, <u>447</u> , 448, 509, 799,	6
1040	
MPI_GRAPH_CREATE, 387, <u>390</u> , 392, 397,	7
400, 404, 411, 412, 1050, 1051	8
MPI_GRAPH_GET, <u>387</u> , <u>400</u>	9
MPI_GRAPH_MAP, 387, <u>411</u> , 412	10
MPI_GRAPH_NEIGHBORS, 387, <u>403</u> , 404,	11
412, 1048	12
MPI_GRAPH_NEIGHBORS_COUNT, 387,	13
$\underline{403}, 404, 1048$	14
MPI_GRAPHDIMS_GET, 387, <u>399</u> , 400	
MPI_GREQUEST_COMPLETE, 626–628, <u>629</u>	15
MPI_GREQUEST_START, <u>626</u> , 868, 870,	16
1049	17
MPI_GROUP_C2F, <u>834</u>	18
MPI_GROUP_COMPARE, <u>314</u> , 317	19
MPI_GROUP_DIFFERENCE, <u>316</u>	20
MPI_GROUP_EXCL, <u>317</u> , 318, <u>319</u>	21
MPI_GROUP_F2C, <u>834</u>	22
MPI_GROUP_FREE, <u>320</u> , 321, 322, 456, 491,	
498,1051	23
MPI_GROUP_FROM_SESSION_PSET, <u>320</u>	24
MPI_GROUP_INCL, <u>317</u> , 319	25
MPI_GROUP_INTERSECTION, <u>315</u>	26
MPI_GROUP_RANGE_EXCL, <u>319</u>	27
MPI_GROUP_RANGE_INCL, <u>318</u> , 319 MPI_GROUP_RANK, <u>312</u> , 322	28
MPI_GROUP_RANK, <u>312</u> , 322 MPI_GROUP_SIZE, <u>312</u> , 322	29
MPI_GROUP_SIZE, <u>312</u> , 322 MPI_GROUP_TRANSLATE_RANKS, <u>313</u> ,	30
$\frac{1050}{1000}$	31
MPI_GROUP_UNION, <u>315</u>	32
MPI_GROUP_UNION, <u>515</u> MPI_IALLGATHER, 185, 189, 190, <u>258</u>	
MPI_Iallgather_c, 258	33
MPI_IALLGATHERV, 185, 189, 190, <u>260</u>	34
MPI_Iallgatherv_c, 260	35
MPI_IALLREDUCE, 185, 189, 190, <u>268</u>	36
MPI_Iallreduce_c, 268	37
MPI_IALLTOALL, 185, 189, 190, <u>261</u>	38
MPI_Ialltoall_c, 261	39
MPI_IALLTOALLV, 185, 189, 190, <u>263</u>	40
MPI_Ialltoallv_c, 263	41
MPI_IALLTOALLW, 185, 189, 190, <u>265</u>	42
MPI_Ialltoallw_c, 265	
MPI_IBARRIER, 185, 189, 248, <u>250</u> , 303	43
MPI_IBCAST, 14, 185, 189, <u>251</u> , 306	44
MPI_Ibcast_c, 251	45
MPI_IBSEND, <u>62</u> , 69, 97	46
MPI_Ibsend_c, 62	47
MPI_IEXSCAN, 186, 189, <u>273</u>	48

1	MPI_Iexscan_c, 273
2	MPI_IGATHER, 185, 189, <u>252</u>
3	MPI_Igather_c, 252
4	MPI_IGATHERV, 185, 189, <u>254</u>
5	MPI_Igatherv_c, 254
	MPI_IMPROBE, 14, 82, <u>85</u> , 86, 87, 90, 515,
6	1039, 1043
7	MPI_IMRECV, 85–87, <u>89</u> , 90, 1043
8	MPI_Imrecv_c, 89
9	MPI_INEIGHBOR_ALLGATHER, 388, <u>425</u> ,
10	1044
11	MPI_Ineighbor_allgather_c, 425
12	MPI_INEIGHBOR_ALLGATHERV, $388, \underline{426}$ ,
13	1044
14	$MPI_Ineighbor_allgatherv_c, 427$
15	MPI_INEIGHBOR_ALLTOALL, 388, <u>428</u> ,
	1044
16	MPI_Ineighbor_alltoall_c, 428
17	MPI_INEIGHBOR_ALLTOALLV, 388, <u>429</u> ,
18	1044
19	MPI_Ineighbor_alltoallv_c, 429
20	MPI_INEIGHBOR_ALLTOALLW, 388, <u>431</u> , 1044
21	MPI_Ineighbor_alltoallw_c, 431
22	MPI_INFO_C2F, 509, <u>834</u>
23	MPI_INFO_CREATE, <u>476</u> , 509
24	MPI_INFO_CREATE_ENV, <u>481</u> , 509, 1038
25	MPI_INFO_DELETE, 467, <u>477</u> , 480, 509
26	MPI_INFO_DUP, <u>481</u> , 509
27	MPI_INFO_F2C, 509, 834
	MPI_INFO_FREE, 343, 481, 482, 491, 498,
28	502, 509, 563, 646, 751, 755, 757, 760
29	MPI_INFO_GET, 23, 475, 477, 509, 776, 1037,
30	1052
31	MPI_INFO_GET_NKEYS, 475, <u>480</u> , 509, 1052
32	MPI_INFO_GET_NTHKEY, 475, <u>480</u> , 509,
33	1052
34	MPI_INFO_GET_STRING, 23, 475, <u>479</u> , 776,
35	777, 1037
36	MPI_INFO_GET_VALUELEN, 23, 475, <u>478</u> ,
37	509, <u>777</u> , 1037, 1052
38	MPI_INFO_SET, <u>476</u> , 477–480, 509
	MPI_INIT, 18, 25, 26, 311, 449, 455, 482, 483,
39	484, 485, 487–490, 493–495, 508, 514,
40	519, 520, 522, 539, 540, 723, 726, 734,
41	747, 833, 836, 837, 1038, 1043, 1044, 1047, 1048
42	MPI_INIT_THREAD, 25, 311, 455, 482–485,
43	$\underbrace{487, 488-490, 493-495, 497, 514, 540,}_{487, 488-490, 493-495, 497, 514, 540,}$
44	$\frac{407}{723}, 726, 734, 833, 1043, 1044, 1048$
	,=0,,=0,,00,000,1010,1011,1010

4, 540, 723, 726, 734, 833, 1043, 1044, 1048 45MPI_INITIALIZED, 493, 494, 509, 833, 1040 46MPI_INTERCOMM_CREATE, 324, 330, 331, 47354, 355, 357, 358, 1044

MPI_INTERCOMM_CREATE_FROM_GROUPS, 48

323, 324, 354, <u>356</u>, 449 MPI_INTERCOMM_MERGE, 323, 330, 351, 354, 355, 357, 358, 1046 MPI_IPROBE, 14, 38, 82, 83, 85-87, 90, 514, 1043 MPI_IRECV, 14, 65, 89, 813, 814, 817, 818 MPI_Irecv_c, 65 MPI_IREDUCE, 185, 189, 190, 267, 268 MPI_Ireduce_c, 267 MPI_IREDUCE_SCATTER, 185, 189, 190, 271MPI_IREDUCE_SCATTER_BLOCK, 185, 189, 190, 269 MPI_Ireduce_scatter_block_c, 270 MPI_Ireduce_scatter_c, 271 MPI_IRSEND, 64 MPI_Irsend_c, 64 MPI_IS_THREAD_MAIN, 487, 490, 1040 MPI_ISCAN, 186, 189, 272 MPI_Iscan_c, 272 MPI_ISCATTER, 185, 189, 255 MPI_Iscatter_c, 256 MPI_ISCATTERV, 185, 189, 257 MPI_Iscatterv_c, 257 MPI_ISEND, 14, <u>61</u>, 97, 791, 792, 795, 812, 813, 818, 819 MPI_Isend_c, 61 MPI_ISENDRECV, <u>66</u>, <u>1036</u> MPI_Isendrecv_c, 66 MPI_ISENDRECV_REPLACE, 67, 1036 MPI_Isendrecv_replace_c, 67 MPI_ISSEND, 63 MPI_Issend_c, 63 MPI_KEYVAL_CREATE, 23, 773, 775, 871 MPI_KEYVAL_FREE, 23, 375, 774 MPI_LOOKUP_NAME, 467, 529, 535, 536 MPI_MESSAGE_C2F, 834, 1043 MPI_MESSAGE_F2C, 834, 1043 MPI_MPROBE, 14, 82, 85, 86, 87, 90, 515, 1043MPI_MRECV, 14, 85-87, 88, 89, 1043 MPI_Mrecv_c, 88 MPI_NEIGHBOR_ALLGATHER, 388, 413, 415, 417, 426, 1035, 1044 MPI_Neighbor_allgather_c, 413 MPI_NEIGHBOR_ALLGATHER_INIT, 433 MPI_Neighbor_allgather_init_c, 433 MPI_NEIGHBOR_ALLGATHERV, 388, 416, 427, 1035, 1044 MPI_Neighbor_allgatherv_c, 416 MPI_NEIGHBOR_ALLGATHERV_INIT, 434, 791, 1038 MPI_Neighbor_allgatherv_init_c, 434

MPI_NEIGHBOR_ALLTOALL, 388, 418, 419, 420, 429, 1035, 1044 MPI_Neighbor_alltoall_c, 418 MPI_NEIGHBOR_ALLTOALL_INIT, 436 MPI_Neighbor_alltoall_init_c, 436 MPI_NEIGHBOR_ALLTOALLV, 388, 421, 430, 1035, 1044 MPI_Neighbor_alltoallv_c, 421 MPI_NEIGHBOR_ALLTOALLV_INIT,  $\underline{437}$ , 791, 1038 MPI_Neighbor_alltoallv_init_c, 437 MPI_NEIGHBOR_ALLTOALLW, 388, 423, 432, 1035, 1044 MPI_Neighbor_alltoallw_c, 423 MPI_NEIGHBOR_ALLTOALLW_INIT, 439, 791, 1038 MPI_Neighbor_alltoallw_init_c, 439 MPI_NULL_COPY_FN, 23, 364, 774, 858 MPI_NULL_DELETE_FN, 23, 365, 774, 858 MPI_OP_C2F, 834 MPI_OP_COMMUTATIVE, 239, 1048 MPI_OP_CREATE, 231, 232, 234, 791, 866, 869, 1045, 1046 MPI_Op_create_c, 231, 832 MPI_OP_F2C, 834 MPI_OP_FREE, 234, 491, 498 MPI_OPEN_PORT, 529, 530, 532-537 MPI_PACK, 59, <u>173</u>, 175–177, 180, 693, 699 MPI_Pack_c, 173 MPI_PACK_EXTERNAL, 8, <u>180</u>, 806, 1050 MPI_Pack_external_c, 180 MPI_PACK_EXTERNAL_SIZE, 182 MPI_Pack_external_size_c, 182 MPI_PACK_SIZE, 59, <u>176</u>, 1043 MPI_Pack_size_c, 176 MPI_PARRIVED, 102, 108, 109 MPI_PCONTROL, 718, 719, 720 MPI_PREADY, 102, 104, 106, 107-109 MPI_PREADY_LIST, 108 MPI_PREADY_RANGE, 107, 108 MPI_PRECV_INIT, 102, 105, 106, 110 MPI_PROBE, 14, 36, 38, 39, 82, 83, 84, 85, 87, 89, 90, 514, 1043 MPI_PSEND_INIT, 102, 104, 105–107, 110 MPI_PUBLISH_NAME, 529, 530, 534, 535, 536 MPI_PUT, 545, 563, 564, 568, 572, 580, 587, 593, 603, 605, 610, 611, 620, 812, 823, 1052MPI_Put_c, 565 MPI_QUERY_THREAD, 489, 490, 514, 1040 MPI_RACCUMULATE, 545, 563, 572, 576, 582, 583 MPI_Raccumulate_c, 582

MPI_RECV, 14, 30, <u>35</u> , 37–39, 82, 85, 86, 89,	1
118, 151, 152, 175, 186, 195, 304, 634,	2
708, 747, 819, 822, 823	3
MPI_Recv_c, 35	4
MPI_RECV_INIT, 14, <u>96</u> , 97, 106	5
MPI_Recv_init_c, 96	
MPI_REDUCE, 185, 189, 190, <u>222</u> , 223, 224,	6
232-234, 237, 241, 243, 244, 246, 268,	7
572, 575-577, 1049	8
MPI_Reduce_c, 222	9
MPI_REDUCE_INIT, 292	10
MPI_Reduce_init_c, 292	11
MPI_REDUCE_LOCAL, 223, 224, 232, <u>238</u> ,	
1046, 1048	12
MPI_Reduce_local_c, 238	13
MPI_REDUCE_SCATTER, 185, 189, 190,	14
224, 232, 241, <u>242</u> , 243, 272	15
MPI_REDUCE_SCATTER_BLOCK, 185,	16
189, 190, 224, 232, <u>240</u> , 241, 270, 1048	17
MPI_Reduce_scatter_block_c, 240	
MPI_REDUCE_SCATTER_BLOCK_INIT,	18
294	19
	20
MPI_Reduce_scatter_block_init_c, 294	21
MPI_Reduce_scatter_c, 242	22
MPI_REDUCE_SCATTER_INIT, <u>296</u>	23
MPI_Reduce_scatter_init_c, 296	24
MPI_REGISTER_DATAREP, 467, <u>696</u> ,	
697–700, 713, 868, 871	25
MPI_Register_datarep_c, 696, 832	26
MPI_REQUEST_C2F, <u>834</u>	27
MPI_REQUEST_F2C, <u>834</u>	28
MPI_REQUEST_FREE, <u>71</u> , 72, 90, 98, 249,	29
274, 275, 491, 498, 578, 628, 629, 1047	30
$MPI_REQUEST_GET_STATUS, 39, \underline{81}, 627,$	
1047	31
MPI_RGET, 545, 563, <u>580</u> , 581	32
MPI_RGET_ACCUMULATE, 545, 563, 572,	33
$575, \underline{584}, 585$	34
MPI_Rget_accumulate_c, 584	35
MPI_Rget_c, 580	36
MPI_RPUT, 545, 563, <u>579</u> , 580	37
MPI_Rput_c, 579	
MPI_RSEND, 14, $\underline{51}$	38
MPI_Rsend_c, 51	39
MPI_RSEND_INIT, $95$	40
MPI_Rsend_init_c, 95	41
MPI_SCAN, 186, 189, 224, 232, <u>243</u> , 244, 246,	42
273	43
MPI_Scan_c, 244	
MPI_SCAN_INIT, <u>297</u>	44
MPI_Scan_init_c, 297	45
MPI_SCATTER, 185, 189, <u>204</u> , 205, 207, 208,	46
241, 256	47
MPI_Scatter_c, 204	48

1	MPI_SCATTER_INIT, <u>281</u>	Μ
2	MPI_Scatter_init_c, 281	
3	MPI_SCATTERV, 185, 189, <u>206</u> , 207, 208,	Μ
4	243, 258	Μ
5	MPI_Scatterv_c, 206 MPI_SCATTERV_INIT, 282	М
6	MPI_Scatterv_init_c, 283	111
7	MPI_SEND, 14, 29, <u>30</u> , 31, 39, 44, 118, 151,	Μ
8	173, 304, 638, 708, 720, 819, 820, 822,	
9	823	Μ
10	MPI_Send_c, 30	
11	MPI_SEND_INIT, 14, <u>92</u> , 97, 105	Μ
12	MPI_Send_init_c, 92	М
13	MPI_SENDRECV, <u>40</u> , 407 MPI_Sendrecv_c, <u>41</u>	M
14	MPI_SENDRECV_REPLACE, <u>42</u>	111
15	MPI_Sendrecv_replace_c, 42	Μ
16	$MPI_SESSION_C2F, \underline{834}$	
17	$MPI_SESSION_CALL_ERRHANDLER, \underline{473},$	Μ
18	509	٦.1
19	MPI_SESSION_CREATE_ERRHANDLER, 456, 463, 464, 497, 509, 867, 870	Μ
20	MPI_SESSION_CREATE_KEYVAL, 869	Μ
21	MPI_SESSION_F2C, <u>834</u>	M
22	MPI_SESSION_FINALIZE, 498, 508	
23	MPI_SESSION_GET_ERRHANDLER, 456,	Μ
24	$\underline{464}$	Μ
25	MPI_SESSION_GET_INFO, 497, 501, 502	M
26	MPI_SESSION_GET_NTH_PSET, 320, <u>500</u> , 501	M
27	MPI_SESSION_GET_NUM_PSETS, 499, 500	M M
28	MPI_SESSION_GET_PSET_INFO, <u>502</u>	111
29	MPI_SESSION_INIT, 456, 491, 497, 508, 514,	М
30	540, 727	
31	MPI_SESSION_SET_ERRHANDLER, 456,	Μ
32	<u>464</u>	M
33	MPI_SIZEOF, 23, <u>778</u> , 1038 MPI_SSEND, <u>50</u>	Μ
34	MPI_SSEND, <u>50</u> MPI_Ssend_c, <u>50</u>	Μ
35	MPI_SSEND_INIT, <u>94</u>	M
36	MPI_Ssend_init_c, 94	Μ
37	MPI_START, <u>97</u> , 98, 101, 104, 107, 108, 274,	
38	275, 819	Μ
39	MPI_STARTALL, <u>97</u> , 98, 101, 104, 107, 108,	
40	274, 275, 819	M
41	MPI_STATUS_C2F, <u>836</u> MPI_STATUS_C2F08, <u>836</u> , 1045	M M
42	MPI_STATUS_F082C, <u>836</u> , 1045	M
43	MPI_STATUS_F082F, <u>838</u> , 1035, 1045	M
44	$MPI_STATUS_F2C, \underline{835}$	
45	MPI_STATUS_F2F08, <u>837</u> , 1035, 1045	Μ
46	MPI_STATUS_SET_CANCELLED, <u>633</u> , 1038	_
47	MPI_STATUS_SET_ELEMENTS, <u>632</u> , 633	M
48		Μ

MPI_STATUS_SET_ELEMENTS_X, <u>633</u> ,
1042
MPI_T_CATEGORY_CHANGED, <u>769</u>
MPI_T_CATEGORY_GET_CATEGORIES,
$\underline{768}, 769, 771$
MPI_T_CATEGORY_GET_CVARS, <u>767</u> , 769,
771
MPI_T_CATEGORY_GET_EVENTS, 768,
769
MPI_T_CATEGORY_GET_INDEX, 767, 771,
1041
MPI_T_CATEGORY_GET_INFO, <u>766</u> , 769,
771,1040,1041
MPI_T_CATEGORY_GET_NUM, 765
MPI_T_CATEGORY_GET_NUM_EVENTS,
766, 767
MPI_T_CATEGORY_GET_PVARS, 768, 769,
771
MPI_T_CVAR_GET_INDEX, <u>732</u> , 733, 771,
1041
MPI_T_CVAR_GET_INFO, 728, 730, 731,
732, 734–736, 769, 771, 1040, 1041
MPI_T_CVAR_GET_NUM, <u>730</u> , 735
MPI_T_CVAR_HANDLE_ALLOC, 728, 734,
735, 736, 771
MPI_T_CVAR_HANDLE_FREE, 735, 771
MPI_T_CVAR_READ, <u>735</u> , 771
MPI_T_CVAR_WRITE, <u>735</u> , 771
MPI_T_ENUM_GET_INFO, 728, 729, 771
MPI_T_ENUM_GET_ITEM, <u>729</u> , 771
MPI_T_EVENT_CALLBACK_GET_INFO,
759
MPI_T_EVENT_CALLBACK_SET_INFO,
759
MPI_T_EVENT_COPY, 753, 755, <u>763</u>
MPI_T_EVENT_GET_INDEX, <u>756</u> , 771
MPI_T_EVENT_GET_INFO, 728, 754, 756,
763, 769, 771
MPI_T_EVENT_GET_NUM, <u>754</u>
$MPI_T_EVENT_GET_SOURCE, 753, 764$
MPI_T_EVENT_GET_TIMESTAMP, 751,
753, <u>763</u>
MPI_T_EVENT_HANDLE_ALLOC, <u>756</u> , 759,
771
MPI_T_EVENT_HANDLE_FREE, <u>760</u> , 771
MPI_T_EVENT_HANDLE_GET_INFO, 757
MPI_T_EVENT_HANDLE_GET_INFO, <u>157</u> MPI_T_EVENT_HANDLE_SET_INFO, <u>757</u>
MPI_T_EVENT_HANDLE_SET_HAPO, <u>737</u> MPI_T_EVENT_READ, 753, 755, <u>762</u>
MPI_I_EVENI_KEAD, 735, 755, <u>762</u> MPI_T_EVENT_REGISTER_CALLBACK,
758 MPI_T_EVENT_SET_DROPPED_HANDLER,
$\frac{761}{762}$
MPI_T_FINALIZE, <u>727</u>
$\mathbf{M}\mathbf{I} \mathbf{I} \mathbf{I} \mathbf{I} \mathbf{\Gamma} \mathbf{I} \mathbf{N} \mathbf{A} \mathbf{L} \mathbf{I} \mathbf{Z} \mathbf{E}, \ \underline{I} \underline{Z} \mathbf{I}$

 $\mathrm{MPI_T_INIT_THREAD},\,\underline{726},\,727$ 

MPI_T_PVAR_GET_INDEX, <u>741</u>, 771, 1041 MPI_T_PVAR_GET_INFO, 728, 739, 740, 743, 745, 747, 769, 771, 1040, 1041 MPI_T_PVAR_GET_NUM, 739, 743 MPI_T_PVAR_HANDLE_ALLOC, 728, 743, 745, 771 MPI_T_PVAR_HANDLE_FREE, 743, 744, 771, 1040 MPI_T_PVAR_READ, 745, 746, 753, 771, 1040 MPI_T_PVAR_READRESET, 741, 746, 753, 771, 1040 MPI_T_PVAR_RESET, 746, 753, 771, 1040 MPI_T_PVAR_SESSION_CREATE, 742, 771 MPI_T_PVAR_SESSION_FREE, 742, 771 MPI_T_PVAR_START, 744, 753, 771, 1040 MPI_T_PVAR_STOP, 744, 753, 771, 1040 MPI_T_PVAR_WRITE, 745, 746, 753, 771, 1040 MPI_T_SOURCE_GET_INFO, 750, 751, 764, 771 MPI_T_SOURCE_GET_NUM, 750, 771 MPI_T_SOURCE_GET_TIMESTAMP, 751, 753, 771 MPI_TEST, 39, 68, 69, <u>70</u>, 72, 73, 75, 81, 90, 98, 101, 102, 108, 109, 275, 491, 629, 652.653 MPI_TEST_CANCELLED, 69, 70, 91, 627, 634, 654 MPI_TESTALL, 73, 77, 514, 627-629, 632 MPI_TESTANY, 73, 75, 79, 514, 627, 628, 632 MPI_TESTSOME, 74, 79, 80, 514, 627-629, 632 MPI_TOPO_TEST, 387, 399 MPI_TYPE_C2F, 834 MPI_TYPE_COMMIT, <u>149</u>, 835 MPI_TYPE_CONTIGUOUS, 15, 119, 122, 143, 157, 636, 692 MPI_Type_contiguous_c, 119 MPI_TYPE_CREATE_DARRAY, 15, 39, 134, 135, 157 MPI_Type_create_darray_c, 134 MPI_TYPE_CREATE_F90_COMPLEX, 15, 157, 160, 225, 696, 784, 803, 805, 806 MPI_TYPE_CREATE_F90_INTEGER, 15, 157, 160, 225, 696, 784, 804, 805, 806 MPI_TYPE_CREATE_F90_REAL, 15, 157, 160, 225, 696, 784, 803, 804-806, 1047 MPI_TYPE_CREATE_HINDEXED, 15, 23, 119, 125, 126, 129, 131, 157, 779, 831 MPI_TYPE_CREATE_HINDEXED_BLOCK, 15, 119, <u>128</u>, 157, 831, 1043 MPI_Type_create_hindexed_block_c, 128 MPI_Type_create_hindexed_c, 126

MPI_TYPE_CREATE_HVECTOR, 15, 23,	T
$119, \underline{122}, 157, 779$	2
MPI_Type_create_hvector_c, 122	3
MPI_TYPE_CREATE_INDEXED_BLOCK,	
15, 127, 128, 157	4
MPI_Type_create_indexed_block_c, 127	5
MPI_TYPE_CREATE_KEYVAL, 361, <u>372</u> ,	6
	7
375, 841, 867, 870, 1050	8
MPI_TYPE_CREATE_RESIZED, 23, 119,	
$143, \underline{146}, 147, 157, 692, 780, 1045$	9
MPI_Type_create_resized_c, 146	10
MPI_TYPE_CREATE_STRUCT, 15, 23, 119,	11
$\underline{129}, 130, 131, 143, 157, 220, 779, 831$	12
MPI_Type_create_struct_c, 129	13
MPI_TYPE_CREATE_SUBARRAY, 15, 18,	
$\underline{131}, 133, 136, 157$	14
MPI_Type_create_subarray_c, 131	15
MPI_TYPE_DELETE_ATTR, 361, <u>375</u> , 1045	16
MPI_TYPE_DUP, 15, <u>150</u> , 157, 1045	17
MPI_TYPE_DUP_FN, <u>372</u> , 857, 1040	18
MPI_TYPE_EXTENT, 23, <u>779</u> , 1042	
$MPI_TYPE_F2C, \underline{834}$	19
MPI_TYPE_FREE, <u>149</u> , 159, 373, 491, 498	20
MPI_TYPE_FREE_KEYVAL, 361, <u>373</u> , 375	21
MPI_TYPE_GET_ATTR, 361, <u>374</u> , 375, 792,	22
841, 1045	23
MPI_TYPE_GET_CONTENTS, 156, 157,	24
<u>158</u> , 159–161, 831	25
MPI_Type_get_contents_c, 158	26
MPI_TYPE_GET_ENVELOPE, <u>156</u> , 157,	27
159, 160, 805, 831	28
MPI_Type_get_envelope_c, 156	29
MPI_TYPE_GET_EXTENT, 23, <u>145</u> , 148,	30
779, 807, 838	
$MPI_Type_get_extent_c, 145$	31
$MPI_TYPE_GET_EXTENT_X, \underline{145}, 1042$	32
MPI_TYPE_GET_NAME, <u>381</u> , 1046	33
MPI_TYPE_GET_EXTENT_X, <u>145</u> , 1042 MPI_TYPE_GET_NAME, <u>381</u> , 1046 MPI_TYPE_GET_TRUE_EXTENT, <u>147</u>	34
MPI_Type_get_true_extent_c, 147	35
MPI_TYPE_GET_TRUE_EXTENT_X, 147,	36
$\underline{148}, 1042$	
MPI_TYPE_HINDEXED, 23, <u>779</u> , 1042	37
MPI_TYPE_HVECTOR, 23, <u>779</u> , 1042	38
MPI_TYPE_INDEXED, 15, <u>124</u> , 125, 127, 157	39
MPI_Type_indexed_c, 124	40
MPI_TYPE_LB, 23, 779, 1042	41
MPI_TYPE_MATCH_SIZE, 784, <u>807</u> , 1046	42
MPI_TYPE_NULL_COPY_FN, <u>372</u> , 857,	
1040	43
MPI_TYPE_NULL_DELETE_FN, <u>372</u> , 857,	44
1040, 1046	45
MPI_TYPE_SET_ATTR, 361, <u>374</u> , 375, 792,	46
841, 845, 1045	47
MPI_TYPE_SET_NAME, <u>380</u> , 1045	48
	10

ODDATE INTEGROD 15 00

1	MPI_TYPE_SIZE, <u>141</u> , 142, 720, 1043
2	MPI_Type_size_c, 141
3	MPI_TYPE_SIZE_X, <u>142</u> , 1042
	MPI_TYPE_STRUCT, 23, <u>779</u> , 1042
4	MPI_TYPE_UB, 23, <u>779</u> , 1042
5	
6	MPI_TYPE_VECTOR, 15, 120, <u>121</u> , 122, 125,
7	157 NDL T
8	MPI_Type_vector_c, 121
	MPI_UNPACK, <u>174</u> , 175, 180, 699
9	MPI_Unpack_c, 174
10	MPI_UNPACK_EXTERNAL, 8, <u>181</u> , 806
11	MPI_Unpack_external_c, 181
12	MPI_UNPUBLISH_NAME, 468, <u>535</u> , 536
13	MPI_USER_FUNCTION, 232, 831
14	MPI_User_function_c, 232, 864
	MPI_WAIT, 37, 39, 68, <u>69</u> , 70–74, 76, 90, 98,
15	101, 102, 108, 109, 248, 275, 304, 491,
16	514, 625, 629, 652, 653, 680, 701-703,
17	812, 818, 819, 822
18	MPI_WAITALL, 73, <u>76</u> , 77, 248, 305, 514, 578,
19	627-629, 632
	MPI_WAITANY, 52, 73, <u>74</u> , 80, 514, 627, 628,
20	632
21	MPI_WAITSOME, 74, <u>78</u> , 79, 80, 514,
22	627-629, 632
23	MPI_WIN_ALLOCATE, 546, <u>550</u> , 551, 553,
24	559, 561, 566, 599, 797, 798, 1037,
25	1039
26	MPI_Win_allocate_c, 550
26 27	MPI_Win_allocate_c, 550 MPI_WIN_ALLOCATE_CPTR, 551, 1039
26	MPI_Win_allocate_c, 550 MPI_WIN_ALLOCATE_CPTR, 551, 1039 MPI_WIN_ALLOCATE_SHARED, 335, 546,
26 27	MPI_Win_allocate_c, 550 MPI_WIN_ALLOCATE_CPTR, 551, 1039 MPI_WIN_ALLOCATE_SHARED, 335, 546, 552, 553, 555, 559, 561, 599, 798,
26 27 28	$\begin{array}{l} {\rm MPI_Win_allocate_c,\ 550} \\ {\rm MPI_WIN_ALLOCATE_CPTR,\ 551,\ 1039} \\ {\rm MPI_WIN_ALLOCATE_SHARED,\ 335,\ 546,} \\ \underline{552},\ 553,\ 555,\ 559,\ 561,\ 599,\ 798,} \\ \underline{1037-1039} \end{array}$
26 27 28 29	$\begin{array}{l} {\rm MPI_Win_allocate_c,\ 550} \\ {\rm MPI_WIN_ALLOCATE_CPTR,\ 551,\ 1039} \\ {\rm MPI_WIN_ALLOCATE_SHARED,\ 335,\ 546,} \\ \underline{552},\ 553,\ 555,\ 559,\ 561,\ 599,\ 798,} \\ \underline{1037-1039} \\ {\rm MPI_Win_allocate_shared_c,\ 552} \end{array}$
26 27 28 29 30 31	$\begin{array}{l} {\rm MPI_Win_allocate_c,\ 550} \\ {\rm MPI_WIN_ALLOCATE_CPTR,\ 551,\ 1039} \\ {\rm MPI_WIN_ALLOCATE_SHARED,\ 335,\ 546, \\ \underline{552},\ 553,\ 555,\ 559,\ 561,\ 599,\ 798, \\ 1037-1039} \\ {\rm MPI_Win_allocate_shared_c,\ 552} \\ {\rm MPI_WIN_ALLOCATE_SHARED_CPTR, } \end{array}$
26 27 28 29 30 31 32	$\begin{array}{c} {\rm MPI_Win_allocate_c,\ 550} \\ {\rm MPI_WIN_ALLOCATE_CPTR,\ 551,\ 1039} \\ {\rm MPI_WIN_ALLOCATE_SHARED,\ 335,\ 546, \\ \underline{552},\ 553,\ 555,\ 559,\ 561,\ 599,\ 798, \\ 1037-1039} \\ {\rm MPI_Win_allocate_shared_c,\ 552} \\ {\rm MPI_WIN_ALLOCATE_SHARED_CPTR, \\ \ 553,\ 1039} \end{array}$
26 27 28 29 30 31 32 33	$\begin{array}{l} {\rm MPI_Win_allocate_c,\ 550} \\ {\rm MPI_WIN_ALLOCATE_CPTR,\ 551,\ 1039} \\ {\rm MPI_WIN_ALLOCATE_SHARED,\ 335,\ 546, \\ \underline{552},\ 553,\ 555,\ 559,\ 561,\ 599,\ 798, \\ 1037-1039} \\ {\rm MPI_Win_allocate_shared_c,\ 552} \\ {\rm MPI_WIN_ALLOCATE_SHARED_CPTR, \\ 553,\ 1039} \\ {\rm MPI_WIN_ATTACH,\ 556,\ \underline{557},\ 558,\ 559,\ 599} \end{array}$
26 27 28 29 30 31 32	$\begin{array}{l} {\rm MPI_Win_allocate_c,\ 550} \\ {\rm MPI_WIN_ALLOCATE_CPTR,\ 551,\ 1039} \\ {\rm MPI_WIN_ALLOCATE_SHARED,\ 335,\ 546, \\ \underline{552},\ 553,\ 555,\ 559,\ 561,\ 599,\ 798, \\ 1037-1039} \\ {\rm MPI_Win_allocate_shared_c,\ 552} \\ {\rm MPI_WIN_ALLOCATE_SHARED_CPTR, \\ 553,\ 1039} \\ {\rm MPI_WIN_ATTACH,\ 556,\ \underline{557},\ 558,\ 559,\ 599} \\ {\rm MPI_WIN_C2F,\ \underline{834}} \end{array}$
26 27 28 29 30 31 32 33	$\begin{array}{l} {\rm MPI_Win_allocate_c, 550} \\ {\rm MPI_WIN_ALLOCATE_CPTR, 551, 1039} \\ {\rm MPI_WIN_ALLOCATE_SHARED, 335, 546, \\ \underline{552}, 553, 555, 559, 561, 599, 798, \\ 1037-1039 \\ {\rm MPI_Win_allocate_shared_c, 552} \\ {\rm MPI_WIn_ALLOCATE_SHARED_CPTR, \\ 553, 1039 \\ {\rm MPI_WIN_ATTACH, 556, \underline{557}, 558, 559, 599} \\ {\rm MPI_WIN_C2F, \underline{834} \\ {\rm MPI_WIN_CALL_ERRHANDLER, \underline{472}, 473 \\ \end{array}$
26 27 28 29 30 31 32 33 34	<ul> <li>MPI_Win_allocate_c, 550</li> <li>MPI_WIN_ALLOCATE_CPTR, 551, 1039</li> <li>MPI_WIN_ALLOCATE_SHARED, 335, 546, 552, 553, 555, 559, 561, 599, 798, 1037–1039</li> <li>MPI_Win_allocate_shared_c, 552</li> <li>MPI_WIN_ALLOCATE_SHARED_CPTR, 553, 1039</li> <li>MPI_WIN_ATTACH, 556, 557, 558, 559, 599</li> <li>MPI_WIN_C2F, <u>834</u></li> <li>MPI_WIN_CALL_ERRHANDLER, <u>472</u>, 473</li> <li>MPI_WIN_COMPLETE, 559, 588, <u>592</u>,</li> </ul>
26 27 28 29 30 31 32 33 34 35	$\begin{array}{l} \text{MPI}_\text{Win}_\text{allocate}_\text{c}, 550\\ \text{MPI}_\text{WIN}_\text{ALLOCATE}_\text{CPTR}, 551, 1039\\ \text{MPI}_\text{WIN}_\text{ALLOCATE}_\text{SHARED}, 335, 546, \\ \underline{552}, 553, 555, 559, 561, 599, 798, \\ 1037-1039\\ \text{MPI}_\text{Win}_\text{allocate}_\text{shared}_\text{c}, 552\\ \text{MPI}_\text{Win}_\text{ALLOCATE}_\text{SHARED}_\text{CPTR}, \\ \underline{553}, 1039\\ \text{MPI}_\text{WIN}_\text{ATTACH}, 556, \underline{557}, 558, 559, 599\\ \text{MPI}_\text{WIN}_\text{C2F}, \underline{834}\\ \text{MPI}_\text{WIN}_\text{CALL}_\text{ERRHANDLER}, \underline{472}, 473\\ \text{MPI}_\text{WIN}_\text{COMPLETE}, 559, 588, \underline{592}, \\ \underline{593}-596, 604, 611\\ \end{array}$
26 27 28 29 30 31 32 33 34 35 36 37	$\begin{array}{l} {\rm MPI_Win_allocate_c, 550} \\ {\rm MPI_WIN_ALLOCATE_CPTR, 551, 1039} \\ {\rm MPI_WIN_ALLOCATE_SHARED, 335, 546}, \\ \underline{552}, 553, 555, 559, 561, 599, 798, \\ 1037-1039 \\ {\rm MPI_Win_allocate_shared_c, 552} \\ {\rm MPI_WIn_ALLOCATE_SHARED_CPTR}, \\ \underline{553}, 1039 \\ {\rm MPI_WIN_ATTACH, 556, \underline{557}, 558, 559, 599} \\ {\rm MPI_WIN_C2F, \underline{834}} \\ {\rm MPI_WIN_CALL_ERRHANDLER, \underline{472}, 473} \\ {\rm MPI_WIN_COMPLETE, 559, 588, \underline{592}, \\ \underline{593-596, 604, 611} \\ \\ {\rm MPI_WIN_CREATE, 515, 546, \underline{547}, 549, 551, } \end{array}$
26 27 28 29 30 31 32 33 34 35 36 37 38	$\begin{array}{l} \text{MPI_Win_allocate_c, 550} \\ \text{MPI_WIN_ALLOCATE_CPTR, 551, 1039} \\ \text{MPI_WIN_ALLOCATE_SHARED, 335, 546,} \\ \underline{552}, 553, 555, 559, 561, 599, 798, \\ 1037-1039 \\ \text{MPI_Win_allocate_shared_c, 552} \\ \text{MPI_WIN_ALLOCATE_SHARED_CPTR,} \\ \underline{553}, 1039 \\ \text{MPI_WIN_ATTACH, 556, } \underline{557}, 558, 559, 599 \\ \text{MPI_WIN_ATTACH, 556, } \underline{557}, 558, 559, 599 \\ \text{MPI_WIN_C2F, } \underline{834} \\ \text{MPI_WIN_CALL_ERRHANDLER, } \underline{472}, 473 \\ \text{MPI_WIN_COMPLETE, } 559, 588, \underline{592}, \\ \underline{593}-596, 604, 611 \\ \text{MPI_WIN_CREATE, } 515, 546, \underline{547}, 549, 551, \\ \underline{553}, 557-559, 561, 603 \\ \end{array}$
26 27 28 29 30 31 32 33 34 35 36 35 36 37 38 39	$\begin{array}{l} \text{MPI_Win_allocate_c, 550} \\ \text{MPI_WIN_ALLOCATE_CPTR, 551, 1039} \\ \text{MPI_WIN_ALLOCATE_SHARED, 335, 546,} \\ \underline{552}, 553, 555, 559, 561, 599, 798, \\ 1037-1039 \\ \text{MPI_Win_allocate_shared_c, 552} \\ \text{MPI_WIN_ALLOCATE_SHARED_CPTR,} \\ \underline{553}, 1039 \\ \text{MPI_WIN_ATTACH, 556, } \underline{557}, 558, 559, 599 \\ \text{MPI_WIN_C2F, } \underline{834} \\ \text{MPI_WIN_CALL_ERRHANDLER, } \underline{472}, 473 \\ \text{MPI_WIN_COMPLETE, 559, 588, } \underline{592}, \\ \underline{593}-596, 604, 611 \\ \text{MPI_WIN_CREATE, 515, 546, } \underline{547}, 549, 551, \\ \underline{553}, 557-559, 561, 603 \\ \text{MPI_Win_create_c, 547} \\ \end{array}$
26 27 28 29 30 31 32 33 34 35 36 37 38	$\begin{array}{l} {\rm MPI_Win_allocate_c, 550} \\ {\rm MPI_WIN_ALLOCATE_CPTR, 551, 1039} \\ {\rm MPI_WIN_ALLOCATE_SHARED, 335, 546, \\ \underline{552}, 553, 555, 559, 561, 599, 798, \\ 1037-1039 \\ {\rm MPI_Win_allocate_shared_c, 552} \\ {\rm MPI_WIN_ALLOCATE_SHARED_CPTR, \\ 553, 1039 \\ {\rm MPI_WIN_ATTACH, 556, \underline{557}, 558, 559, 599} \\ {\rm MPI_WIN_C2F, \underline{834} \\ {\rm MPI_WIN_CALL_ERRHANDLER, \underline{472}, 473} \\ {\rm MPI_WIN_COMPLETE, 559, 588, \underline{592}, \\ 593-596, 604, 611 \\ {\rm MPI_WIN_CREATE, 515, 546, \underline{547}, 549, 551, \\ 553, 557-559, 561, 603 \\ {\rm MPI_WIn_create_c, 547} \\ {\rm MPI_WIN_CREATE_DYNAMIC, 468, 546, \\ \end{array}$
26 27 28 29 30 31 32 33 34 35 36 35 36 37 38 39	$\begin{array}{l} {\rm MPI_Win_allocate_c, 550} \\ {\rm MPI_WIN_ALLOCATE_CPTR, 551, 1039} \\ {\rm MPI_WIN_ALLOCATE_SHARED, 335, 546, \\ \underline{552}, 553, 555, 559, 561, 599, 798, \\ 1037-1039 \\ {\rm MPI_Win_allocate_shared_c, 552} \\ {\rm MPI_WIN_ALLOCATE_SHARED_CPTR, \\ 553, 1039 \\ {\rm MPI_WIN_ATTACH, 556, \underline{557}, 558, 559, 599} \\ {\rm MPI_WIN_C2F, \underline{834}} \\ {\rm MPI_WIN_CALL_ERRHANDLER, \underline{472}, 473} \\ {\rm MPI_WIN_COMPLETE, 559, 588, \underline{592}, \\ 593-596, 604, 611 \\ {\rm MPI_WIN_CREATE, 515, 546, \underline{547}, 549, 551, \\ 553, 557-559, 561, 603 \\ {\rm MPI_Win_create_c, 547} \\ {\rm MPI_WIN_CREATE_DYNAMIC, 468, 546, \\ \underline{556}, 557, 558, 560, 561, 604 \\ \end{array}$
26 27 28 30 31 32 33 34 35 36 37 38 39 40	$\begin{array}{l} {\rm MPI_Win_allocate_c, 550} \\ {\rm MPI_WIN_ALLOCATE_CPTR, 551, 1039} \\ {\rm MPI_WIN_ALLOCATE_SHARED, 335, 546, \\ \underline{552}, 553, 555, 559, 561, 599, 798, \\ 1037-1039 \\ {\rm MPI_Win_allocate_shared_c, 552} \\ {\rm MPI_WIN_ALLOCATE_SHARED_CPTR, \\ 553, 1039 \\ {\rm MPI_WIN_ATTACH, 556, \underline{557}, 558, 559, 599} \\ {\rm MPI_WIN_C2F, \underline{834}} \\ {\rm MPI_WIN_CALL_ERRHANDLER, \underline{472}, 473} \\ {\rm MPI_WIN_COMPLETE, 559, 588, \underline{592}, \\ 593-596, 604, 611 \\ {\rm MPI_WIN_CREATE, 515, 546, \underline{547}, 549, 551, \\ 553, 557-559, 561, 603 \\ {\rm MPI_Win_create_c, 547} \\ {\rm MPI_Win_create_c, 547} \\ {\rm MPI_WIN_CREATE_DYNAMIC, 468, 546, \\ \underline{556}, 557, 558, 560, 561, 604 \\ {\rm MPI_WIN_CREATE_ERRHANDLER, 456, \\ \end{array}$
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41	$\begin{array}{l} \text{MPI_Win_allocate_c, 550} \\ \text{MPI_WIN_ALLOCATE_CPTR, 551, 1039} \\ \text{MPI_WIN_ALLOCATE_SHARED, 335, 546}, \\ \underline{552}, 553, 555, 559, 561, 599, 798, \\ 1037-1039 \\ \text{MPI_Win_allocate_shared_c, 552} \\ \text{MPI_WIN_ALLOCATE_SHARED_CPTR, \\ 553, 1039 \\ \text{MPI_WIN_ATTACH, 556, } \underline{557}, 558, 559, 599 \\ \text{MPI_WIN_C2F, } \underline{834} \\ \text{MPI_WIN_CALL_ERRHANDLER, } \underline{472}, 473 \\ \text{MPI_WIN_COMPLETE, } 559, 588, \\ \underline{593}-596, 604, 611 \\ \text{MPI_WIN_CREATE, } 515, 546, \\ \underline{547}, 553, 557-559, 561, 603 \\ \text{MPI_WIN_CREATE_DYNAMIC, 468, 546, \\ \\ \underline{556}, 557, 558, 560, 561, 604 \\ \text{MPI_WIN_CREATE_ERRHANDLER, } 456, \\ \\ \underline{459}, 460, 867, 870, 1046 \\ \end{array}$
26 27 28 29 30 31 32 33 34 35 36 35 36 37 38 39 40 41 42	$\begin{array}{llllllllllllllllllllllllllllllllllll$
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43	$\begin{array}{llllllllllllllllllllllllllllllllllll$
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44	$\begin{array}{llllllllllllllllllllllllllllllllllll$
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43	$\begin{array}{llllllllllllllllllllllllllllllllllll$
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44	$\begin{array}{llllllllllllllllllllllllllllllllllll$
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46	$\begin{array}{llllllllllllllllllllllllllllllllllll$

MPI_WIN_FENCE, 559, 568, 588, 590, 591, 592, 601, 602, 604, 605, 608, 613, 823 MPI_WIN_FLUSH, 554, 578, 580, 599, 600, 604, 618, 620 MPI_WIN_FLUSH_ALL, 578, 580, 600, 604 MPI_WIN_FLUSH_LOCAL, 578, 600, 604 MPI_WIN_FLUSH_LOCAL_ALL, 578, 601, 604 MPI_WIN_FREE, 370, 491, 498, 543, 559, 560 MPI_WIN_FREE_KEYVAL, 361, 370, 375 MPI_WIN_GET_ATTR, 361, <u>371</u>, 375, 560, 792, 841, 844 MPI_WIN_GET_ERRHANDLER, 456, 460, 1051 MPI_WIN_GET_GROUP, 561 MPI_WIN_GET_INFO, 561, 562, 563, 1037, 1043 MPI_WIN_GET_NAME, 382 MPI_WIN_LOCK, 548, 559, 588, 596 597-599, 601, 603, 605, 608-610 MPI_WIN_LOCK_ALL, 548, 588, 597, 598, 601, 603, 605, 610, 618, 619 MPI_WIN_NULL_COPY_FN, 369, 857, 1040 MPI_WIN_NULL_DELETE_FN, 369, 857, 1040 MPI_WIN_POST, 559, 588, 592, 593, 594-596, 598, 601, 602, 605, 611, 613 MPI_WIN_SET_ATTR, 361, 370, 375, 560, 792, 841, 845 MPI_WIN_SET_ERRHANDLER, 456, 460 MPI_WIN_SET_INFO, <u>562</u>, 1037, 1043 MPI_WIN_SET_NAME, 381 MPI_WIN_SHARED_QUERY, 553, 554, 798, 1039MPI_Win_shared_query_c, 554 MPI_WIN_SHARED_QUERY_CPTR, 556, 1039 MPI_WIN_START, 559, 588, 592, 593-596, 601, 602, 611, 618 MPI_WIN_SYNC, 601, 605-608, 611, 618-620 MPI_WIN_TEST, 594, 595 MPI_WIN_UNLOCK, 559, 580, 588, 597, 599, 604, 605, 608, 609 MPI_WIN_UNLOCK_ALL, 580, 588, 597, 598, 604, 605, 608, 619 MPI_WIN_WAIT, 559, 588, 594, 595, 596, 598, 604, 605, 608, 611, 612 MPI_WTICK, 24, 474, 752 MPI_WTIME, 14, 24, 450, 474, 720, 739, 752 mpiexec, 485, 486, 488, 510, 1038 mpirun, 510

PMPI_, 717, 792 PMPI_AINT_ADD, 24 PMPI_AINT_DIFF, 24 PMPI_ISEND, 792, 795 PMPI_T_EVENT_COPY, 753 PMPI_T_EVENT_GET_SOURCE, 753 PMPI_T_EVENT_GET_TIMESTAMP, 753 PMPI_T_EVENT_READ, 753 PMPI_T_PVAR_READ, 753 PMPI_T_PVAR_READRESET, 753 PMPI_T_PVAR_RESET, 753 PMPI_T_PVAR_START, 753 PMPI_T_PVAR_STOP, 753 PMPI_T_PVAR_WRITE, 753 PMPI_T_SOURCE_GET_TIMESTAMP, 753 PMPI_WTICK, 24 PMPI_WTIME, 24  $\mathbf{2}$