MPI: A Message-Passing Interface Standard Version 4.0

(Draft)

Unofficial, for comment only

Message Passing Interface Forum

September 2, 2020

1	This document describes the 2019 Draft Specification of the Message-Passing Interface
2	(MPI) standard, intended for comment. It is not an official version of the standard. The
3	MPI standard includes point-to-point message-passing, collective communications, group
4	and communicator concepts, process topologies, environmental management, process cre-
5	ation and management, one-sided communications, extended collective operations, external
6	interfaces, I/O, some miscellaneous topics, and a profiling interface. Language bindings for
7	C and Fortran are defined.
8	Historically, the evolution of the standards is from MPI-1.0 (May 5, 1994) to MPI-1.1
9	(June 12, 1995) to MPI-1.2 (July 18, 1997), with several clarifications and additions and
10	published as part of the MPI-2 document, to MPI-2.0 (July 18, 1997), with new functionality,
11	to MPI-1.3 (May 30, 2008), combining for historical reasons the documents 1.1 and 1.2
12	and some errata documents to one combined document, and to MPI-2.1 (June 23, 2008),
13	combining the previous documents. Version MPI-2.2 (September 4, 2009) added additional
14	clarifications and seven new routines. Version MPI-3.0 (September 21, 2012) is an extension
15	of MPI-2.2. Version MPI-3.1 (June 4, 2015) adds clarifications and minor extensions to
16	MPI-3.0.
17	WIT 1-5.0.
18	Comments. Please send comments on MPI to the MPI Forum as follows:
19	comments. I lease send comments on MFT to the MFT for thin as follows.
20	1. Subscribe to https://lists.mpi-forum.org/mailman/listinfo/mpi-comments
21	2. Send your comment to: mpi-comments@lists.mpi-forum.org, together with the version
22	of the MPI standard and the page and line numbers on which you are commenting.
23	Your comment will be forwarded to MPI Forum committee members for consideration.
24	Messages sent from an unsubscribed e-mail address will not be considered.
25	Messages sent from an unsubscribed e-man address will not be considered.
26	
27	
28 29	
30	
31	
32	
33	
34	
35	
36	
37	
38	
39	
40	
41	
42	
43	
44	
45	©1993, 1994, 1995, 1996, 1997, 2008, 2009, 2012, 2015, 2019 University of Tennessee,
46	Knoxville, Tennessee. Permission to copy without fee all or part of this material is granted,
47	provided the University of Tennessee copyright notice and the title of this document appear,
48	and notice is given that copying is by permission of the University of Tennessee.

Unofficial Draft for Comment Only

2019 Draft Specification, November, 2019. 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 persistent collectives, 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 draft version of the MPI-3 standard contains significant extensions to MPI functionality, including nonblocking collectives, new one-sided communication operations, and Fortran 2008 bindings. Unlike MPI-2.2, this standard is considered a major update to the MPI standard. As with previous versions, new features have been adopted only when there were compelling needs for the users. Some features, however, may have more than a minor impact on existing MPI implementations.

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

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

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

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

 $\mathbf{5}$

 31

Version 1.2: July 18, 1997. The MPI-2 Forum introduced MPI-1.2 as Chapter 3 in the $\mathbf{2}$ 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 $\mathbf{5}$ 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 $\overline{7}$ and MPI-2.

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

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

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

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

~ .	dian was processed by EAEA on May 5, 1994.
24	
25	
26	
27	
28	
29	
30	
31	
32	
33	
34	
35	
36	
37	
38	
39	
40	
41	
42	
43	
44	
45	
46	
47	

Contents

Li	st of	Figures x	viii
Li	st of	Tables	xx
A	cknov	wledgments	cxii
1	Intr	roduction 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 2019 Draft Specification	5
	1.9	Who Should Use This Standard?	5
	1.10	What Platforms Are Targets for Implementation?	5
		What Is Included in the Standard?	5
		What Is Not Included in the Standard?	6
	1.13	Organization of This Document	6
_			0
2		Terms and Conventions	9
	2.1	Document Notation	9
	2.2	Naming Conventions	9
	2.3	Procedure Specification	10
	2.4	Semantic Terms	11
	2.5	Data Types	12
		2.5.1 Opaque Objects	12
		2.5.2 Array Arguments	14
		2.5.3 State	14
		2.5.4 Named Constants	15
		2.5.5 Choice	16
		2.5.6 Absolute Addresses and Relative Address Displacements	16
		2.5.7 File Offsets	16
	0.0	2.5.8 Counts	17
	2.6	Language Binding	17
		2.6.1 Deprecated and Removed Interfaces	17
		2.6.2 Fortran Binding Issues	18

		2.6.3	C Binding Issues	3
2.6.4 Functions and Ma		2.6.4	Functions and Macros	D
	2.7	Proces	sses	D
	2.8	Error	Handling	D
	2.9	Imple	mentation Issues	2
		2.9.1	Independence of Basic Runtime Routines	2
		2.9.2	Interaction with Signals	2
	2.10	Exam	ples $\ldots \ldots 23$	3
3	Poir	nt-to-F	Point Communication 25	5
Ŭ	3.1		luction	
	3.2		ing Send and Receive Operations	
		3.2.1	Blocking Send	
		3.2.2		
		3.2.3	Message Data 2' Message Envelope 2	
		3.2.4	Blocking Receive	
		3.2.5	Return Status	
		3.2.6	Passing MPI_STATUS_IGNORE for Status	
		3.2.7	Send-Receive	
	3.3	Data '	Type Matching and Data Conversion	8
		3.3.1	Type Matching Rules	
			Type MPI_CHARACTER	9
		3.3.2	Data Conversion	0
	3.4	Comm	nunication Modes	2
	3.5		ntics of Point-to-Point Communication	6
	3.6	Buffer	Allocation and Usage	9
		3.6.1	Model Implementation of Buffered Mode	1
	3.7	Nonbl	ocking Communication	2
		3.7.1	Communication Request Objects	3
		3.7.2	Communication Initiation	4
		3.7.3	Communication Completion	0
		3.7.4	Semantics of Nonblocking Communications	4
		3.7.5	Multiple Completions	5
		3.7.6	Non-Destructive Test of status	2
	3.8	Probe	and Cancel	3
		3.8.1	Probe	3
		3.8.2	Matching Probe	6
		3.8.3	Matched Receives	3
		3.8.4	Cancel	0
	3.9	Persis	tent Communication Requests	2
	3.10	Null F	Processes	3
4	Dat	atypes	8	9
	4.1	Derive	ed Datatypes	9
		4.1.1	Type Constructors with Explicit Addresses 9	
		4.1.2	Datatype Constructors 91	
		4.1.3	Subarray Datatype Constructor	
		4.1.4	Distributed Array Datatype Constructor	3

		4.1.5 Address and Size Functions	108
		4.1.6 Lower-Bound and Upper-Bound Markers	112
		4.1.7 Extent and Bounds of Datatypes	114
		4.1.8 True Extent of Datatypes	116
		4.1.9 Commit and Free	117
		4.1.10 Duplicating a Datatype	119
		4.1.11 Use of General Datatypes in Communication	119
		4.1.12 Correct Use of Addresses	123
		4.1.13 Decoding a Datatype	124
		4.1.14 Examples	131
	4.2	Pack and Unpack	139
	4.3	Canonical MPI_PACK and MPI_UNPACK	146
	1.0		110
5	Coll	lective Communication	151
	5.1	Introduction and Overview	151
	5.2	Communicator Argument	
		5.2.1 Specifics for Intracommunicator Collective Operations	
		5.2.2 Applying Collective Operations to Intercommunicators	155
		5.2.3 Specifics for Intercommunicator Collective Operations	156
	5.3	Barrier Synchronization	157
	5.4	Broadcast	158
	0.1	5.4.1 Example using MPI_BCAST	159
	5.5	Gather	159
	0.0	5.5.1 Examples using MPI_GATHER, MPI_GATHERV	162
	5.6	Scatter	169
	0.0	5.6.1 Examples using MPI_SCATTER, MPI_SCATTERV	172
	5.7	Gather-to-all	175
	0.1	5.7.1 Example using MPI_ALLGATHER	178
	5.8	All-to-All Scatter/Gather	178
	5.9	Global Reduction Operations	184
	0.9	5.9.1 Reduce	184
		5.9.2 Predefined Reduction Operations	184
		5.9.3 Signed Characters and Reductions	180
		5.9.3 Signed Characters and Reductions 5.9.4 MINLOC and MAXLOC	189 189
			193
		Example of User-Defined Reduce	196
		5.9.6 All-Reduce	198
	F 10	5.9.7 Process-Local Reduction	199
	5.10	Reduce-Scatter	201
		5.10.1 MPI_REDUCE_SCATTER_BLOCK	201
		5.10.2 MPI_REDUCE_SCATTER	202
	5.11	Scan	204
		5.11.1 Inclusive Scan	204
		5.11.2 Exclusive Scan	205
		5.11.3 Example using MPI_SCAN	206
	5.12	Nonblocking Collective Operations	208
		5.12.1 Nonblocking Barrier Synchronization	210
		5.12.2 Nonblocking Broadcast	210

			Example using MPI_IBCAST	211
		5.12.3	Nonblocking Gather	212
		5.12.4	Nonblocking Scatter	214
		5.12.5	Nonblocking Gather-to-all	216
		5.12.6	Nonblocking All-to-All Scatter/Gather	218
		5.12.7	Nonblocking Reduce	221
			Nonblocking All-Reduce	
		5.12.9	Nonblocking Reduce-Scatter with Equal Blocks	223
			Nonblocking Reduce-Scatter	
			Nonblocking Inclusive Scan	225
			Nonblocking Exclusive Scan	226
	5.13		ent Collective Operations	
			Persistent Barrier Synchronization	
			Persistent Broadcast	
			Persistent Gather	
		5.13.4	Persistent Scatter	232
		5.13.5	Persistent Gather-to-all	234
		5.13.6	Persistent All-to-All Scatter/Gather	236
			Persistent Reduce	
			Persistent All-Reduce	
			Persistent Reduce-Scatter with Equal Blocks	
			Persistent Reduce-Scatter	
			Persistent Inclusive Scan	
		5 13 12	Persistent Exclusive Scan	244
	5 14	Correc	tness	245
	· · · · ·			
				210
6	Gro	ups, C	ontexts, Communicators, and Caching	2 10 2 53
6	Gro 6.1		ontexts, Communicators, and Caching	253
6				253 253
6		Introd	uction	253 253 253
6		Introd 6.1.1 6.1.2	uction	253 253 253 254
6	6.1	Introd 6.1.1 6.1.2	uction	253 253 253 254 256
6	6.1	Introd 6.1.1 6.1.2 Basic (6.2.1 6.2.2	uction	253 253 253 254 256 256
6	6.1	Introd 6.1.1 6.1.2 Basic (6.2.1 6.2.2	uction	253 253 253 254 256 256
6	6.1	Introd 6.1.1 6.1.2 Basic (6.2.1 6.2.2	uction	253 253 254 256 256 256
6	6.1	Introd 6.1.1 6.1.2 Basic 0 6.2.1 6.2.2 6.2.3 6.2.3 6.2.4	uction	253 253 253 254 256 256 256 256
6	6.16.2	Introd 6.1.1 6.1.2 Basic 0 6.2.1 6.2.2 6.2.3 6.2.3 6.2.4	uction	253 253 254 256 256 256 256 257 257
6	6.16.2	Introd 6.1.1 6.1.2 Basic (6.2.1 6.2.2 6.2.3 6.2.4 Group	uction	253 253 254 256 256 256 256 257 257 258
6	6.16.2	Introd 6.1.1 6.1.2 Basic 0 6.2.1 6.2.2 6.2.3 6.2.3 6.2.4 Group 6.3.1	uction	253 253 254 256 256 256 257 257 257 258 258 258 260
6	6.16.2	Introd 6.1.1 6.1.2 Basic 0 6.2.1 6.2.2 6.2.3 6.2.4 Group 6.3.1 6.3.2 6.3.3	uction Features Needed to Support Libraries MPI's Support for Libraries MPI's Support for Libraries Concepts Concepts Groups Contexts Intra-Communicators Contexts Predefined Intra-Communicators Contexts Group Accessors Constructors Group Destructors Constructors	253 253 254 256 256 256 256 257 257 258 258 258 260 266
6	6.16.26.3	Introd 6.1.1 6.1.2 Basic 0 6.2.1 6.2.2 6.2.3 6.2.4 Group 6.3.1 6.3.2 6.3.3	uction Features Needed to Support Libraries MPI's Support for Libraries MPI's Support for Libraries Concepts Groups Groups Groups Intra-Communicators Intra-Communicators Predefined Intra-Communicators Group Accessors Group Destructors Group Destructors Group Destructors Group Management	253 253 254 256 256 256 256 257 257 258 258 258 260 266 267
6	6.16.26.3	Introd 6.1.1 6.1.2 Basic 0 6.2.1 6.2.2 6.2.3 6.2.3 6.2.4 Group 6.3.1 6.3.2 6.3.3 Comm	uction Features Needed to Support Libraries MPI's Support for Libraries	253 253 254 256 256 256 257 257 258 258 258 260 266 267 267
6	6.16.26.3	Introd 6.1.1 6.1.2 Basic 0 6.2.1 6.2.2 6.2.3 6.2.3 6.2.4 Group 6.3.1 6.3.2 6.3.3 Comm 6.4.1	uction Features Needed to Support Libraries MPI's Support for Libraries MPI's Support for Libraries Concepts Groups Groups Groups Intra-Communicators Intra-Communicators Predefined Intra-Communicators Group Accessors Group Destructors Group Destructors Group Destructors Group Management	253 253 254 256 256 256 257 257 257 258 258 260 266 267 267 269
6	6.16.26.3	Introd 6.1.1 6.1.2 Basic 0 6.2.1 6.2.2 6.2.3 6.2.4 Group 6.3.1 6.3.2 6.3.3 Comm 6.4.1 6.4.2	uctionFeatures Needed to Support LibrariesMPI's Support for LibrariesMPI's Support for LibrariesConceptsGroupsContextsIntra-CommunicatorsIntra-CommunicatorsPredefined Intra-CommunicatorsManagementGroup AccessorsGroup DestructorsGroup DestructorsGroup DestructorsIntra-CommunicatorsUnicator ManagementCommunicator AccessorsCommunicator ConstructorsCommunicator Constructors	253 253 254 256 256 256 257 257 258 258 260 266 267 267 269 286
6	6.16.26.36.4	Introd 6.1.1 6.1.2 Basic 0 6.2.1 6.2.2 6.2.3 6.2.3 6.2.4 Group 6.3.1 6.3.2 6.3.3 Comm 6.4.1 6.4.2 6.4.3 6.4.4	uction Features Needed to Support Libraries MPI's Support for Libraries	253 253 254 256 256 256 257 257 258 258 260 266 267 267 267 269 286 286
6	6.16.26.3	Introd 6.1.1 6.1.2 Basic 0 6.2.1 6.2.2 6.2.3 6.2.4 Group 6.3.1 6.3.2 6.3.3 Comm 6.4.1 6.4.2 6.4.3 6.4.4 Motiva	uctionFeatures Needed to Support LibrariesMPI's Support for Libraries	253 253 254 256 256 256 257 257 258 258 260 266 267 267 269 286 286 289
6	6.16.26.36.4	Introd 6.1.1 6.1.2 Basic 0 6.2.1 6.2.2 6.2.3 6.2.4 Group 6.3.1 6.3.2 6.3.3 Comm 6.4.1 6.4.2 6.4.3 6.4.4 Motiva 6.5.1	uction Features Needed to Support Libraries MPI's Support for Libraries	253 253 254 256 256 256 257 257 258 258 260 266 267 269 286 289 289
6	6.16.26.36.4	Introd 6.1.1 6.1.2 Basic 0 6.2.1 6.2.2 6.2.3 6.2.4 Group 6.3.1 6.3.2 6.3.3 Comm 6.4.1 6.4.2 6.4.3 6.4.4 Motiva	uctionFeatures Needed to Support LibrariesMPI's Support for Libraries	253 253 254 256 256 256 257 257 258 258 260 266 267 267 269 286 286 289

		6.5.4	Example #4	291
		6.5.5	Library Example $\#1$	292
		6.5.6	Library Example $\#2$	294
	6.6	Inter-0	Communication	296
		6.6.1	Inter-Communicator Accessors	
		6.6.2	Inter-Communicator Operations	
		6.6.3	Inter-Communication Examples	303
			Example 1: Three-Group "Pipeline"	
			Example 2: Three-Group "Ring"	
	6.7	Cachi	ng	306
		6.7.1	Functionality	306
		6.7.2	Communicators	307
		6.7.3	Windows	313
		6.7.4	Datatypes	
		6.7.5	Error Class for Invalid Keyval	
		6.7.6	Attributes Example	320
	6.8		ng Objects	
	6.9		lizing the Loosely Synchronous Model	
		6.9.1	Basic Statements	
		6.9.2	Models of Execution	
		0.0.2	Static Communicator Allocation	
			Dynamic Communicator Allocation	
			The General Case	
7	Pro	$\mathbf{cess} \ \mathbf{T}$	opologies	329
7	Pro 7.1	cess T Introd	opologies uction	
7		Introd Virtua	luction	329 330
7	7.1	Introd Virtua Embe	uction	329 330 330
7	$7.1 \\ 7.2$	Introd Virtua Embee Overv	Juction	329 330 330 330
7	$7.1 \\ 7.2 \\ 7.3$	Introd Virtua Embee Overv	Juction	329 330 330 330 332
7	7.1 7.2 7.3 7.4	Introd Virtua Embee Overv	Juction	329 330 330 330 332
7	7.1 7.2 7.3 7.4	Introd Virtua Embee Overv Topole	Juction	329 330 330 330 332 332 333
7	7.1 7.2 7.3 7.4	Introd Virtua Embed Overv Topola 7.5.1 7.5.2 7.5.3	Juction	329 330 330 330 332 332 333 333 334
7	7.1 7.2 7.3 7.4	Introd Virtua Embed Overv Topola 7.5.1 7.5.2 7.5.3	Juction	329 330 330 330 332 332 333
7	7.1 7.2 7.3 7.4	Introd Virtua Embed Overv Topola 7.5.1 7.5.2 7.5.3	Juction	329 330 330 330 332 332 333 333 334
7	7.1 7.2 7.3 7.4	Introd Virtua Embed Overv Topold 7.5.1 7.5.2 7.5.3 7.5.4	Juction	329 330 330 332 332 332 333 334 334
7	7.1 7.2 7.3 7.4	Introd Virtua Embed Overv Topola 7.5.1 7.5.2 7.5.3 7.5.4 7.5.5	Juction	329 330 330 332 332 333 334 336 343
7	7.1 7.2 7.3 7.4	Introd Virtua Embed Overv Topola 7.5.1 7.5.2 7.5.3 7.5.4 7.5.5 7.5.6	Juction	329 330 330 332 332 333 334 336 343 351
7	7.1 7.2 7.3 7.4	Introd Virtua Embed Overv Topola 7.5.1 7.5.2 7.5.3 7.5.4 7.5.5 7.5.6 7.5.7 7.5.8	Juction	329 330 330 332 332 333 334 336 343 351 353
7	7.1 7.2 7.3 7.4 7.5	Introd Virtua Embed Overv Topola 7.5.1 7.5.2 7.5.3 7.5.4 7.5.5 7.5.6 7.5.7 7.5.8	Juction	329 330 330 332 332 333 334 336 343 351 353 353
7	7.1 7.2 7.3 7.4 7.5	Introd Virtua Embed Overv Topold 7.5.1 7.5.2 7.5.3 7.5.4 7.5.5 7.5.6 7.5.7 7.5.8 Neighl 7.6.1 7.6.2	uction	329 330 330 332 332 333 334 334 336 343 351 353 353 355
7	7.1 7.2 7.3 7.4 7.5	Introd Virtua Embed Overv Topold 7.5.1 7.5.2 7.5.3 7.5.4 7.5.5 7.5.6 7.5.7 7.5.8 Neighl 7.6.1 7.6.2	uction	329 330 330 332 332 333 334 336 343 351 353 355 356
7	7.1 7.2 7.3 7.4 7.5 7.6	Introd Virtua Embed Overv Topold 7.5.1 7.5.2 7.5.3 7.5.4 7.5.5 7.5.6 7.5.7 7.5.8 Neighl 7.6.1 7.6.2	uction	329 330 330 332 332 333 334 351 353 355 356 360 365 366
7	7.1 7.2 7.3 7.4 7.5 7.6	Introd Virtua Embed Overv Topold 7.5.1 7.5.2 7.5.3 7.5.4 7.5.5 7.5.6 7.5.7 7.5.8 Neighl 7.6.1 7.6.2 Nonbl 7.7.1 7.7.2	uction	329 330 330 332 332 333 334 353 353 355 356 360 365
7	7.1 7.2 7.3 7.4 7.5 7.6	Introd Virtua Embed Overv Topold 7.5.1 7.5.2 7.5.3 7.5.4 7.5.5 7.5.6 7.5.7 7.5.8 Neighl 7.6.1 7.6.2 Nonbl 7.7.1 7.7.2	uction	329 330 330 332 332 333 334 351 353 355 356 360 365 366
7	7.1 7.2 7.3 7.4 7.5 7.6 7.6	Introd Virtua Embed Overv Topold 7.5.1 7.5.2 7.5.3 7.5.4 7.5.5 7.5.6 7.5.7 7.5.8 Neighl 7.6.1 7.6.2 Nonbl 7.7.1 7.7.2	uction	329 330 330 332 332 333 334 351 353 355 356 360 365 366 368
7	7.1 7.2 7.3 7.4 7.5 7.6 7.6	Introd Virtua Embed Overv Topold 7.5.1 7.5.2 7.5.3 7.5.4 7.5.5 7.5.6 7.5.7 7.5.8 Neighl 7.6.1 7.6.1 7.6.2 Nonbl 7.7.1 7.7.2 Persis: 7.8.1 7.8.2	uction	329 330 330 332 332 333 334 353 353 355 356 360 365 366 368 371

8	MPI		onmental Management	383
	8.1	Implem	nentation Information	383
		8.1.1	Version Inquiries	383
		8.1.2	Environmental Inquiries	385
			Tag Values	385
			Host Rank	385
			IO Rank	386
			Clock Synchronization	386
			Inquire Processor Name	
	8.2		y Allocation	
	8.3		Handling	
			Error Handlers for Communicators	
			Error Handlers for Windows	
			Error Handlers for Files	
			Error Handlers for Sessions	
			Freeing Errorhandlers and Retrieving Error Strings	
	8.4	Error (Codes and Classes	401
	8.5		Classes, Error Codes, and Error Handlers	
	8.6		and Synchronization	
	0.0	1 1111010		100
9	The	Info O	bject	411
10			itialization, Creation, and Management	419
	10.1	Introdu	action	419
	10.2	The W	orld Model	420
		10.2.1	Starting MPI Processes	420
			Finalizing MPI	
			Determining Whether MPI Has Been Initialized When Using the World	
			Model	
			Allowing User Functions at MPI Finalization	
	10.3		ssions Model	
			Session Creation and Destruction Methods $\ldots \ldots \ldots \ldots \ldots$	
		10.3.2	Processes Sets	434
	A		Runtime Query Functions	
		10.3.4	Sessions Model Examples	438
	10.4	Commo	on Elements of Both Process Models	444
		10.4.1	MPI Functionality that is Always Available	444
		10.4.2	Aborting MPI Processes	444
	10.5	Portab	le MPI Process Startup	446
	10.6	MPI an	d Threads	448
		10.6.1	General	448
		10.6.2	Clarifications	449
	10.7	The Dy	ynamic Process Model	452
		10.7.1	Starting Processes	452
			The Runtime Environment	
	10.8		s Manager Interface	
			Processes in MPI	
		10.8.2	Starting Processes and Establishing Communication	454

10.8.3 Starting Multiple Executables and Establishing Communication	459
10.8.4 Reserved Keys	462
10.8.5 Spawn Example	463
10.9 Establishing Communication	464
10.9.1 Names, Addresses, Ports, and All That	465
10.9.2 Server Routines	466
10.9.3 Client Routines	468
10.9.4 Name Publishing	469
10.9.5 Reserved Key Values	472
10.9.6 Client/Server Examples	472
10.10Other Functionality	475
10.10.1 Universe Size	475
10.10.2 Singleton MPI Initialization	476
10.10.3 MPI_APPNUM	
$10.10.4$ Releasing Connections $\ldots \ldots \ldots$	477
10.10.5 Another Way to Establish MPI Communication	
11 One-Sided Communications	481
11.1 Introduction	
11.2 Initialization	
11.2.1 Window Creation	
11.2.2 Window That Allocates Memory	485
11.2.3 Window That Allocates Shared Memory	
11.2.4 Window of Dynamically Attached Memory	
11.2.5 Window Destruction	
11.2.6 Window Attributes	494
11.2.7 Window Info	496
11.3 Communication Calls	
11.3.1 Put	
11.3.2 Get \ldots	
11.3.3 Examples for Communication Calls	
11.3.4 Accumulate Functions	
Accumulate Function	
Get Accumulate Function	507
$ Fetch and Op Function \qquad \dots \qquad $	
Compare and Swap Function	
11.3.5 Request-based RMA Communication Operations	
11.4 Memory Model	
11.5 Synchronization Calls	
11.5.1 Fence	
11.5.2 General Active Target Synchronization	523
11.5.3 Lock	527
11.5.4 Flush and Sync	530
11.5.5 Assertions \ldots	
11.5.6 Miscellaneous Clarifications	534
11.6 Error Handling	534
11.6.1 Error Handlers	534
11.6.2 Error Classes	534

11.7	Seman	tics and Correctness	534
	11.7.1	Atomicity	543
	11.7.2	Ordering	543
	11.7.3	Progress	544
	11.7.4	Registers and Compiler Optimizations	546
11.8	Examp	\mathbf{ples}	547
12 Ext	ornal I	nterfaces	557
		uction	557
		alized Requests	557
12.2		Examples	562
12.3		ating Information with Status	562
12.0	1155001		004
13 I/O			567
13.1	Introd	uction	567
	13.1.1	Definitions	567
13.2		anipulation	
	13.2.1	Opening a File	569
	13.2.2	Closing a File	572
		Deleting a File	572
	13.2.4	Resizing a File	573
	13.2.5	Preallocating Space for a File	574
		Querying the Size of a File	575
	13.2.7	Querying File Parameters	575
	13.2.8	File Info	577
		Reserved File Hints	578
13.3	File Vi	iews	580
13.4	Data A	Access	583
	13.4.1	Data Access Routines	583
		Positioning	584
		Synchronism	584
		Coordination	585
		Data Access Conventions	585
	13.4.2	Data Access with Explicit Offsets	586
		Data Access with Individual File Pointers	591
		Data Access with Shared File Pointers	599
		Noncollective Operations	600
		Collective Operations	602
		Seek	604
	13.4.5		606
13.5		teroperability	613
		Datatypes for File Interoperability	615
		External Data Representation: "external32"	617
		User-Defined Data Representations	620
		Extent Callback	621
		Datarep Conversion Functions	621
	13.5.4	Matching Data Representations	623
13.6		tency and Semantics	624
		· ·	

	13.6.1	File Consistency	624
	13.6.2	Random Access vs. Sequential Files	627
	13.6.3	Progress	628
	13.6.4	Collective File Operations	628
		Nonblocking Collective File Operations	
		Type Matching	
		Miscellaneous Clarifications	
		MPI_Offset Type	
		Logical vs. Physical File Layout	
		OFile Size	
		1 Examples	
		Asynchronous I/O	
13.7	I/O E	rror Handling	
		rror Classes	
13.9	Exami	ples	635
		Double Buffering with Split Collective I/O	
		Subarray Filetype Constructor	
14 Too			641
14.1	Introd	uction	641
14.2	Profili	ng Interface	641
	14.2.1	Requirements	641
	14.2.2	Discussion	642
	14.2.3	Logic of the Design	642
		Miscellaneous Control of Profiling	
		Profiler Implementation Example	
		MPI Library Implementation Example	
		Systems with Weak Symbols	644
		Systems without Weak Symbols	645
	14.2.7	Complications	645
		Multiple Counting	
		Linker Oddities	646
		Fortran Support Methods	646
	14.2.8	Multiple Levels of Interception	646
14.3		IPI Tool Information Interface	647
	14.3.1	Verbosity Levels	648
	14.3.2	Binding MPI Tool Information Interface Variables to MPI Objects .	648
	14.3.3		
	14.3.4	Initialization and Finalization	
	14.3.5	Datatype System	651
		Control Variables	
		Control Variable Query Functions	654
		Example: Printing All Control Variables	
		Handle Allocation and Deallocation	
		Control Variable Access Functions	659
		Example: Reading the Value of a Control Variable	660
	14.3.7	Performance Variables	
		Performance Variable Classes	661

	Performance Variable Query Functions	663
	Performance Experiment Sessions	666
	Handle Allocation and Deallocation	666
	Starting and Stopping of Performance Variables	668
	Performance Variable Access Functions	669
	Example: Tool to Detect Receives with Long Unexpected Message	
	Queues	671
14.3.8	Events	673
	Event Sources	673
	Callback Safety Requirements	675
	Event Type Query Functions	677
	Handle Allocation and Deallocation	680
	Handling Dropped Events	684
	Reading Event Data	686
	Reading Event Meta Data	687
14.3.9	Variable Categorization	688
	Category Query Functions	
	Category Member Query Functions	691
14.3.1	$0 \text{ Return Codes for the MPI Tool Information Interface} \dots \dots \dots$	693
14.3.1	1 Profiling Interface	694
		a o F
15 Deprecate	cated since MPI-2.0	697
	cated since MPI-2.2	
	cated since MPI-3.2	
15.4 Depre	cated since MPI-4.0	700
16 Removed	Interfaces	703
	ved MPI-1 Bindings	
16.1.1	Overview	703
	Removed MPI-1 Functions	
	Removed MPI-1 Datatypes	
	Removed MPI-1 Constants	
	Removed MPI-1 Callback Prototypes	704
	Bindings	704
	, č	
17 Backward	Incompatibilities	705
17.1 Backw	vard Incompatible since MPI-3.2	705
10 T amount and	Diadiana	707
18 Language 18.1 Fortra		707 707
	Overview	707
	Fortran Support Through the mpi_f08 Module	708
	Fortran Support Through the mpi_100 Module	
	Fortran Support Through the mpif.h Include File	713
	Interface Specifications, Procedure Names, and the Profiling Interface	
	MPI for Different Fortran Standard Versions	719
	Requirements on Fortran Compilers	
10.1.1	requirements on retrien compacts	. 20

	18.1.8	Additional Support for Fortran Register-Memory-Synchronization	724
	18.1.9	Additional Support for Fortran Numeric Intrinsic Types	725
		Parameterized Datatypes with Specified Precision and Exponent Rang	<mark>e</mark> 726
		Support for Size-specific MPI Datatypes	730
		Communication With Size-specific Types	732
	18.1.10	Problems With Fortran Bindings for MPI	733
		Problems Due to Strong Typing	735
		2 Problems Due to Data Copying and Sequence Association with Sub-	
		script Triplets	735
	18.1.13	Problems Due to Data Copying and Sequence Association with Vector	
		Subscripts	738
	18.1.14	Special Constants	739
		5 Fortran Derived Types	739
		Optimization Problems, an Overview	741
		Problems with Code Movement and Register Optimization	742
		Nonblocking Operations	742
		Persistent Operations	743
		One-sided Communication	743
		MPI_BOTTOM and Combining Independent Variables in Datatypes .	743
		Solutions	744
		The Fortran ASYNCHRONOUS Attribute	744
		Calling MPI_F_SYNC_REG	746
		A User Defined Routine Instead of MPI_F_SYNC_REG	747
		Module Variables and COMMON Blocks	747
		The (Poorly Performing) Fortran VOLATILE Attribute	748
		The Fortran TARGET Attribute	748
	18.1.18	8 Temporary Data Movement and Temporary Memory Modification .	748
		Permanent Data Movement	750
	18.1.20	$ Comparison with C \dots $	750
18.2	Langua	age Interoperability	755
	18.2.1	Introduction	755
		Assumptions	755
	18.2.3	Initialization	755
	18.2.4	Transfer of Handles	756
	18.2.5	Status	758
	18.2.6	MPI Opaque Objects	761
		Datatypes	761
		Callback Functions	762
		Error Handlers	763
		Reduce Operations	763
	18.2.7	Attributes	763
	18.2.8	Extra-State	767
		Constants	767
	18.2.10	Interlanguage Communication	768

Α	Lan	guage	Bindings Summary	771
	A.1	Define	d Values and Handles	. 771
		A.1.1	Defined Constants	. 771
		A.1.2	Types	. 785
		A.1.3	Prototype Definitions	. 786
			C Bindings	
			Fortran 2008 Bindings with the mpi_f08 Module	
			Fortran Bindings with mpif.h or the mpi Module	
		A.1.4	Deprecated Prototype Definitions	
			Info Keys	
			Info Values	
	A.2		dings	
			Point-to-Point Communication C Bindings	
		A.2.2	Datatypes C Bindings	
		A.2.3	Collective Communication C Bindings	. 799
		A.2.4	Groups, Contexts, Communicators, and Caching C Bindings	
			Process Topologies C Bindings	
		A.2.6	MPI Environmental Management C Bindings	
		A.2.7		. 809
			Process Creation and Management C Bindings	
			One-Sided Communications C Bindings	
) External Interfaces C Bindings	
			I/O C Bindings	
			2 Language Bindings C Bindings	
			B Tools / Profiling Interface C Bindings	
			Tools / MPI Tool Information Interface C Bindings	
			Deprecated C Bindings	
	Δ 3		n 2008 Bindings with the mpi_f08 Module	
	А.0		Point-to-Point Communication Fortran 2008 Bindings	
		A.3.1 A.3.2	Datatypes Fortran 2008 Bindings	
		A.3.2		
			Groups, Contexts, Communicators, and Caching Fortran 2008 Binding	
			Process Topologies Fortran 2008 Bindings	<u> </u>
			MPI Environmental Management Fortran 2008 Bindings	
			The Info Object Fortran 2008 Bindings	
		A.3.7	•	
		A.3.9	Process Creation and Management Fortran 2008 Bindings	
			One-Sided Communications Fortran 2008 Bindings	
			External Interfaces Fortran 2008 Bindings	
			I/O Fortran 2008 Bindings	
			Language Bindings Fortran 2008 Bindings	
			Tools / Profiling Interface Fortran 2008 Bindings	
	A 4		Deprecated Fortran 2008 Bindings	
	A.4		n Bindings with mpif.h or the mpi Module	
		A.4.1	Point-to-Point Communication Fortran Bindings	
		A.4.2	Datatypes Fortran Bindings	
		A.4.3	Collective Communication Fortran Bindings	
		A.4.4	Groups, Contexts, Communicators, and Caching Fortran Bindings	
		A.4.5	Process Topologies Fortran Bindings	. 894

A.4.6 MPI Environmental Management Fortran Bindings 8	897	
A.4.7 The Info Object Fortran Bindings	899	
A.4.8 Process Creation and Management Fortran Bindings	900	
A.4.9 One-Sided Communications Fortran Bindings	902	
A.4.10 External Interfaces Fortran Bindings	906	
A.4.11 I/O Fortran Bindings	907	
A.4.12 Language Bindings Fortran Bindings	911	
	912	
A.4.14 Deprecated Fortran Bindings	912	
B Change-Log 9	913	
B.1 Changes from Version 3.2 to Version 4.0	913	
	913	
	914	
	914	
	915	
	915	
B.3.2 Changes in MPI-3.1	917	
	918	
	918	
B.4.2 Changes in MPI-3.0	919	
	924	
B.6 Changes from Version 2.0 to Version 2.1	926	
Bibliography 9	931	
General Index 9	937	
General Index 5	101	
Examples Index 9	942	
MPI Constant and Predefined Handle Index 9	945	
MPI Declarations Index 9	951	
MPI Callback Function Prototype Index 9	952	
MPI Function Index		

List of Figures

5.1	Collective communications, an overview	153
5.2	Intercommunicator allgather	156
5.3	Intercommunicator reduce-scatter	157
5.4	Gather example	163
5.5	Gatherv example with strides	164
5.6	Gatherv example, 2-dimensional	165
5.7	Gatherv example, 2-dimensional, subarrays with different sizes	166
5.8	Gatherv example, 2-dimensional, subarrays with different sizes and strides .	168
5.9	Scatter example	173
5.10	Scattery example with strides	173
5.11	Scatterv example with different strides and counts	175
5.12		247
5.13	Overlapping Communicators Example	251
6.1	Intercommunicator creation using MPI_COMM_CREATE	275
6.2	Intercommunicator construction with MPI_COMM_SPLIT	279
6.3	Recursive communicator creation with MPI_COMM_SPLIT_TYPE	284
6.4	Three-group pipeline	303
6.5	Three-group ring	304
7.1	Neighborhood gather communication example.	358
7.2	Set-up of process structure for two-dimensional parallel Poisson solver	378
7.3	Communication routine with local data copying and sparse neighborhood	
	all-to-all	379
7.4	Communication routine with sparse neighborhood all-to-all-w and without	
	local data copying.	380
7.5	Two-dimensional parallel Poisson solver with persistent sparse neighborhood	0.01
	all-to-all-w and without local data copying.	381
10.1	Session handle to communicator	432
	Process set examples	435
10.2		100
11.1	Schematic description of the public/private window operations in the	
	MPI_WIN_SEPARATE memory model for two overlapping windows	517
11.2	Active target communication	520
	Active target communication, with weak synchronization	521
	Passive target communication	522
	Active target communication with several processes	525
11.6	Symmetric communication	545

11.7	Deadlock situation	5
11.8	No deadlock	ý
13.1	Etypes and filetypes	3
13.2	Partitioning a file among parallel processes	3
13.3	Displacements	L
13.4	Example array file layout	3
13.5	Example local array filetype for process 1	3
18.1	Status conversion routines)

List of Tables

2.1	Deprecated and Removed constructs	9
3.1 3.2 3.3 3.4	Predefined MPI datatypes corresponding to C datatypes	27 28 29 29
4.1	combiner values returned from MPI_TYPE_GET_ENVELOPE 12	25
6.1	MPI_COMM_* Function Behavior (in Inter-Communication Mode) 29)8
8.1 8.2	Error classes (Part 1)40Error classes (Part 2)40	
10.1	List of MPI Functions that can be called at any time within an MPI program, including prior to MPI initialization and following MPI finalization 44	15
	C types of attribute value argument to MPI_WIN_GET_ATTR and MPI_WIN_SET_ATTR	95
11.2	MPI_WIN_SET_ATTR. 49 Error classes in one-sided communication routines 53	
13.1	Data access routines	33
13.2	"external32" sizes of predefined datatypes	.8
	"external32" sizes of optional datatypes	.9
13.4	"external32" sizes of C++ datatypes $\ldots \ldots \ldots$	9
13.5	I/O Error Classes	6
14.1	MPI tool information interface verbosity levels	19
14.2	Constants to identify associations of variables	19
14.3	MPI datatypes that can be used by the MPI tool information interface 65	52
14.4	Scopes for control variables	6
14.5	Hierarchy of safety requirement levels for event callback routines	' 6
14.6	List of MPI functions that when called from within a callback function may not return MPI_T_ERR_NOT_ACCESSIBLE	77
14.7	Return codes used in functions of the MPI tool information interface 69	
16.1	Removed MPI-1 functions and their replacements)3
	Removed MPI-1 datatypes and their replacements	
	Removed MPI-1 constants	
16.4	Removed MPI-1 callback prototypes and their replacements)4

8.1 Specific Fortran procedure names and related calling conventions8.2 Occurrence of Fortran optimization problems	. 715 . 741

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		
-		mmission of the Eur	opean Community through	18		
Esprit project P6643 (PI	PPE).			19		
				20		
MPI-1.2 and MPI-2.0:				21		
<i>π</i> ι				22		
Those who served as prin	mary coordinators in	n MPI-1.2 and MPI-	2.0 are:	23 24		
• Ewing Lusk, Convener and Meeting Chair						
• Ewing Ease, Conve	ther and wreeting en	liuli		25 26		
• Steve Huss-Lederman, Editor						
. Ening Luch Miner	11			27 28		
• Ewing Lusk, Miscellany						
• Bill Saphir, Process Creation and Management						
• Marc Snir, One-Sid	led Communications	3		31		
				32		
• Bill Gropp and Ant	thony Skjellum, Ext	ended Collective Op	perations	33		
• Steve Huss-Lederm	an, External Interfa	ices		34		
				35 36		
• Bill Nitzberg, I/O				37		
• Andrew Lumsdaine	e, Bill Saphir, and J	eff Souvres Langua	ge Bindings	38		
	, Dii Sapini, and S	on squyres, Langua	ge Dindings	39		
• Anthony Skjellum	and Arkady Kanevs	ky, Real-Time		40		
				41		
0		active participants w	who attended MPI-2 Forum	42		
meetings and are not me	entioned 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	nect Solutions, Inc.		
43	=			
44		el Computing Centre		
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	• Bill 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	• Bill Gropp, Front matter, Introduction, and Bibliography
23	• 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	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.
46 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.			20				
Argonno National Labora			21				
Argonne National Labora Bull	atory		22				
			23				
Cisco Systems, Inc.			24				
-	Cray Inc.						
The HDF Group							
Hewlett-Packard							
	IBM T.J. Watson Research						
	Indiana University Institut National da Bacharaba en Informativus et Automativus (Innia)						
Institut National de Recherche en Informatique et Automatique (Inria) Intel Corporation							
Lawrence Berkeley National Laboratory							
Lawrence Livermore Natio	-		31 32				
Los Alamos National Lab			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			44 45				
Silicon Graphics Incorpor	rated		45 46				
Sun Microsystems, Inc.	D' ' I		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 23	• Jesper Larsson Träff, Process Topologies, Info-Object and One-Sided Communications
24	• George Bosilca, Datatypes and Environmental Management
25 26	• David Solt, Process Creation and Management
27	• Bronis R. de Supinski, External Interfaces, and Profiling
28 29	• Rajeev Thakur, I/O
30 21	 Jeffrey M. Squyres, Language Bindings and MPI-2.2 Secretary
31 32	
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	
40	

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			29
Cisco Systems, Inc.			30
Cray Inc.			31
Forschungszentrum Jülich			32
Fujitsu			33
The HDF Group			34
Hewlett-Packard			35
International Business Mach	nines		36
Indiana University			37
0	che en Informatique et Auton	natique (Inria)	38
	nce & Engineering Corporatio	,	39
Intel Corporation			40
Lawrence Berkeley National	Laboratory		41
Lawrence Livermore Nation	-		42
Los Alamos National Labor			43
Mathworks			44
Mellanox Technologies			45
Microsoft			46
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 23	MPI-3.0:
24	MPI-3.0 is a significant effort to extend and modernize the MPI Standard.
25	The editors and organizers of the MPI-3.0 have been:
26	
27	• William Gropp, Steering committee, Front matter, Introduction, Groups, Contexts,
28	and Communicators, One-Sided Communications, and Bibliography
29	• Richard Graham, Steering committee, Point-to-Point Communication, Meeting Con-
30	vener, and MPI-3.0 chair
31	vener, and with 1-5.0 chair
32	• Torsten Hoefler, Collective Communication, One-Sided Communications, and Process
33	Topologies
34	· Course Desiles Detetmos and Environmental Management
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	• Droms R. de Suphiski, External interfaces and roor Support
40	• Rajeev Thakur, I/O and One-Sided Communications
41 42	• Darius Buntinas, Info Object
43	• Jeffrey M. Squyres, Language Bindings and MPI-3.0 Secretary
44	• Rolf Rabenseifner, Steering committee, Terms and Definitions, and Fortran Bindings,
45	• Iton Rabensenner, Steering committee, Terms and Demittons, and Fortran Bindings, Deprecated Functions, Annex Change-Log, and Annex Language Bindings
46 47	Depression Functions, minex change bog, and minex banguage bindings
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	
29	Tokyo Institute of Technology
30	University of Alabama at Birmingham
31	University of Chicago
32	University of Houston
33	University of Illinois at Urbana-Champaign
34	University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)
35	University of Tennessee, Knoxville
36	University of Tokyo
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	
40	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	• William Gropp, Steering committee, Front matter, Introduction, One-Sided Commu-
47	• within Gropp, Steering committee, Front matter, introduction, One-Sided Commi- nications, and Bibliography; Overall editor
48	measions, and Disnography, Overall curver

,	0	nd Definitions, and Fortran Bindings, Annex Language Bindings	$\frac{1}{2}$
• Richard L. Graham, Ste	• Richard L. Graham, Steering committee, Meeting Convener		
· · · · · ·	, , , , ,		
• 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 Balaii, Groups, C	ontexts. and Communic	cators, and External Interfaces	12
			13
• Jeff Hammond, The Info	o Object		14
• David Solt, Process Crea	ation and Management		15
• Ouingou Kazial I/O			16 17
• Quincey Koziol, I/O			18
• Kathryn Mohror, Tool S	upport		19
• Rajeev Thakur, One-Sid	• Rajeev Thakur, One-Sided Communications		
21			21
The following list includes some of the active participants who attended MPI Forum			22 23
meetings or participated in th	e e-mail discussions.		23 24
Charles Archer	Pavan Balaji	Purushotham V. Bangalore	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. Pẽna	Sreeram Potluri	42

Rolf Rabenseifner

Raghunath Raja

Christian Siebert

Kento Sato

David Solt

Nicholas Radcliffe

Craig Rasmussen

Anthony Skjellum

Jeffrey M. Squyres

Martin Schulz

Howard Pritchard

Ken Raffenetti

Davide Rossetti

Sangmin Seo

Brian Smith

43

44

45

46

1	Hari Subramoni Shinji Sumimoto Alexander Supalov	
2	Bronis R. de Supinski Sayantan Sur Masamichi Takagi	
3	Keita Teranishi Rajeev Thakur Fabian Tillier	
4	Yuichi Tsujita Geoffroy Vallée Rolf vandeVaart	
5	Akshay Venkatesh Jerome Vienne Venkat Vishwanath	
6	Anh Vo Huseyin S. Yildiz Junchao Zhang	
7	Xin Zhao	
8	The MDI Ferryr also colored and enpresisted the voluchle input from people vi	
9	The MPI Forum also acknowledges and appreciates the valuable input from people via e-mail and in person.	a
10	The following institutions supported the MPI-3.1 effort through time and travel support	•+
11	for the people listed above.	U
12		
13	Argonne National Laboratory	
14	Auburn University	
15	Cisco Systems, Inc.	
16	Cray	
17	EPCC, The University of Edinburgh	
18	ETH Zurich	
19	Forschungszentrum Jülich	
20	Fujitsu	
21	German Research School for Simulation Sciences	
22	The HDF Group	
23	International Business Machines	
24	Institut National de Recherche en Informatique et Automatique (Inria)	
25 26	Intel Corporation	
20	Jülich Aachen Research Alliance, High-Performance Computing (JARA-HPC)	
28	Kyushu University	
29	Lawrence Berkeley National Laboratory	
30	Lawrence Livermore National Laboratory Lenovo	
31		
32	Los Alamos National Laboratory Mellanox Technologies, Inc.	
33	Microsoft Corporation	
34	NEC Corporation	
35	NVIDIA Corporation	
36	Oak Ridge National Laboratory	
37	The Ohio State University	
38	RIKEN AICS	
39	Sandia National Laboratories	
40	Texas Advanced Computing Center	
41	Tokyo Institute of Technology	
42	University of Alabama at Birmingham	
43	University of Houston	
44	University of Illinois at Urbana-Champaign	
45	University of Oregon	
46	University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)	
47	University of Tennessee, Knoxville	
48	University of Tokyo	

MPI-4.0: 1
MPI-4.0 is a major update to the MPI Standard.3The editors and organizers of the MPI-4.0 have been:4
• Martin Schulz, MPI-4.0 chair, Info Object
• Wesley Bland, MPI-4.0 Secretary 7
• William Gropp, Steering committee, Front matter, Introduction, One-Sided Commu- nications, and Bibliography; Overall editor
• Rolf Rabenseifner, Steering committee, Terms and Conventions, Deprecated Func- tions, Removed Interfaces, Annex Change-Log, and Annex Language Bindings
• Daniel Holmes, Point-to-Point Communication, Sessions
• George Bosilca, Datatypes and Environmental Management
• Torsten Hoefler, Collective Communication and Process Topologies
• Pavan Balaji, Groups, Contexts, and Communicators, and External Interfaces
• Howard Pritchard, Process Creation and Management
• Anthony Skjellum, I/O
• Kathryn Mohror, Tools
• Puri Bangalore, Language Bindings
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: 29
Collective Communication, Topology, Communicators Torsten Hoefler, Andrew 30 Lumsdaine, and Anthony Skjellum 31
Fault Tolerance Wesley Bland, Aurélien Bouteiller, and Richard Graham
Hardware-Topologies Guillaume Mercier 34
Hybrid Pavan Balaji and Jim Dinan
Large Counts Jeff Hammond ³⁸
Persistence Anthony Skjellum 40
Point to Point Communication Richard Graham and Dan Holmes
42
Keniote Memory Access william Gropp and Kajeev Thakur
Semantic Terms Rolf Rabenseifner and Purushotham Bangalore 45 46
Sessions Daniel Holmes 47
Tools Kathryn Mohror and Marc-André Hermanns48

Unofficial Draft for Comment Only

The following list includes some of the active participants who attended MPI Forum $\mathbf{2}$ meetings or participated in the e-mail discussions.

3

4748

1

4 Julien Adam Abdelhalim Amer Charles Archer $\mathbf{5}$ Ammar Ahmad Awan Marc Gamell Balmana Pavan Balaji 6 Purushotham Bangalore Mohammadreza Bayatpour Jean-Baptiste Besnard 7 Claudia Blaas-Schenner Wesley Bland Gil Bloch 8 George Bosilca Aurelien Bouteiller Ben Bratu 9 Sourav Chakraborty Alexander Calvert Nicholas Chaimov 10 Mikhail Chuvelev Steffen Christgau Ching-Hsiang Chu 11 James Clark Carsten Clauss Giuseppe Congiu 12Brandon Cook James Custer Anna Daly 13 Hoang-Vu Dang James Dinan Matthew Dosanjh 14 Murali Emani Christian Engelmann Noah Evans 15Ana Gainaru Esthela Gallardo Balazs Gerofi 16 Salvatore Di Girolamo **Richard Graham** Brice Goglin 17Stanley Graves William Gropp Ryan Grant 18 Siegmar Gross Taylor Groves Yanfei Guo 19 Marc-André Hermanns Khaled Hamidouche Jeff Hammond 20Nathan Hjelm Torsten Hoefler Daniel Holmes 21Atsushi Hori Josh Hursey Ilya Ivanov 22 **Emmanuel Jeannot** Sylvain Jeaugev Julien Jaeger 23Krishna Kandalla Jithin Jose Takahiro Kawashima 24Chulho Kim Michael Knobloch Alice Koniges 25Sameer Kumar Kim Kyunghun Ignacio Laguna 26 Stefan Lankes Tonglin Li Xioyi Lu 27Kavitha Madhu Alexey Malhanov Ryan Marshall 28 William Marts Guillaume Mercier Kathryn Mohror 29 Takeshi Nanri Thomas Naughton Takafumi Nose 30 Lena Oden Steve Oyanagi Guillaume Papauré 31Ivy Peng Ignacio Laguna Peralta Antonio Peña 32 Simon Pickartz Artem Polyakov Sreeram Potluri 33 Howard Pritchard Martina Prugger Marc Pérache 34 **Rolf Rabenseifner** Nicholas Radcliffe Ken Raffenetti 35 Soren Rasmussen Hubert Ritzdorf Craig Rasmussen 36 Sergio Rivas-Gomez Davide Rossetti Martin Ruefenacht 37 Joseph Schuchart Martin Schulz Amit Ruhela 38 Sangmin Seo Sameh Sharkawi Sameer Shende 39 Min Si Anthony Skjellum Brian Smith 40David Solt Srinivas Sridharan Jeff Squyres 41 Hari Subramoni Nawrin Sultana Shinji Sumimoto 42Savantan Sur Hugo Taboada Keita Teranishi 43 **Rajeev** Thakur Keith Underwood Isaias Alberto Compres Urena 44 Geoffrov Vallee Manjunath Gorentla Venkata Akshav Venkatesh 4546

Jerome Vienne Anh Vo Justin Wozniak Junchao Zhang Dong Zhong Hui Zhou

	5
The MPI Forum also acknowledges and appreciates the valuable input from people via	6
e-mail and in person.	7
The following institutions supported the MPI-4.0 effort through time and travel support	8
for the people listed above.	9
ATOS/BULL	10
Argonne National Laboratory	11
Arm	12
Auburn University	13
Barcelona Supercomputing Center	14
CEA	15
Cisco Systems Inc.	16
Cray Inc.	17
EPCC, The University Of Edinburgh	18
ETH Zürich	19
Forschungszentrum Jülich	20
Fujitsu	21
Fulda University of Applied Sciences	22
German Research School for Simulation Sciences	23
Hewlett Packard Enterprise	24
International Business Machines	25
Institut National de Recherche en Informatique et Automatique (Inria)	26
Intel Corporation	27
Jülich Aachen Research Alliance, High-Performance Computing (JARA-HPC)	28
Jülich Supercomputing Center	29
KTH Royal Institute Of Technology	30
Kyushu University	31
Lawrence Berkeley National Laboratory	32
Lawrence Livermore National Laboratory	33
Lenovo	34
Los Alamos National Laboratory	35
Mellanox Technologies, Inc.	36
Microsoft Corporation	37
NEC Corporation	38
NVIDIA Corporation	39
Oak Ridge National Laboratory	40
PAR-TEC	41
Paratools, Inc.	42
RIKEN AICS (R-CCS as of 2017)	43
RWTH Aachen University	44
Rutgers University	45
Sandia National Laboratories	46
Silicon Graphics, Inc.	47
VSC Research Center, TU Wien	48

xxxvii

1	Technical University Of Munich
2	The HDF Group
3	The Ohio State University
4	Texas Advanced Computing Center
5	Tokyo Institute of Technology
6	University of Alabama at Birmingham
7	University of Houston
8	University of Illinois at Urbana-Champaign and the National Center for Supercomput-
9	ing Applications
10	University of Innsbruck
11	University of Oregon
12	University of Pottsdam
13	University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)
14	University of Tennessee, Chattanooga
15	University of Tennessee, Knoxville
16	University of Texas at El Paso
17	University of Tokyo
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	

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 [53], Express [14], nCUBE's Vertex [49], p4 [9, 10], and PARMACS [6, 11]. Other ¹⁶ important contributions have come from Zipcode [56, 57], Chimp [20, 21], PVM [5, 18], ¹⁷ Chameleon [28], 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 [64]. 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

6

7 8

9 10

11

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 2019 Draft Specification

The 2019 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 persistent collectives, 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,
- Collective operations,

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	• Process groups,
2 3	• Communication contexts,
4	• Process topologies,
5 6	• Environmental management and inquiry,
7 8	• The Info object,
9	• Process creation and management,
10 11	• One-sided communication,
12 13	• External interfaces,
14	• Parallel file I/O,
15 16	• Language bindings for Fortran and C,
17 18	• Tool support.
19	1.10 What Is Nat Included in the Standard?
20 21	1.12 What Is Not Included in the Standard?
22	The standard does not specify:
23 24 25	• Operations that require more operating system support than is currently standard; for example, interrupt-driven receives, remote execution, or active messages,
26	• Program construction tools,
27 28	• Debugging facilities.
29 30 31 32 33 34 35	There are many features that have been considered and not included in this standard. This happened for a number of reasons, one of which is the time constraint that was self- imposed in finishing the standard. Features that are not included can always be offered as extensions by specific implementations. Perhaps future versions of MPI will address some of these issues.
36	1.13 Organization of This Document
37 38 39 40	The following is a list of the remaining chapters in this document, along with a brief description of each.
40 41 42	• Chapter 2, MPI Terms and Conventions, explains notational terms and conventions used throughout the MPI document.
43 44 45 46	• 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.
47 48	• Chapter 4, Datatypes, defines a method to describe any data layout, e.g., an array of structures in the memory, which can be used as message send or receive buffer.

Unofficial Draft for Comment Only

 $\mathbf{6}$

- Chapter 5, Collective Communication, defines process-group collective communication operations. Well known examples of this are barrier and broadcast over a group of processes (not necessarily all the processes). With MPI-2, the semantics of collective communication was extended to include intercommunicators. It also adds two new collective operations. MPI-3 adds nonblocking collective operations.
- Chapter 6, Groups, Contexts, Communicators, and Caching, shows how groups of processes are formed and manipulated, how unique communication contexts are obtained, and how the two are bound together into a *communicator*.
- Chapter 7, Process Topologies, explains a set of utility functions meant to assist in the mapping of process groups (a linearly ordered set) to richer topological structures such as multi-dimensional grids.
- Chapter 8, MPI Environmental Management, explains how the programmer can manage and make inquiries of the current MPI environment. These functions are needed for the writing of correct, robust programs, and are especially important for the construction of highly-portable message-passing programs.
- Chapter 9, The Info Object, defines an opaque object, that is used as input in several MPI routines.
- Chapter 10, Process Initialization, Creation, and Management, defines routines that allow for creation of processes.
- Chapter 11, One-Sided Communications, defines communication routines that can be completed by a single process. These include shared-memory operations (put/get) and remote accumulate operations.
- Chapter 12, External Interfaces, defines routines designed to allow developers to layer on top of MPI. This includes generalized requests, routines that decode MPI opaque objects, and threads.
- Chapter 13, I/O, defines MPI support for parallel I/O.
- Chapter 14, 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 14.2 (Profiling Interface), which was a chapter in previous versions of MPI.
- Chapter 15, Deprecated Interfaces, describes routines that are kept for reference. However usage of these functions is discouraged, as they may be deleted in future versions of the standard.
- Chapter 16, Removed Interfaces, describes routines and constructs that have been removed from MPI. Some of these were deprecated in MPI-2, and the MPI Forum decided to remove these from the MPI-3 standard. Others of these were deprecated in MPI-3, and the MPI Forum decided to remove these from the MPI-4 standard.
- Chapter 17, Backward Incompatibilities, describes incompatibilities with previous versions of MPI.

Unofficial Draft for Comment Only

 $\overline{7}$

 $45 \\ 46$

1 • Chapter 18, Language Bindings, discusses Fortran issues, and describes language in- $\mathbf{2}$ teroperability aspects between C and Fortran. 3 The Appendices are: 4 5• Annex A, Language Bindings Summary, gives specific syntax in C and Fortran, for 6 all MPI functions, constants, and types. 7 8 • Annex B, Change-Log, summarizes some changes since the previous version of the 9 standard. 10 11 • Several Index pages show the locations of examples, constants and predefined handles, 12callback routine prototypes, and all MPI functions. 13 14MPI provides various interfaces to facilitate interoperability of distinct MPI implementations. Among these are the canonical data representation for MPI I/O and for 1516MPI_PACK_EXTERNAL and MPI_UNPACK_EXTERNAL. The definition of an actual bind-17ing of these interfaces that will enable interoperability is outside the scope of this document. 18 A separate document consists of ideas that were discussed in the MPI Forum during the 19MPI-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 2021to provide a starting point for further work. The chapters in the JOD are 22 • Chapter 2, Spawning Independent Processes, includes some elements of dynamic pro-23cess management, in particular management of processes with which the spawning 24processes do not intend to communicate, that the Forum discussed at length but 2526ultimately decided not to include in the MPI Standard. 27• Chapter 3, Threads and MPI, describes some of the expected interaction between an 28MPI implementation and a thread library in a multi-threaded environment. 29 30 • Chapter 4, Communicator ID, describes an approach to providing identifiers for com-31municators. 32 • Chapter 5, Miscellany, discusses Miscellaneous topics in the MPI JOD, in particu-33 lar single-copy routines for use in shared-memory environments and new datatype 34 constructors. 35 36 • Chapter 6, Toward a Full Fortran 90 Interface, describes an approach to providing a 37 more elaborate Fortran 90 interface. 38 39 • Chapter 7, Split Collective Communication, describes a specification for certain non-40 blocking collective operations. 41 • Chapter 8, Real-Time MPI, discusses MPI support for real time processing. 4243 44 4546 4748

Chapter 2

MPI Terms and Conventions

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

2.1 Document Notation

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

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

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

2.2 Naming Conventions

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

- 1. In C, all routines associated with a particular type of MPI object should be of the form MPI_Class_action_subset or, if no subset exists, of the form MPI_Class_action. In Fortran, all routines associated with a particular type of MPI object should be of the form MPI_CLASS_ACTION_SUBSET or, if no subset exists, of the form MPI_CLASS_ACTION.
- 2. If the routine is not associated with a class, the name should be of the form MPI_Action_subset in C and MPI_ACTION_SUBSET in Fortran.

 $45 \\ 46$

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

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

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

2.3 Procedure Specification

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

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

• OUT: the call may update the argument but does not use its input value,

• INOUT: the call may both use and update the argument.

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

28 29

30

31 32 Rationale. The definition of MPI tries to avoid, to the largest possible extent, the use of INOUT arguments, because such use is error-prone, especially for scalar arguments. (*End of rationale.*)

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

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

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

⁴⁵ Unless specified otherwise, an argument of type OUT or type INOUT cannot be aliased ⁴⁶ with any other argument passed to an MPI procedure. An example of argument aliasing in ⁴⁷ C appears below. If we define a C procedure like this,

48

1

2

3

4

5

6

7 8

9

10 11 12

13 14

15

16

17

18 19

20 21

```
void copyIntBuffer(int *pin, int *pout, int len)
    int i;
{
    for (i=0; i<len; ++i) *pout++ = *pin++;</pre>
}
then a call to it in the following code fragment has aliased arguments.
int a[10];
copyIntBuffer(a, a+3, 7);
Although the C language allows this, such usage of MPI procedures is forbidden unless
otherwise specified. Note that Fortran prohibits aliasing of arguments.
    All MPI functions are first specified in the language-independent notation. Immediately
below this, language dependent bindings follow:
   • The ISO C version of the function.
   • The Fortran version used with USE mpi_f08.
   • The Fortran version of the same function used with USE mpi or INCLUDE 'mpif.h'.
    An exception is Section 14.3 "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.
      Semantic Terms
2.4
When discussing MPI procedures the following semantic terms are used.
nonblocking A procedure is nonblocking if it may return before the associated operation
     completes, and before the user is allowed to reuse resources (such as buffers) specified
     in the call. The word complete is used with respect to operations and any associated
     requests and/or communications. An operation completes when the user is allowed
     to reuse resources, and any output buffers have been updated.
blocking A procedure is blocking if return from the procedure indicates the user is allowed
```

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

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

44

45

	12 CHAPTER 2. MPI TERMS AND CONVENTION	NS
1 2 3 4 5 6	predefined A predefined datatype is a datatype with a predefined (constant) name (su 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 a unnamed .	
7	derived A derived datatype is any datatype that is not predefined.	
8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25	portable A datatype is portable if it is a predefined datatype, or it is derived from a portable datatype using only the type constructors MPI_TYPE_CONTIGUOU MPI_TYPE_VECTOR, MPI_TYPE_INDEXED, MPI_TYPE_CREATE_INDEXED_BLOCK, MPI_TYPE_CREATE_SUBARRAY, MPI_TYPE_DUP, and MPI_TYPE_CREATE_DARRAY. Such a datatype is portable because all displacements in the datatype are in terms of extents of one predefined datatype. Therefore, if such a datatype fits a data layout in one memory, it we fit the corresponding data layout in another memory, if the same declarations we used, even if the two systems have different architectures. On the other hand, if datatype was constructed using MPI_TYPE_CREATE_HINDEXED, MPI_TYPE_CREATE_HINDEXED_BLOCK, MPI_TYPE_CREATE_HVECTOR or MPI_TYPE_CREATE_STRUCT, then the datatype contains explicit byte displacements (e.g., providing padding to meet alignment restrictions). These displacement architecture. equivalent Two datatypes are equivalent if they appear to have been created with the same datatype are in the same data layouts on another process.	JS, ble ed vill ere f a ce- nts are ent
26 27 28	sequence of calls (and arguments) and thus have the same typemap. Two equivaled datatypes do not necessarily have the same cached attributes or the same names.	
29 30	2.5 Data Types	
31 32	2.5.1 Opaque Objects	
33 34 35 36 37	MPI manages system memory that is used for buffering messages and for storing internative representations of various MPI objects such as groups, communicators, datatypes, etc. The memory is not directly accessible to the user, and objects stored there are opaque : the size and shape is not visible to the user. Opaque objects are accessed via handles , whit wist in user space. MPI procedures that operate on opaque objects are passed have	his eir ich

size and shape is not visible to the user. Opaque objects are accessed via handles, which
 exist in user space. MPI procedures that operate on opaque objects are passed handle
 arguments to access these objects. In addition to their use by MPI calls for object access,
 handles can participate in assignments and comparisons.

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

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

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

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

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

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

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

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

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

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

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

The explicit separation of handles in user space and objects in system space allows space-reclaiming and deallocation calls to be made at appropriate points in the user

Unofficial Draft for Comment Only

 24

 31

 $46 \\ 47$

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

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

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

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

2.5.2 Array Arguments

An MPI call may need an argument that is an array of opaque objects, or an array of 35 handles. The array-of-handles is a regular array with entries that are handles to objects 36 of the same type in consecutive locations in the array. Whenever such an array is used, 37 an additional len argument is required to indicate the number of valid entries (unless this 38 number can be derived otherwise). The valid entries are at the beginning of the array; 39 len indicates how many of them there are, and need not be the size of the entire array. 40 The same approach is followed for other array arguments. In some cases NULL handles are 41 considered valid entries. When a NULL argument is desired for an array of statuses, one 42uses MPI_STATUSES_IGNORE. 43

44 45

2.5.3 State

 $^{46}_{47}$ MPI procedures use at various places arguments with *state* types. The values of such a data type are all identified by names, and no operation is defined on them. For example, the

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

MPI_TYPE_CREATE_SUBARRAY routine has a state argument order with values MPI_ORDER_C and MPI_ORDER_FORTRAN.

2.5.4 Named Constants

MPI procedures sometimes assign a special meaning to a special value of a basic type argument; e.g., tag is an integer-valued argument of point-to-point communication operations, with a special wild-card value, MPI_ANY_TAG. Such arguments will have a range of regular values, which is a proper subrange of the range of values of the corresponding basic type; special values (such as MPI_ANY_TAG) will be outside the regular range. The range of regular values, such as tag, can be queried using environmental inquiry functions, see Chapter 8. The range of other values, such as source, depends on values given by other MPI routines (in the case of **source** it is the communicator size).

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

All named constants, with the exceptions noted below for Fortran, can be used in initialization expressions or assignments, but not necessarily in array declarations or as labels in C switch or Fortran select/case statements. This implies named constants to be link-time but not necessarily compile-time constants. The named constants listed below are required to be compile-time constants in both C and Fortran. These constants do not change values during execution. Opaque objects accessed by constant handles are 20defined and do not change value between MPI initialization (MPI_INIT) and MPI completion 21(MPI_FINALIZE). The handles themselves are constants and can be also used in initialization 22expressions or assignments. 23

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

are:	26
MPI_MAX_PROCESSOR_NAME	27
MPI_MAX_LIBRARY_VERSION_STRING	28
MPI_MAX_ERROR_STRING	29
MPI_MAX_DATAREP_STRING	30
MPI_MAX_INFO_KEY	31
MPI_MAX_INFO_VAL	32
MPI_MAX_OBJECT_NAME	33
MPI_MAX_PORT_NAME	34
MPI_VERSION	35
MPI_SUBVERSION	36
MPI_STATUS_SIZE (Fortran only)	37
MPI_ADDRESS_KIND (Fortran only)	38
MPI_COUNT_KIND (Fortran only)	39
MPI_INTEGER_KIND (Fortran only)	40
MPI_OFFSET_KIND (Fortran only)	41
MPI_SUBARRAYS_SUPPORTED (Fortran only)	42
MPI_ASYNC_PROTECTS_NONBLOCKING (Fortran only)	43
The constants that cannot be used in initialization expressions or assignments in Fo	- 44
tran are as follows:	45
MPI_BOTTOM	46
MPI_STATUS_IGNORE	47
MPI_STATUSES_IGNORE	48

1 2

3 4

5

6

 $\overline{7}$

8

9

10

11

12

13

14

15

16

17

18

19

24

1	MPI_ERRCODES_IGNORE
2	MPI_IN_PLACE
3	MPI_ARGV_NULL
4	MPI_ARGVS_NULL
5	MPI_UNWEIGHTED
6	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

¹⁹ MPI functions sometimes use arguments with a *choice* (or union) data type. Distinct calls to ²⁰ the same routine may pass by reference actual arguments of different types. The mechanism ²¹ for providing such arguments will differ from language to language. For Fortran with the ²² include file mpif.h or the mpi module, the document uses <type> to represent a choice ²³ variable; with the Fortran mpi_f08 module, such arguments are declared with the Fortran ²⁴ 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 18.1.1. (End of advice to implementors.)

30 31 32

33

34

35

25

26

27

28

29

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=

36 MPI_ADDRESS_KIND) in Fortran. These types must have the same width and encode address 37 values in the same manner such that address values in one language may be passed directly 38 to another language without conversion. There is the MPI constant MPI_BOTTOM to in-39 dicate the start of the address range. For retrieving absolute addresses or any calculation 40 with absolute addresses, one should use the routines and functions provided in Section 4.1.5. 41 Section 4.1.12 provides additional rules for the correct use of absolute addresses. For ex-42pressions with relative displacements or other usage without absolute addresses, intrinsic 43 operators (e.g., +, -, *) can be used. 44

44 45

46

2.5.7 File Offsets

⁴⁷ For I/O there is a need to give the size, displacement, and offset into a file. These quantities ⁴⁸ can easily be larger than 32 bits which can be the default size of a Fortran integer. To

Unofficial Draft for Comment Only

7

8

9

10

11

12

13

14

15

16 17

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) in Fortran.

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

2.6 Language Binding

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

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

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

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

2.6.1 Deprecated and Removed Interfaces

A number of chapters refer to deprecated or replaced MPI constructs. These are constructs that continue to be part of the MPI standard, as documented in Chapter 15, but that users

Unofficial Draft for Comment Only

 $\overline{7}$

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 deprecated constructs are now removed, as documented in Chapter 16.
 ¹⁰ They may still be provided by an implementation for backwards compatibility, but are not required.

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

- 15
- ¹⁶ 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 8 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 18.2.9.

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

Array arguments are indexed from one.

The older MPI Fortran bindings (mpif.h and use mpi) are inconsistent with the Fortran standard in several respects. These inconsistencies, such as register optimization problems, have implications for user codes that are discussed in detail in Section 18.1.16.

41 42 43

36

37

38

39

40

2.6.3 C Binding Issues

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

Unofficial Draft for Comment Only

2.6. LANGUAGE BINDING

Deprecated or removed	deprecated	removed	Replacement		
construct	since	since	22		
MPI_ADDRESS	MPI-2.0	MPI-3.0	MPL CET 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 4		
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 ⁸		
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 1		
MPI_ERRHANDLER_GET	MPI-2.0	MPI-3.0	MPI_COMM_GET_ERRHANDLER 1		
MPI_ERRHANDLER_SET	MPI-2.0	MPI-3.0	MPL 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 1		
MPI_KEYVAL_FREE	MPI-2.0		MPI_COMM_FREE_KEYVAL 1		
MPI_DUP_FN ³	MPI-2.0		MPI_COMM_DUP_FN ³		
MPI_NULL_COPY_FN ³	MPI-2.0		MPL COMM NULL COPY EN3		
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 2		
MPI_ATTR_PUT	MPI-2.0		MPI_COMM_SET_ATTR ²		
MPI_COMBINER_HVECTOR_INTEGER ⁴	-	MPI-3.0	MPI_COMBINER_HVECTOR ⁴ 2		
MPI_COMBINER_HINDEXED_INTEGER ⁴	-	MPI-3.0	MPI_COMBINER_HINDEXED ⁴ 2		
MPI_COMBINER_STRUCT_INTEGER ⁴	-	MPI-3.0	MPI_COMBINER_STRUCT ⁴		
MPI::	MPI-2.2	MPI-3.0	C language binding		
MPI_CANCEL for send requests	MPI-3.2		no direct replacement 2		
MPI_INFO_GET	MPI-4.0		MPI_INFO_GET_STRING 2		
MPI_INFO_GET_VALUELEN	MPI-4.0		MPI_INFO_GET_STRING 2		
MPI_T_ERR_INVALID_ITEM	MPI-3.2		MPI_T_ERR_INVALID_INDEX 2		
MPI_SIZEOF	MPI-4.0)	storage size() ⁵		
¹ Predefined datatype.			3		
² Callback prototype definition.			3		
³ Predefined callback routine.			3		
⁴ Constant.			3		
5 Fortran intrinsic. It returns the size in bits instead of bytes. $_{3}$					
Other entries are regular MPI routines.					
			3		
			3		
Table 2.1: D	eprecated a	nd Remov	red constructs 3		

letters are either upper or lower case. To support the profiling interface, programs must not declare functions with names beginning with any prefix of the form PMPI_, where any

of the letters are either upper or lower case.

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

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

Type declarations are provided for handles to each category of opaque objects. Array arguments are indexed from zero.

Unofficial Draft for Comment Only

37 38 39

40

41

42

43

44

45

46

47

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

Choice arguments are pointers of type void*.

2.6.4 Functions and Macros

An implementation is allowed to implement MPI_WTIME, PMPI_WTIME, MPI_WTICK, PMPI_WTICK, MPI_AINT_ADD, PMPI_AINT_ADD, MPI_AINT_DIFF, PMPI_AINT_DIFF, and the handle-conversion functions (MPI_Group_f2c, etc.) in Section 18.2.4, and no others, as macros in C.

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

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

2.7 Processes

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

This document specifies the behavior of a parallel program assuming that only MPI 24 calls are used. The interaction of an MPI program with other possible means of commu-25nication, I/O, and process management is not specified. Unless otherwise stated in the 26specification of the standard, MPI places no requirements on the result of its interaction 27with external mechanisms that provide similar or equivalent functionality. This includes, 28but is not limited to, interactions with external mechanisms for process control, shared and 29 remote memory access, file system access and control, interprocess communication, process 30 signaling, and terminal I/O. High quality implementations should strive to make the results 31 of such interactions intuitive to users, and attempt to document restrictions where deemed 32 necessary. 33

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

The interaction of MPI and threads is defined in Section 10.6.

39 40 41

42

34

35

36

37 38

2.8 Error Handling

MPI provides the user with reliable message transmission. A message sent is always received correctly, and the user does not need to check for transmission errors, time-outs, or other error conditions. In other words, MPI does not provide mechanisms for dealing with transmission failures in the communication system. If the MPI implementation is built on an unreliable underlying mechanism, then it is the job of the implementor of the MPI subsystem to insulate the user from this unreliability, and to reflect only unrecoverable

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4 5

6

7

8

9

10 11

12

13 14

15

16 17

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.

In C and Fortran, almost all MPI calls return a code that indicates successful completion of the operation. Whenever possible, MPI calls return an error code if an error occurred during the call. By default, an error detected during the execution of the MPI library causes the parallel computation to abort, except for file operations. However, MPI provides mechanisms for users to change this default and to handle recoverable errors. The user may 20specify that no error is fatal, and handle error codes returned by MPI calls by himself or 21herself. Also, the user may provide his or her own error-handling routines, which will be invoked whenever an MPI call returns abnormally. The MPI error handling facilities are 2223described in Section 8.3.

Several factors limit the ability of MPI calls to return with meaningful error codes when an error occurs. MPI may not be able to detect some errors; other errors may be too expensive to detect in normal execution mode: finally some errors may be "catastrophic" and may prevent MPI from returning control to the caller. On the other hand, some errors 27may 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 29errors will be raised on the communicator MPI_COMM_SELF. When MPI_COMM_SELF is not 30 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 10.8.4).

33 An example of such a case arises because of the nature of asynchronous communications: 34MPI 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 35 error exception to be raised. If there is a subsequent call that relates to the same operation 36 37 (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 38 39 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 40 41 receiver in a send with the ready mode).

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

Implementations may go beyond this document in supporting in a meaningful manner MPI calls that are defined here to be erroneous. For example, MPI specifies strict type

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

 $\mathbf{6}$ $\overline{7}$

8 9

10

11

12

13

1415

16

17

18

19

 24 25

26

28

31

32

42

43

44

4546

47

matching rules between matching send and receive operations: it is erroneous to send a
 floating point variable and receive an integer. Implementations may go beyond these type
 matching rules, and provide automatic type conversion in such situations. It will be helpful
 to generate warnings for such non-conforming behavior.

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

2.9 Implementation Issues

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

14 15 16

39

43

5

 $\frac{6}{7}$

8 9

10

11

12

13

2.9.1 Independence of Basic Runtime Routines

¹⁷ MPI programs require that library routines that are part of the basic language environment
 ¹⁸ (such as write in Fortran and printf and malloc in ISO C) and are executed after
 ¹⁹ MPI_INIT and before MPI_FINALIZE operate independently and that their *completion* is

²⁰ 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).

```
25
26 int rank;
```

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

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

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

```
30 MPI_Finalize();
```

 $\frac{31}{32}$ 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).

```
<sup>37</sup> MPI_Comm_rank(MPI_COMM_WORLD, &rank);
<sup>38</sup> 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).

44 2.9.2 Interaction with Signals

⁴⁵ MPI does not specify the interaction of processes with signals and does not require that MPI ⁴⁶ be signal safe. The implementation may reserve some signals for its own use. It is required ⁴⁷ that the implementation document which signals it uses, and it is strongly recommended

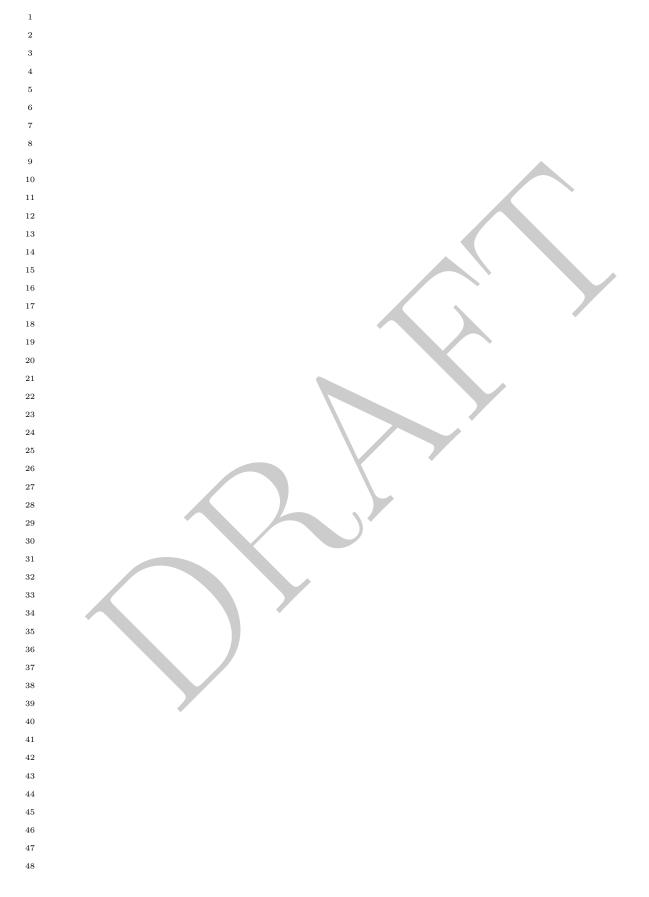
Unofficial Draft for Comment Only

that it not use SIGALRM, SIGFPE, or SIGIO. Implementations may also prohibit the use of MPI calls from within signal handlers.

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

2.10 Examples

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



Chapter 3

Point-to-Point Communication

3.1Introduction

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

```
20
#include "mpi.h"
                                                                                   21
int main(int argc, char *argv[])
                                                                                   22
{
                                                                                   23
  char message[20];
  int myrank;
 MPI_Status status;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  if (myrank == 0)
                       /* code for process zero */
                                                                                   29
  {
      strcpy(message,"Hello, there");
      MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
  }
                                                                                   33
  else if (myrank == 1)
                         /* code for process one */
                                                                                   34
  {
                                                                                   35
      MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
                                                                                   36
      printf("received :%s:\n", message);
                                                                                   37
  }
 MPI_Finalize();
  return 0;
}
```

42In this example, process zero (myrank = 0) sends a message to process one using the send operation MPI_SEND. The operation specifies a send buffer in the sender memory 4344from which the message data is taken. In the example above, the send buffer consists of the storage containing the variable **message** in the memory of process zero. The location, size and type of the send buffer are specified by the first three parameters of the send operation. The message sent will contain the 13 characters of this variable. In addition, the send operation associates an **envelope** with the message. This envelope specifies the

1 $\mathbf{2}$ 3

5

6

1516

17

18

19

24

25

26

27

28

30

31

32

38

39

40

41

4546

47

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

¹¹ The next sections describe the blocking send and receive operations. We discuss send, ¹² receive, blocking communication semantics, type matching requirements, type conversion in ¹³ heterogeneous environments, and more general communication modes. Nonblocking com-¹⁴ munication is addressed next, followed by probing and canceling a message, channel-like ¹⁵ constructs and send-receive operations, ending with a description of the "dummy" process, ¹⁶ MPI_PROC_NULL.

3.2 Blocking Send and Receive Operations

3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

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

26	IN	buf	initial address of send buffer (choice)
27 28	IN	count	number of elements in send buffer (non-negative integer)
29 30	IN	datatype	datatype of each send buffer element (handle)
31	IN	dest	rank of destination (integer)
32 33	IN	tag	message tag (integer)
34	IN	comm	communicator (handle)

36 C binding

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

```
<sup>39</sup> 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
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
    Fortran binding
```

```
<sup>48</sup> MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
```

17 18

19 20

21

22 23 24

25

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

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

3.2.2 Message Data

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

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

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

Table 3.1: Predefined MPI datatypes corresponding to Fortran datatypes

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

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

MPI requires support of these datatypes, which match the basic datatypes of Fortran and ISO C. Additional MPI datatypes should be provided if the host language has additional data types¹: MPI_DOUBLE_COMPLEX for double precision complex in Fortran declared to be of type DOUBLE COMPLEX; MPI_REAL2, MPI_REAL4, and MPI_REAL8 for Fortran reals, declared to be of type REAL*2, REAL*4 and REAL*8, respectively; MPI_INTEGER1, MPI_INTEGER2, and MPI_INTEGER4 for Fortran integers, declared to be of type INTEGER*1, INTEGER*2, and INTEGER*4, respectively; etc.

Rationale. One goal of the design is to allow for MPI to be implemented as a library, with no need for additional preprocessing or compilation. Thus, one cannot

 31

¹These types, such as DOUBLE COMPLEX and INTEGER*4, are not specified by any Fortran standard but are extensions commonly accepted by Fortran compilers.

1	MPI datatype	C datatype
2	MPI_CHAR	char
3	_	(treated as printable character)
L	MPI_SHORT	signed short int
	MPI_INT	signed int
	MPI_LONG	signed long int
	MPI_LONG_LONG_INT	signed long long int
	MPI_LONG_LONG (as a synonym)	signed long long int
	MPI_SIGNED_CHAR	signed char
)		(treated as integral value)
	MPI_UNSIGNED_CHAR	unsigned char
:		(treated as integral value)
	MPI_UNSIGNED_SHORT	unsigned short int
	MPI_UNSIGNED	unsigned int
	MPI_UNSIGNED_LONG	unsigned long int
	MPI_UNSIGNED_LONG_LONG	unsigned long long int
;		float
		double
)	MPI_LONG_DOUBLE	long double
	MPI_WCHAR	wchar_t
		(defined in <stddef.h>)</stddef.h>
2		(treated as printable character)
3	MPI_C_BOOL	_Bool
	MPI_INT8_T	int8_t
5	MPI_INT16_T	int16_t
5	MPI_INT32_T	int32_t
	MPI_INT64_T	int64_t
;	MPI_UINT8_T	uint8_t
)	MPI_UINT16_T	uint16_t
)	MPI_UINT32_T	uint32_t
	MPI_UINT64_T	uint64_t
:	MPI_C_COMPLEX	float _Complex
	MPI_C_FLOAT_COMPLEX (as a synonym)	float _Complex
L	MPI_C_DOUBLE_COMPLEX	double _Complex
	MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
;	MPI_BYTE	5 · · · · · · · · · · · · · · · · · · ·
,	MPI_PACKED	
3		<u> </u>
)		
	Table $\overline{3.2}$: Predefined MPI datatypes co	prresponding to C datatypes
:	assume that a communication call has informa	ation on the datatype of variables in
	communication buffer; this information mus	
, L	,	** * * 0
•	The need for such datatype information will	become crear in Section 5.5.2. (En
;	rationale.)	
	The datatypes MPI_AINT, MPI_OFFSET, and	MPL COUNT correspond to the N
	The datatypes MPLAINT, MPLOFFSET, and	

defined C types MPI_Aint, MPI_Offset, and MPI_Count and their Fortran equivalents

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

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 18.2.6 and 18.2.10 on page 761 and 768 for information on interlanguage communication with these types.

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

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

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

	30
source	31
destination	32
tag	33
communicator	34

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

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

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

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

A communicator specifies the communication context for a communication operation. 4546Each communication context provides a separate "communication universe": messages are 47always received within the context they were sent, and messages sent in different contexts 48 do not interfere.

Unofficial Draft for Comment Only

7 8

9

10

11

12

13

23 24 25

26

27

28

3536

37

38

39

40 41

42

1 The communicator also specifies the set of processes that share this communication $\mathbf{2}$ context. This **process group** is ordered and processes are identified by their rank within 3 this group. Thus, the range of valid values for dest is $0, \ldots, n-1 \cup \{\text{MPI}_{PROC}_{NULL}\}$, where 4 n is the number of processes in the group. (If the communicator is an inter-communicator, 5then destinations are identified by their rank in the remote group. See Chapter 6.) 6 When using the World Model (see Section 10.1), a predefined communicator $\overline{7}$ MPI_COMM_WORLD is provided by MPI. It allows communication with all processes that 8 are accessible after MPI initialization and processes are identified by their rank in the group 9 of MPI_COMM_WORLD. 10 Users that are comfortable with the notion of a flat name space 11 Advice to users. for processes, and a single communication context, as offered by most existing com-12munication libraries, need only use the World Model for MPI initialization, and the 13

predefined variable MPI_COMM_WORLD as the comm argument. This will allow com-14munication with all the processes available at initialization time. 15

16Users may define new communicators, as explained in Chapter 6. Communicators 17 provide an important encapsulation mechanism for libraries and modules. They allow 18 modules to have their own disjoint communication universe and their own process 19 numbering scheme. (End of advice to users.) 20

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

3.2.4 Blocking Receive

Fortran 2008 binding

The syntax of the blocking receive operation is given below.

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

32		(bui, count, datatype, source,	lag, comm, statusj
33	OUT	buf	initial address of receive buffer (choice)
34 35	IN	count	number of elements in receive buffer (non-negative integer)
36 37	IN	datatype	datatype of each receive buffer element (handle)
38	IN	source	rank of source or MPI_ANY_SOURCE (integer)
39	IN	tag	message tag or MPI_ANY_TAG (integer)
40 41	IN	comm	communicator (handle)
42	OUT	status	status object (Status)
43			
44	C binding	r	
45	int MPI_R	ecv(void *buf, int count,	MPI_Datatype datatype, int source,
46		int tag, MPI_Comm con	nm, MPI_Status *status)

21

22

23

 24

25

2627

28

29 30 31

47

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

Fortran binding

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

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

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

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

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

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

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

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

Unofficial Draft for Comment Only

 $\mathbf{2}$

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.

¹⁸ 3.2.5 Return Status

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

⁴¹ MPI_ERROR, and TYPE(MPI_Status) are defined to allow conversion between both status ⁴² representations. Conversion routines are provided in Section 18.2.5.

- Rationale. The Fortran TYPE(MPI_Status) is defined as a BIND(C) derived type so
 that it can be used at any location where the status integer array representation can
 be used, e.g., in user defined common blocks. (End of rationale.)
- Rationale. It is allowed to have the same name (e.g., MPI_SOURCE) defined as a constant (e.g., Fortran parameter) and as a field of a derived type. (*End of rationale.*)

Unofficial Draft for Comment Only

6

 $\overline{7}$

8

9

10

11

12

13

14 15

16

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

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

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

MPI_GET_COUNT(status, datatype, count)

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

C binding

Fortran 2008 binding

MP1_Get_count(status, datatype, count, ierror)	
TYPE(MPI_Status), INTENT(IN) :: status	
TYPE(MPI_Datatype), INTENT(IN) :: datatype	
INTEGER, INTENT(OUT) :: count	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	

Fortran binding

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

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

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

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\overline{7}$

 24

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

The value returned as the count argument of MPI_GET_COUNT for a datatype of length zero where zero bytes have been transferred is zero. If the number of bytes transferred is 9 greater than zero, MPI_UNDEFINED is returned. 10

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.

26 Passing MPI_STATUS_IGNORE for Status 3.2.6 27

28Every call to MPI_RECV includes a status argument, wherein the system can return details 29about the message received. There are also a number of other MPI calls where status 30 is returned. An object of type MPI_Status is not an MPI opaque object; its structure 31 is declared in mpi.h and mpif.h, and it exists in the user's program. In many cases, 32 application programs are constructed so that it is unnecessary for them to examine the 33 status fields. In these cases, it is a waste for the user to allocate a status object, and it is 34particularly wasteful for the MPI implementation to fill in fields in this object.

35 To cope with this problem, there are two predefined constants, MPI_STATUS_IGNORE 36 and MPI_STATUSES_IGNORE, which when passed to a receive, probe, wait, or test function, 37 inform the implementation that the status fields are not to be filled in. Note that

38 MPI_STATUS_IGNORE is not a special type of MPI_Status object; rather, it is a special 39 value for the argument. In C one would expect it to be NULL, not the address of a special 40MPI_Status.

41 MPI_STATUS_IGNORE, and the array version MPI_STATUSES_IGNORE, can be used every-42where a status argument is passed to a receive, wait, or test function. MPI_STATUS_IGNORE 43cannot be used when status is an IN argument. Note that in Fortran MPI_STATUS_IGNORE 44and MPI_STATUSES_IGNORE are objects like MPI_BOTTOM (not usable for initialization or 45assignment). See Section 2.5.4.

46 In general, this optimization can apply to all functions for which status or an array of 47statuses is an OUT argument. Note that this converts status into an INOUT argument. The 48functions that can be passed MPI_STATUS_IGNORE are all the various forms of MPI_RECV,

Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8

11

12

13

14

15

1617

18

19

20

2122

23

 24

MPI_PROBE, MPI_TEST, and MPI_WAIT, as well as MPI_REQUEST_GET_STATUS. When an array is passed, as in the MPI_{TEST|WAIT}{ALL|SOME} functions, a separate constant, MPI_STATUSES_IGNORE, is passed for the array argument. It is possible for an MPI function to return MPI_ERR_IN_STATUS even when MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE has been passed to that function.

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

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

3.2.7 Send-Receive

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

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

1 2	MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)							
$\frac{3}{4}$	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	type of elements in send buffer (handle)					
8 9	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	IN	recvtype	type of elements 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	status	status object (Status)					
21								
22 23	C bindin	-						
24	int MPI_S		<pre>buf, int sendcount, MPI_Datatype sendtype, g, void *recvbuf, int recvcount,</pre>					
25			e, int source, int recvtag, MPI_Comm comm,					
26 27		MPI_Status *status)						
28	Fortran 2	2008 binding						
29	MPI_Sendi		sendtype, dest, sendtag, recvbuf,					
30	TVDF	recvcount, recvtype, (*), DIMENSION(), INTEN	source, recvtag, comm, status, ierror)					
31 32			unt, dest, sendtag, recvcount, source,					
33		recvtag	-					
34		(MPI_Datatype), INTENT(IN						
35 36		(*), DIMENSION() :: rec (MPI_Comm), INTENT(IN) ::						
37		(MPI_Status) :: status						
38	INTEC	GER, OPTIONAL, INTENT(OUT) :: ierror					
39	Fortran k	binding						
40 41	MPI_SENDF		SENDTYPE, DEST, SENDTAG, RECVBUF,					
42	(+····	RECVCOUNT, RECVTYPE, >> SENDBUF(*), RECVBUF(*)	SOURCE, RECVTAG, COMM, STATUS, IERROR)					
43	• 1		DEST, SENDTAG, RECVCOUNT, RECVTYPE,					
44			MM, STATUS(MPI_STATUS_SIZE), IERROR					
45 46	Execu	te a blocking send and receiv	ve operation. Both send and receive use the same					
47	communicator, but possibly different tags. The send buffer and receive buffers must be							
48	disjoint, and may have different lengths and datatypes.							

3.2. BLOCKING SEND AND RECEIVE OPERATIONS

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

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

	514145)		(
INOUT	buf	initial address of send and receive buffer (choice)	8
IN	count	number of elements in send and receive buffer	9
		(non-negative integer)	10 11
IN	datatype	type of elements in send and receive buffer (handle)	11
IN	dest	rank of destination (integer)	13
IN	sendtag	send message tag (integer)	14
IN	source	rank of source or MPI_ANY_SOURCE (integer)	15 16
IN	recvtag	receive message tag or MPI_ANY_TAG (integer)	10
	5		18
IN	comm	communicator (handle)	19
OUT	status	status object (Status)	20

C binding

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

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

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

3.3 Data Type Matching and Data Conversion

3.3.1 Type Matching Rules

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

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

3. Data is pulled from the incoming message and disassembled into the receive buffer.

¹¹ Type matching has to be observed at each of these three phases: The type of each ¹² variable in the sender buffer has to match the type specified for that entry by the send ¹³ operation; the type specified by the send operation has to match the type specified by the ¹⁴ receive operation; and the type of each variable in the receive buffer has to match the type ¹⁵ specified for that entry by the receive operation. A program that fails to observe these three ¹⁶ rules is erroneous.

¹⁷ To define type matching more precisely, we need to deal with two issues: matching of ¹⁸ types of the host language with types specified in communication operations; and matching ¹⁹ of types at sender and receiver.

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

 24 The type of a variable in a host program matches the type specified in the commu-25nication operation if the datatype name used by that operation corresponds to the basic 26type of the host program variable. For example, an entry with type name MPI_INTEGER 27matches a Fortran variable of type INTEGER. A table giving this correspondence for Fortran 28and C appears in Section 3.2.2. There are two exceptions to this last rule: an entry with 29type name MPI_BYTE or MPI_PACKED can be used to match any byte of storage (on a 30 byte-addressable machine), irrespective of the datatype of the variable that contains this 31 byte. The type MPI_PACKED is used to send data that has been explicitly packed, or receive data that will be explicitly unpacked, see Section 4.2. The type MPI_BYTE allows 32 33 one to transfer the binary value of a byte in memory unchanged.

34 35 36

37

38 39

40

41

42

43 44

45

46

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

- Communication of typed values (e.g., with datatype different from MPI_BYTE), where the datatypes of the corresponding entries in the sender program, in the send call, in the receive call and in the receiver program must all match.
- Communication of untyped values (e.g., of datatype MPI_BYTE), where both sender and receiver use the datatype MPI_BYTE. In this case, there are no requirements on the types of the corresponding entries in the sender and the receiver programs, nor is it required that they be the same.
- Communication involving packed data, where MPI_PACKED is used.
- The following examples illustrate the first two cases.
- $_{48}^{47}$ **Example 3.1** Sender and receiver specify matching types.

Unofficial Draft for Comment Only

1 2

3 4

5 6

7

8 9

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

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

Example 3.2 Sender and receiver do not specify matching types.

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

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

Example 3.3 Sender and receiver specify communication of untyped values.

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

IF (rank .EQ. 0) THEN

CALL MPI_SEND(a(1), 40, MPI_BYTE, 1, tag, comm, ierr)

ELSE IF (rank .EQ. 1) THEN

CALL MPI_RECV(b(1), 60, MPI_BYTE, 0, tag, comm, status, ierr)

END IF
```

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

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

Type MPI_CHARACTER

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

Unofficial Draft for Comment Only

 $45 \\ 46$

1 Example 3.4 Transfer of Fortran CHARACTERs. $\mathbf{2}$ 3 CHARACTER*10 a 4 CHARACTER*10 b 56 CALL MPI_COMM_RANK(comm, rank, ierr) IF (rank .EQ. 0) THEN 7 CALL MPI_SEND(a, 5, MPI_CHARACTER, 1, tag, comm, ierr) 8 9 ELSE IF (rank .EQ. 1) THEN CALL MPI_RECV(b(6:10), 5, MPI_CHARACTER, 0, tag, comm, status, ierr) 10 11 END IF 12The last five characters of string **b** at process 1 are replaced by the first five characters 13 of string a at process 0. 1415Rationale. The alternative choice would be for MPI_CHARACTER to match a char-16acter of arbitrary length. This runs into problems. 1718 A Fortran character variable is a constant length string, with no special termina-19 tion symbol. There is no fixed convention on how to represent characters, and how 20to store their length. Some compilers pass a character argument to a routine as a 21pair of arguments, one holding the address of the string and the other holding the 22 length of string. Consider the case of an MPI communication call that is passed a 23communication buffer with type defined by a derived datatype (Section 4.1). If this 24communicator buffer contains variables of type CHARACTER then the information on 25their length will not be passed to the MPI routine. 26This problem forces us to provide explicit information on character length with the 27MPI call. One could add a length parameter to the type MPI_CHARACTER, but this 28does not add much convenience and the same functionality can be achieved by defining 29 a suitable derived datatype. (End of rationale.) 30 31Advice to implementors. Some compilers pass Fortran CHARACTER arguments as a 32 structure with a length and a pointer to the actual string. In such an environment, 33 the MPI call needs to dereference the pointer in order to reach the string. (End of 34 advice to implementors.) 35 36 Data Conversion 3.3.2 37

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

41 42

43

44

45

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

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

The type matching rules imply that MPI communication never entails type conversion. On the other hand, MPI requires that a representation conversion be performed when a typed value is transferred across environments that use different representations for the

datatype of this value. MPI does not specify rules for representation conversion. Such conversion is expected to preserve integer, logical and character values, and to convert a floating point value to the nearest value that can be represented on the target system.

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

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

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

Consider the three examples, 3.1-3.3. The first program is correct, assuming that a and b are REAL arrays of size ≥ 10 . If the sender and receiver execute in different environments, then the ten real values that are fetched from the send buffer will be converted to the representation for reals on the receiver site before they are stored in the receive buffer. While the number of real elements fetched from the send buffer equal the number of real elements stored in the receive buffer, the number of bytes stored need not equal the number of bytes loaded. For example, the sender may use a four byte representation and the receiver an eight byte representation for reals.

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

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

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

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

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

MPI requires support for inter-language communication, 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 18.2.

 $\mathbf{2}$

 $\mathbf{5}$

 31

 $44 \\ 45$

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. 6

Message buffering decouples the send and receive operations. A blocking send can com-7 plete as soon as the message was buffered, even if no matching receive has been executed by 8 the receiver. On the other hand, message buffering can be expensive, as it entails additional 9 memory-to-memory copying, and it requires the allocation of memory for buffering. MPI 10 offers the choice of several communication modes that allow one to control the choice of the 11 communication protocol. 12

The send call described in Section 3.2.1 uses the standard communication mode. In 13 this mode, it is up to MPI to decide whether outgoing messages will be buffered. MPI may 14buffer outgoing messages. In such a case, the send call may complete before a matching 15receive is invoked. On the other hand, buffer space may be unavailable, or MPI may choose 16not to buffer outgoing messages, for performance reasons. In this case, the send call will 17not complete until a matching receive has been posted, and the data has been moved to the 18 receiver. 19

Thus, a send in standard mode can be started whether or not a matching receive has 20been posted. It may complete before a matching receive is posted. The standard mode send 21is non-local: successful completion of the send operation may depend on the occurrence of 22 a matching receive. 23

 24 25

26

27

28

29

30

 31

32

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.)

33 34

35

There are three additional communication modes.

A **buffered** mode send operation can be started whether or not a matching receive 36 has been posted. It may complete before a matching receive is posted. However, unlike the 37 standard send, this operation is *local*, and its completion does not depend on the occurrence 38 of a matching receive. Thus, if a send is executed and no matching receive is posted, then 39 MPI must buffer the outgoing message, so as to allow the send call to complete. An error will 40 occur if there is insufficient buffer space. The amount of available buffer space is controlled 41 by the user — see Section 3.6. Buffer allocation by the user may be required for the buffered 42mode to be effective. 43

A send that uses the **synchronous** mode can be started whether or not a matching 44receive was posted. However, the send will complete successfully only if a matching receive is 45posted, and the receive operation has started to receive the message sent by the synchronous 46 send. Thus, the completion of a synchronous send not only indicates that the send buffer 47can be reused, but it also indicates that the receiver has reached a certain point in its 48

1 2

> 3 4

execution, namely that it has started executing the matching receive. If both sends and receives are blocking operations then the use of the synchronous mode provides synchronous communication semantics: a communication does not complete at either end before both processes rendezvous at the communication. A send executed in this mode is non-local.

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

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

MPI_BSEND(buf, count, datatype, dest, tag, comm)				
IN	buf initial address of send buffer (choice)			
IN	count	number of elements in send buffer (non-negative	23 24	
		integer)	25	
IN	datatype	datatype of each send buffer element (handle)	26	
IN	dest	rank of destination (integer)	27	
IN	tag	message tag (integer)	28	
			29	
IN	comm	communicator (handle)	30	
			31	
C binding	g		32	
int MPI_E	send(const void *buf, int	count, MPI_Datatype datatype, int dest,	33	
	int tag, MPI_Comm con	nm)	34	
			35	
Fortran 2008 binding				

Fortran 2008 binding MPI_Bsend(buf, count, datatype, dest, tag, comm, ierror) TYPE(*), DIMENSION(...), INTENT(IN) :: buf INTEGER, INTENT(IN) :: count, dest, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

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

Send in buffered mode.

1

2

3

4

5

6 $\overline{7}$

8

9 10

11

12

13 14

15

16

17

18

19 20

36

37

38

39

40

41

42

43

44

45

4647

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

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

Fortran binding

Send in ready mode.

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

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

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

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

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

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

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

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

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

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

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

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

In a multithreaded environment, the execution of a blocking communication should block only the executing thread, allowing the thread scheduler to de-schedule this thread and schedule another thread for execution. (*End of advice to implementors.*) $\frac{45}{46}$

Unofficial Draft for Comment Only

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

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.

6 **Order** Messages are *non-overtaking*: If a sender sends two messages in succession to the 7same destination, and both match the same receive, then this operation cannot receive the 8 second message if the first one is still pending. If a receiver posts two receives in succession, 9 and both match the same message, then the second receive operation cannot be satisfied 10 by this message, if the first one is still pending. This requirement facilitates matching of 11sends to receives. It guarantees that message-passing code is deterministic, if processes are single-threaded and the wildcard MPI_ANY_SOURCE is not used in receives. (Some of the 12calls described later, such as MPI_CANCEL or MPI_WAITANY, are additional sources of 13 14nondeterminism.)

If a process has a single thread of execution, then any two communications executed 1516by this process are ordered. On the other hand, if the process is multithreaded, then the 17semantics of thread execution may not define a relative order between two send operations 18 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 19any order. Similarly, if two receive operations that are logically concurrent receive two 2021successively sent messages, then the two messages can match the two receives in either 22order.

Example 3.5 An example of non-overtaking messages.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
26
     IF (rank .EQ. 0) THEN
27
        CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag, comm, ierr)
28
        CALL MPI_BSEND(buf2, count, MPI_REAL, 1, tag, comm, ierr)
29
     ELSE IF (rank .EQ. 1) THEN
30
        CALL MPI_RECV(buf1, count, MPI_REAL, 0, MPI_ANY_TAG, comm, status, ierr)
^{31}
        CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag, comm, status, ierr)
32
     END IF
33
```

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.

 $_{44}^{43}$ **Example 3.6** An example of two, intertwined matching pairs.

1 2

3

4 5

23

25

37

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

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

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

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

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

A buffered send operation that cannot complete because of a lack of buffer space is erroneous. When such a situation is detected, an error is signaled that may cause the program to terminate abnormally. On the other hand, a standard send operation that cannot complete because of lack of buffer space will merely block, waiting for buffer space to become available or for a matching receive to be posted. This behavior is preferable in many situations. Consider a situation where a producer repeatedly produces new values and sends them to a consumer. Assume that the producer produces new values faster than the consumer can consume them. If buffered sends are used, then a buffer overflow

Unofficial Draft for Comment Only

 24

 31

1 will result. Additional synchronization has to be added to the program so as to prevent $\mathbf{2}$ this from occurring. If standard sends are used, then the producer will be automatically 3 throttled, as its send operations will block when buffer space is unavailable. 4 In some situations, a lack of buffer space leads to deadlock situations. This is illustrated $\mathbf{5}$ by the examples below. 6 **Example 3.7** An exchange of messages. 78 CALL MPI_COMM_RANK(comm, rank, ierr) 9 IF (rank .EQ. 0) THEN 10 CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr) 11 CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr) 12ELSE IF (rank .EQ. 1) THEN 13 CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr) 14CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr) 15END IF 1617This 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. 1819**Example 3.8** An errant attempt to exchange messages. 2021CALL MPI_COMM_RANK(comm, rank, ierr) 22IF (rank .EQ. 0) THEN 23CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr) 24 CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr) 25ELSE IF (rank .EQ. 1) THEN 26CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr) 27CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr) 28END IF 29The receive operation of the first process must complete before its send, and can complete 30 only if the matching send of the second processor is executed. The receive operation of the 31 second process must complete before its send and can complete only if the matching send 32 of the first process is executed. This program will always deadlock. The same holds for any 33 other send mode. 34 35**Example 3.9** An exchange that relies on buffering. 36 CALL MPI_COMM_RANK(comm, rank, ierr) 37 IF (rank .EQ. 0) THEN 38 CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr) 39 CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr) 40 ELSE IF (rank .EQ. 1) THEN 41 CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr) 42CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr) 43 END IF 4445The message sent by each process has to be copied out before the send operation returns 46and the receive operation starts. For the program to complete, it is necessary that at least 47one of the two messages sent be buffered. Thus, this program can succeed only if the

```
<sup>48</sup> communication system can buffer at least count words of data.
```

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

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

Many programmers prefer to have more leeway and opt to use the "unsafe" programming style shown in Example 3.9. In such cases, the use of standard sends is likely to provide the best compromise between performance and robustness: quality implementations will provide sufficient buffering so that "common practice" programs will not deadlock. The buffered send mode can be used for programs that require more buffering, or in situations where the programmer wants more control. This mode might also be used for debugging purposes, as buffer overflow conditions are easier to diagnose than deadlock conditions.

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

3.6 Buffer Allocation and Usage

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

MPI_BUFFER_ATTACH(buffer, size)						
	IN buffer	initial buffer address (choice)	34			
	in Burier	mitial build address (choice)	35			
	IN size	buffer size, in bytes (non-negative integer)	36			
			37			
	C binding		38			
	<pre>int MPI_Buffer_attach(void *buffe</pre>	r, int size)	39			
	Fortran 2008 binding		40			
	0	`	41			
	MPI_Buffer_attach(buffer, size, ierror)					
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buffer						
	INTEGER, INTENT(IN) :: size		44			
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror	45			
	Fortron hinding		46			
	Fortran binding					
	MPI_BUFFER_ATTACH(BUFFER, SIZE, I	ERROR)	47			
	<type> BUFFER(*)</type>					

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

1	INTE	GER SIZE, IERROR					
2 3 4 5 6 7 8	Provides to MPI a buffer in the user's memory to be used for buffering outgoing mes- sages. The buffer is used only by messages sent in buffered mode. Only one buffer can be attached to a process at a time. In C, buffer is the starting address of a memory region. In Fortran, one can pass the first element of a memory region or a whole array, which must be 'simply contiguous' (for 'simply contiguous,' see also Section 18.1.12).						
9	MPI_BUF	FER_DETACH(buffer_	_addr, size)				
10 11	OUT	buffer_addr	initial buffer address (choice)				
12	OUT	size	buffer size, in bytes (non-negative integer)				
13							
14 15	C bindin int MPI_		*buffer_addr, int *size)				
16 17	Fortran	2008 binding					
18			ddr, size, ierror)				
19		INTRINSIC :: ISO_ (C_PTR), INTENT(OU	C_BINDING, ONLY : C_PTR				
20 21		GER, INTENT(OUT) :					
22	INTE	GER, OPTIONAL, INT	ENT(OUT) :: ierror				
23	Fortran	binding					
24		MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)					
25 26	• -	e> BUFFER_ADDR(*)					
27	INTEGER SIZE, IERROR						
28 29 30 31	size of the buffer hav	e detached buffer. T	y associated with MPI. The call returns the address and the his operation will block until all messages currently in the Jpon return of this function, the user may reuse or deallocate				
32 33	Example	e 3.10 Calls to attack	n and detach buffers.				
34		BUFFSIZE 10000	₹				
35 36	int size						
37	char *bu MPT Buff		UFFSIZE), BUFFSIZE);				
38			can now be used by MPI_Bsend */				
39		er_detach(&buff, &	•				
40 41		r size reduced to					
41		er_attach(buff, si					
43	/* BUIIE	r of 10000 bytes a	vallable again */				
44			though the C functions MPI_Buffer_attach and				
45			have a first argument of type void*, these arguments are				
46 47			er to the buffer is passed to MPI_Buffer_attach; the address o MPI_Buffer_detach, so that this call can return the pointer				
48		* *	e mpi module or mpif.h, the type of the buffer_addr argument				

is wrongly defined and the argument is therefore unused. In Fortran with the mpi_f08 module, the address of the buffer is returned as TYPE(C_PTR), see also Example 8.1 about the use of C_PTR pointers. (*End of advice to users.*)

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

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

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

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

Rationale. There is a wide spectrum of possible implementations of buffered communication: buffering can be done at sender, at receiver, or both; buffers can be dedicated to one sender-receiver pair, or be shared by all communications; buffering can be done in real or in virtual memory; it can use dedicated memory, or memory shared by other processes; buffer space may be allocated statically or be changed dynamically; etc. It does not seem feasible to provide a portable mechanism for querying or controlling buffering that would be compatible with all these choices, yet provide meaningful information. (*End of rationale.*)

3.6.1 Model Implementation of Buffered Mode

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

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

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

- Traverse sequentially the PME queue from head towards the tail, deleting all entries for communications that have completed, up to the first entry with an uncompleted request; update queue head to point to that entry.
- Compute the number, n, of bytes needed to store an entry for the new message. An upper bound on n can be computed as follows: A call to the function

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\overline{7}$

 31

MPI_PACK_SIZE(count, datatype, comm, size), with the count, datatype and comm arguments used in the MPI_BSEND call, returns an upper bound on the amount of space needed to buffer the message data (see Section 4.2). The MPI constant MPI_BSEND_OVERHEAD provides an upper bound on the additional space consumed by the entry (e.g., for pointers or envelope information).

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

3.7 Nonblocking Communication

19One can improve performance on many systems by overlapping communication and com-20putation. This is especially true on systems where communication can be executed au-21tonomously by an intelligent communication controller. Light-weight threads are one mech-22anism for achieving such overlap. An alternative mechanism that often leads to better 23performance is to use **nonblocking communication**. A nonblocking **send start** call ini- 24 tiates the send operation, but does not complete it. The send start call can return before 25the message was copied out of the send buffer. A separate send complete call is needed 26to complete the communication, i.e., to verify that the data has been copied out of the send 27buffer. With suitable hardware, the transfer of data out of the sender memory may proceed 28concurrently with computations done at the sender after the send was initiated and before it 29completed. Similarly, a nonblocking **receive start call** initiates the receive operation, but 30 does not complete it. The call can return before a message is stored into the receive buffer. 31 A separate **receive complete** call is needed to complete the receive operation and verify 32 that the data has been received into the receive buffer. With suitable hardware, the transfer 33 of data into the receiver memory may proceed concurrently with computations done after 34the receive was initiated and before it completed. The use of nonblocking receives may also 35 avoid system buffering and memory-to-memory copying, as information is provided early 36 on the location of the receive buffer.

37 Nonblocking send start calls can use the same four modes as blocking sends: *standard*, 38 buffered, synchronous and ready. These carry the same meaning. Sends of all modes, ready 39 excepted, can be started whether a matching receive has been posted or not; a nonblocking 40ready send can be started only if a matching receive is posted. In all cases, the send start 41 call is local: it returns immediately, irrespective of the status of other processes. If the call 42causes some system resource to be exhausted, then it will fail and return an error code. 43 Quality implementations of MPI should ensure that this happens only in "pathological" 44cases. That is, an MPI implementation should be able to support a large number of pending 45nonblocking operations.

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

1

2

3

4

5

6

7

8

9 10

11

12 13

14

15 16 17

18

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

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

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

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

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

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

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

3.7.1 Communication Request Objects

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

46 47

48

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

```
3.7.2
            Communication Initiation
\mathbf{2}
     We use the same naming conventions as for blocking communication: a prefix of B, S,
3
     or R is used for buffered, synchronous or ready mode. In addition a prefix of I (for
4
     immediate) indicates that the call is nonblocking.
5
6
\overline{7}
     MPI_ISEND(buf, count, datatype, dest, tag, comm, request)
8
       IN
                 buf
                                              initial address of send buffer (choice)
9
10
       IN
                                              number of elements in send buffer (non-negative
                 count
11
                                              integer)
12
       IN
                 datatype
                                              datatype of each send buffer element (handle)
13
       IN
                 dest
                                              rank of destination (integer)
14
                                              message tag (integer)
15
       IN
                 tag
16
       IN
                                              communicator (handle)
                 comm
17
       OUT
                                              communication request (handle)
                 request
18
19
     C binding
20
     int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest,
21
                     int tag, MPI_Comm comm, MPI_Request *request)
22
23
     Fortran 2008 binding
24
     MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror)
25
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
26
          INTEGER, INTENT(IN) :: count, dest, tag
27
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
28
          TYPE(MPI_Comm), INTENT(IN) :: comm
29
          TYPE(MPI_Request), INTENT(OUT) :: request
30
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
     Fortran binding
32
     MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
33
         <type> BUF(*)
34
          INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
35
36
          Start a standard mode, nonblocking send.
37
38
39
40
41
42
43
44
45
46
47
48
```

3.7. NONBLOCKING COMMUNICATION

MFI_IDSEND(bul, count, datatype, dest, tag, connil, request)					
	IN	buf	initial address of send buffer (choice)	2	
				3	
	IN	count	number of elements in send buffer (non-negative	4	
			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	
	OUT	request	communication request (handle)	11	
	001	icquest	communication request (nanule)	12	

MPI_IBSEND(buf_count_datatype_dest_tag_comm_request)

C binding

C binding
<pre>int MPI_Ibsend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>
<pre>int tag, MPI_Comm comm, MPI_Request *request)</pre>
Fortran 2008 binding

Fortran 2008 binding

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

Fortran binding

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

Start a buffered mode, nonblocking send.

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

IN	buf	initial address of send buffer (choice)	34
IN	count	number of elements in send buffer (non-negative integer)	35 36
IN	datatype	datatype of each send buffer element (handle)	37 38
IN	dest	rank of destination (integer)	39 40
IN	tag	message tag (integer)	41
IN	comm	communicator (handle)	42
OUT	request	communication request (handle)	43
a 1 ·			44 45

C binding

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

 31

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

3.7. NONBLOCKING COMMUNICATION

MPI_IRECV(buf, count, datatype, source, tag, comm, request)					
OUT	OUT buf initial address of receive buffer (choice)				
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 $MPI_ANY_TAG\xspace$ (integer)	8 9		
IN	comm	communicator (handle)	10		
OUT	request	communication request (handle)	11 12		
C binding					

int	MPI_Irecv(void	*buf,	int co	ount,	MPI_Datatype	datatype,	int	source,
	:	int t	ag, MP	I_Comm	comm,	MPI_Request	<pre>*request)</pre>		

Fortran 2008 binding

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

Fortran binding

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

Start a nonblocking receive.

 24

$\frac{1}{2}$	MPI_ISENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, request)			
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)	
15	IN	recvtype	datatype of each receive buffer element (handle)	
16	IN	source	rank of source or MPI_ANY_SOURCE (integer)	
17 18	IN	recvtag	receive tag or MPI_ANY_TAG (integer)	
19	IN	comm	communicator (handle)	
20	OUT	request	communication request (handle)	
21				
22 23	C binding	-		
24	<pre>int MPI_Isendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf,</pre>			
25	int recvcount, MPI_Datatype recvtype, int source, int recvtag,			
26 27	MPI_Comm comm, MPI_Request *request)			
28	Fortran 2008 binding			
29	MPI_Isendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,			
30	recvcount, recvtype, source, recvtag, comm, request, ierror)			
31 32	<pre>TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source,</pre>			
33	recvtag			
34	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype			
35	TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf TYPE(MPI_Comm), INTENT(IN) :: comm			
36 37	TYPE(MPI_COMM), INTENI(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request			
38	INTEG	ER, OPTIONAL, INTENT(OUT)) :: ierror	
39	Fortran b	binding		
40 41		DRECV (SENDBUF, SENDCOUNT,	SENDTYPE, DEST, SENDTAG, RECVBUF,	
41			SOURCE, RECVTAG, COMM, REQUEST, IERROR)	
43	• -	<pre>> SENDBUF(*), RECVBUF(*) FR SENDCOUNT SENDTYPE I</pre>	DEST, SENDTAG, RECVCOUNT, RECVTYPE,	
44		SOURCE, RECVTAG, CO		
45 46	Initiat		on request for a <i>send and receive</i> operation.	
40	1110100	to a nonorosting communicati	on request for a seria and receive operation.	
48				

MPI_ISEN	DRECV_REPLACE(buf, count, request)	ualalype, dest, sendlag, source, recviag, comm,	1 2	
INOUT	buf	initial address of send and receive buffer (choice)	3 4	
IN	count	number of elements in send and receive buffer	4 5 6	
IN	datatype	type of elements in send and receive buffer (handle)	7	
IN	dest	rank of destination (integer)	8 9	
IN	sendtag		9 10	
IN	source	rank of source or MPI_ANY_SOURCE (integer)	11	
IN	recvtag	receive message tag or MPLANY_TAG (integer)	12	
IN	comm	······································	$13 \\ 14$	
OUT	request		14 15	
001			16	
C binding	5		17	
int MPI_I	-	f, int count, MPI_Datatype datatype,	18 19	
		g, int source, int recvtag, MPI_Comm comm,	20	
	MPI_Request *request)		21	
	Fortran 2008 binding			
MPI_Isendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag, 23				
			$\frac{24}{25}$	
INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag				
	TYPE(MPI_Datatype), INTENT(IN) :: datatype			
	TYPE(MPI_Datatype), INTENI(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm			
	MPI_Request), INTENT(OUT)	-	29	
INTEG	ER, OPTIONAL, INTENT(OUT)		30	
Fortran b	inding		31	
		DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,	32 33	
	COMM, REQUEST, IERROF	<i>{</i>)	34	
	> BUF(*)	SENDTAG, SOURCE, RECVTAG, COMM, REQUEST,	35	
	IERROR	SENDING, SUCHCE, RECVING, COMM, REQUEST,	36	
T:+:-+			37	
	3		38 39	
	sage received.		40	
-	These calls allocate a communication request object and associate it with the request			
	handle (the argument request). The request can be used later to query the status of the 4			
	communication or wait for its completion.			
A non	A nonblocking send call indicates that the system may start copying data out of the 44			

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

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

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 18.1.10–18.1.20. (End of advice to users.)

3.7.3 Communication Completion

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

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

We shall use the following terminology: A null handle is a handle with value MPI_REQUEST_NULL. A persistent request and the handle to it are inactive if the request is not associated with any ongoing communication (see Section 3.9). A handle is active if it is neither null nor inactive. An empty status is a status which is set to return tag = MPI_ANY_TAG, source = MPI_ANY_SOURCE, error = MPI_SUCCESS, and is also internally configured so that calls to MPI_GET_COUNT, MPI_GET_ELEMENTS, and

²⁸ MPI_GET_ELEMENTS_X return count = 0 and MPI_TEST_CANCELLED returns false. We ²⁹ set a status variable to empty when the value returned by it is not significant. Status is set ³⁰ 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).

40 41

42

MPI_WAIT(request, status)

43	INOUT	request		request (hand	le)
44	OUT	status		status object ((Status)
45					
46	C binding	,			
47	int MPI_Wa	ait(MPI_Request	*request,	MPI_Status	*status)
48		-	•		

1

 $\mathbf{2}$

3

4

5

6

7 8 9

Fortran 2008 binding		1	
MPI_Wait(request, status, ierr		2	
TYPE(MPI_Request), INTENT(-	3	
TYPE(MPI_Status) :: status		4	
INTEGER, OPTIONAL, INTENT((UUT) :: lerror	5 6	
Fortran binding		7	
MPI_WAIT(REQUEST, STATUS, IERF		8	
INTEGER REQUEST, STATUS(MF	'I_STATUS_SIZE), IERROR	9	
A call to MPI_WAIT returns wh	en the operation identified by request is complete. If the	10	
request is an active persistent reque	est, it is marked inactive. Any other type of request is	11	
-	is set to MPI_REQUEST_NULL. MPI_WAIT is a non-local	12	
operation.		13 14	
	ormation on the completed operation. The content of	14	
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 ¹⁶			
(see Section 3.8).	may be queried by a can to with_TEST_CANCELLED	17	
	IT with a null or inactive request argument. In this case	18	
the operation returns immediately w		19	
		20	
	return of MPI_WAIT after a MPI_IBSEND implies that	21	
		22 23	
a buffer attached with MPI_BUFFER_ATTACH. Note that, at this point, we can no longer cancel the send (see Section 3.8). If a matching receive is never posted, then the			
buffer cannot be freed. This runs somewhat counter to the stated goal of MPI_CANCEL			
	gram space that was committed to the communication	26	
subsystem). (End of advice to	users.)	27	
Advise to involve where Tree		28	
_	multithreaded environment, a call to MPI_WAIT should allowing the thread scheduler to schedule another thread	29	
for execution. (<i>End of advice</i>		30 31	
		32	
		33	
MPI_TEST(request, flag, status)	, ,	34	
		35	
INOUT request	communication request (handle)	36	
OUT flag	true if operation completed (logical)	37	
OUT status	status object (Status)	38 39	
		40	
C binding		41	
int MPI_Test(MPI_Request *requ	nest, int *flag, MPI_Status *status)	42	
Fortran 2008 binding		43	
MPI_Test(request, flag, status		44	
TYPE(MPI_Request), INTENT(INOUT) :: request			
LOGICAL, INTENT(OUT) :: fl	0	46 47	
TYPE(MPI_Status) :: status		47	
INTEGER, OPTIONAL, INTENT((DOI) ·· TELLOL		

```
1
     Fortran binding
\mathbf{2}
     MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
3
          INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
4
          LOGICAL FLAG
5
          A call to MPI_TEST returns flag = true if the operation identified by request is complete.
6
     In such a case, the status object is set to contain information on the completed operation.
7
     If the request is an active persistent request, it is marked as inactive. Any other type of
8
     request is deallocated and the request handle is set to MPI_REQUEST_NULL. The call returns
9
     flag = false if the operation identified by request is not complete. In this case, the value of
10
     the status object is undefined. MPI_TEST is a local operation.
11
          The return status object for a receive operation carries information that can be accessed
12
     as described in Section 3.2.5. The status object for a send operation carries information
13
     that can be accessed by a call to MPI_TEST_CANCELLED (see Section 3.8).
14
          One is allowed to call MPI_TEST with a null or inactive request argument. In such a
15
     case the operation returns with flag = true and empty status.
16
          The functions MPI_WAIT and MPI_TEST can be used to complete both sends and
17
     receives.
18
19
                              The use of the nonblocking MPL_TEST call allows the user to
           Advice to users.
20
           schedule alternative activities within a single thread of execution. An event-driven
21
           thread scheduler can be emulated with periodic calls to MPI_TEST. (End of advice to
22
           users.)
23
24
25
     Example 3.11 Simple usage of nonblocking operations and MPI_WAIT.
26
     CALL MPI_COMM_RANK(comm, rank, ierr)
27
     IF (rank .EQ. 0) THEN
28
         CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)
29
         **** do some computation to mask latency ****
30
         CALL MPI_WAIT(request, status, ierr)
^{31}
     ELSE IF (rank .EQ. 1) THEN
32
         CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr)
33
         **** do some computation to mask latency ****
34
         CALL MPI_WAIT(request, status, ierr)
35
     END IF
36
37
          A request object can be deallocated by using the following operation.
38
39
     MPI_REQUEST_FREE(request)
40
41
       INOUT
                                              communication request (handle)
                 request
42
43
     C binding
44
     int MPI_Request_free(MPI_Request *request)
45
46
     Fortran 2008 binding
47
     MPI_Request_free(request, ierror)
48
          TYPE(MPI_Request), INTENT(INOUT) :: request
```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

MPI_REQUEST_FREE(REQUEST, IERROR) INTEGER REQUEST, IERROR

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

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

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

Example 3.12 An example using MPI_REQUEST_FREE.

```
CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
IF (rank .EQ. 0) THEN
   DO i=1,n
      CALL MPI_ISEND(outval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)
      CALL MPI_REQUEST_FREE(req, ierr)
      CALL MPI_IRECV(inval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)
                                                                                 34
      CALL MPI_WAIT(req, status, ierr)
                                                                                 35
   END DO
                                                                                 36
ELSE IF (rank .EQ. 1) THEN
                                                                                 37
   CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
   CALL MPI_WAIT(req, status, ierr)
   DO I=1,n-1
      CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
      CALL MPI_REQUEST_FREE(req, ierr)
      CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
      CALL MPI_WAIT(req, status, ierr)
   END DO
   CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
   CALL MPI_WAIT(req, status, ierr)
END IF
```

1

2

3

4

56

 $\overline{7}$

8

9

10

11

1213

14

1516

17

18

19

20

21

22

23

242526

2728

29

30

31

32

33

38

39

40

41

42

43

44

45

46

47

```
1
            Semantics of Nonblocking Communications
     3.7.4
2
     The semantics of nonblocking communication is defined by suitably extending the definitions
3
     in Section 3.5.
4
5
     Order Nonblocking communication operations are ordered according to the execution order
6
     of the calls that initiate the communication. The non-overtaking requirement of Section 3.5
7
     is extended to nonblocking communication, with this definition of order being used.
8
9
     Example 3.13 Message ordering for nonblocking operations.
10
11
     CALL MPI_COMM_RANK(comm, rank, ierr)
12
     IF (RANK .EQ. O) THEN
13
         CALL MPI_ISEND(a, 1, MPI_REAL, 1, 0, comm, r1, ierr)
14
         CALL MPI_ISEND(b, 1, MPI_REAL, 1, 0, comm, r2, ierr)
15
     ELSE IF (rank .EQ. 1) THEN
16
         CALL MPI_IRECV(a, 1, MPI_REAL, 0, MPI_ANY_TAG, comm, r1, ierr)
17
         CALL MPI_IRECV(b, 1, MPI_REAL, 0, 0, comm, r2, ierr)
18
     END IF
19
     CALL MPI_WAIT(r1, status, ierr)
20
     CALL MPI_WAIT(r2, status, ierr)
21
     The first send of process zero will match the first receive of process one, even if both messages
22
     are sent before process one executes either receive.
23
^{24}
     Progress A call to MPI_WAIT that completes a receive will eventually terminate and return
25
26
     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
27
     call is executed by the sender to complete the send. Similarly, a call to MPI_WAIT that
28
     completes a send will eventually return if a matching receive has been started, unless the
29
     receive is satisfied by another send, and even if no call is executed to complete the receive.
30
^{31}
     Example 3.14 An illustration of progress semantics.
32
33
     CALL MPI_COMM_RANK(comm, rank, ierr)
34
     IF (RANK .EQ. 0) THEN
35
         CALL MPI_SSEND(a, 1, MPI_REAL, 1, 0, comm, ierr)
36
         CALL MPI_SEND(b, 1, MPI_REAL, 1, 1, comm, ierr)
37
     ELSE IF (rank .EQ. 1) THEN
38
         CALL MPI_IRECV(a, 1, MPI_REAL, 0, 0, comm, r, ierr)
39
         CALL MPI_RECV(b, 1, MPI_REAL, 0, 1, comm, status, ierr)
40
         CALL MPI_WAIT(r, status, ierr)
41
     END IF
42
43
          This code should not deadlock in a correct MPI implementation. The first synchronous
^{44}
     send of process zero must complete after process one posts the matching (nonblocking)
45
     receive even if process one has not yet reached the completing wait call. Thus, process zero
```

⁴⁶ will continue and execute the second send, allowing process one to complete execution.

⁴⁷ If an MPI_TEST that completes a receive is repeatedly called with the same arguments, ⁴⁸ and a matching send has been started, then the call will eventually return flag = true, unless

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

3.7.5 Multiple Completions

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

MPI_WAITANY(count, array_of_requests, index, status)

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

C binding

Fortran 2008 binding
MPI_Waitany(count, array_of_requests, index, status, ierror)
INTEGER, INTENT(IN) :: count
TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
INTEGER, INTENT(OUT) :: index
TYPE(MPI_Status) :: status
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

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

Blocks until one of the operations associated with the active requests in the array has completed. If more than one operation is enabled and can terminate, one is arbitrarily chosen. Returns in index the index of that request in the array and returns in status the status of the completing operation. (The array is indexed from zero in C, and from one in Fortran.) If the request is an active persistent request, it is marked inactive. Any other type of request is deallocated and the request handle is set to MPI_REQUEST_NULL.

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

1 The execution of MPI_WAITANY with an array containing multiple entries has the $\mathbf{2}$ same effect as the execution of MPI_WAIT with the array entry indicated by the output 3 value of index (unless the output value of index is MPI_UNDEFINED). MPI_WAITANY with 4 an array containing one active entry is equivalent to MPI_WAIT. 56 MPI_TESTANY(count, array_of_requests, index, flag, status) $\overline{7}$ 8 IN count list length (non-negative integer) 9 INOUT array_of_requests array of requests (array of handles) 10 index of operation that completed or OUT index 11 MPI_UNDEFINED if none completed (integer) 1213OUT true if one of the operations is complete (logical) flag 14OUT status status object (Status) 1516C binding 17int MPI_Testany(int count, MPI_Request array_of_requests[], int *index, 18 int *flag, MPI_Status *status) 1920Fortran 2008 binding 21MPI_Testany(count, array_of_requests, index, flag, status, ierror) 22INTEGER, INTENT(IN) :: count 23TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count) 24INTEGER, INTENT(OUT) :: index 25LOGICAL, INTENT(OUT) :: flag 26TYPE(MPI_Status) :: status 27INTEGER, OPTIONAL, INTENT(OUT) :: ierror 28Fortran binding 29 MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR) 30 INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE), 31IERROR 32 LOGICAL FLAG 33 34 Tests for completion of either one or none of the operations associated with active 35handles. In the former case, it returns flag = true, returns in index the index of this request 36 in the array, and returns in status the status of that operation. If the request is an active 37 persistent request, it is marked as inactive. Any other type of request is deallocated and 38 the handle is set to MPI_REQUEST_NULL. (The array is indexed from zero in C, and from 39 one in Fortran.) In the latter case (no operation completed), it returns flag = false, returns 40a value of MPI_UNDEFINED in index and status is undefined. 41 The array may contain null or inactive handles. If the array contains no active handles 42then the call returns immediately with flag = true, index = MPI_UNDEFINED, and an empty 43status. 44If the array of requests contains active handles then the execution of MPI_TESTANY 45has the same effect as the execution of MPI_TEST with each of the array elements in some 46arbitrary order, until one call returns flag = true, or all fail. In the former case, index is 47set to indicate which array element returned flag = true and in the latter case, it is set to

48

MPI_UNDEFINED. MPI_TESTANY with an array containing one active entry is equivalent to MPI_TEST.

MPI_WAITALL(count, array_of_requests, array_of_statuses)

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

C binding

Fortran 2008 binding

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

Fortran binding

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

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

The error-free execution of MPI_WAITALL has the same effect as the execution of MPI_WAIT for each of the array elements in some arbitrary order. MPI_WAITALL with an array of length one is equivalent to MPI_WAIT.

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

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

Unofficial Draft for Comment Only

1 MPI_TESTALL(count, array_of_requests, flag, array_of_statuses) 2 IN count lists length (non-negative integer) 3 INOUT array_of_requests array of requests (array of handles) 4 5OUT flag (logical) 6 OUT array_of_statuses array of status objects (array of Status) 7 8 C binding 9 int MPI_Testall(int count, MPI_Request array_of_requests[], int *flag, 10 MPI_Status array_of_statuses[]) 11 12Fortran 2008 binding 13MPI_Testall(count, array_of_requests, flag, array_of_statuses, ierror) 14INTEGER, INTENT(IN) :: count 15TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count) 16LOGICAL, INTENT(OUT) :: flag 17 TYPE(MPI_Status) :: array_of_statuses(*) 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 19Fortran binding 20MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR) 21INTEGER COUNT, ARRAY_OF_REQUESTS(*), ARRAY_OF_STATUSES(MPI_STATUS_SIZE, 22 *), IERROR 23LOGICAL FLAG 2425Returns flag = true if all communications associated with active handles in the array 26have completed (this includes the case where no handle in the list is active). In this case, each 27status entry that corresponds to an active request is set to the status of the corresponding 28operation. Active persistent requests are marked inactive. Requests of any other type are 29deallocated and the corresponding handles in the array are set to MPI_REQUEST_NULL. 30 Each status entry that corresponds to a null or inactive handle is set to empty. 31 Otherwise, flag = false is returned, no request is modified and the values of the status 32 entries are undefined. This is a local operation. 33 Errors that occurred during the execution of MPI_TESTALL are handled in the same 34manner as errors in MPI_WAITALL. 35 36 37 38 39 40 41 4243 44 4546 47 48

MPI_WAITSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)			1
			2
IN	incount	length of array_of_requests (non-negative integer)	3
INOUT	array_of_requests	array of requests (array of handles)	4
		· <u>-</u> · · · · · · · · · · · · · · · · · · ·	5
OUT	outcount	number of completed requests (integer)	6 7
OUT	array_of_indices	array of indices of operations that completed (array of integers)	8 9
OUT	array_of_statuses	array of status objects for operations that completed	9 10
	5	(array of Status)	11
			12
C binding	g		13
<pre>int MPI_Waitsome(int incount, MPI_Request array_of_requests[],</pre>			
<pre>int *outcount, int array_of_indices[],</pre>			15
<pre>MPI_Status array_of_statuses[])</pre>			16
Fortran 2008 binding			17
MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices,			18
	array_of_statuses, i	-	19 20
INTEG	INTEGER, INTENT(IN) :: incount		
	-	<pre>UT) :: array_of_requests(incount)</pre>	21 22
	INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)		
TYPE(TYPE(MPI_Status) :: array_of_statuses(*)		
INTEG	INTEGER, OPTIONAL, INTENT(OUT) :: ierror		
Fortran b	vinding		25 26
	3	QUESTS, OUTCOUNT, ARRAY_OF_INDICES,	27
TH L_WAILS	ARRAY_OF_STATUSES, I		28
INTEG		UESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),	29
ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *), IERROR			30
TT 7 •			31

Waits until at least one of the operations associated with active handles in the list have completed. Returns in outcount the number of requests from the list array_of_requests that have completed. Returns in the first outcount locations of the array array_of_indices the indices of these operations (index within the array array_of_requests; the array is indexed from zero in C and from one in Fortran). Returns in the first outcount locations of the array array_of_status the status for these completed operations. Completed active persistent requests are marked as inactive. Any other type or request that completed is deallocated, and the associated handle is set to MPI_REQUEST_NULL.

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

When one or more of the communications completed by MPI_WAITSOME fails, then 42it is desirable to return specific information on each communication. The arguments 43 outcount, array_of_indices and array_of_statuses will be adjusted to indicate completion of 44all communications that have succeeded or failed. The call will return the error code 45MPI_ERR_IN_STATUS and the error field of each status returned will be set to indicate 46success or to indicate the specific error that occurred. The call will return MPI_SUCCESS 47if no request resulted in an error, and will return another error code if it failed for other 48

Unofficial Draft for Comment Only

32

33

34

35

36

37

38

39

40

```
1
     reasons (such as invalid arguments). In such cases, it will not update the error fields of the
\mathbf{2}
     statuses.
3
4
     MPI_TESTSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)
5
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
11
       OUT
                 array_of_indices
                                              array of indices of operations that completed (array
12
                                              of integers)
13
       OUT
                 array_of_statuses
                                              array of status objects for operations that completed
14
                                              (array of Status)
15
16
     C binding
17
     int MPI_Testsome(int incount, MPI_Request array_of_requests[],
18
                     int *outcount, int array_of_indices[],
19
                     MPI_Status array_of_statuses[])
20
21
     Fortran 2008 binding
22
     MPI_Testsome(incount, array_of_requests, outcount, array_of_indices,
23
                     array_of_statuses, ierror)
^{24}
          INTEGER, INTENT(IN) :: incount
25
          TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)
26
          INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
27
          TYPE(MPI_Status) :: array_of_statuses(*)
28
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
     Fortran binding
30
     MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
^{31}
                     ARRAY_OF_STATUSES, IERROR)
32
          INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
33
                      ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *), IERROR
34
35
          Behaves like MPI_WAITSOME, except that it returns immediately. If no operation has
36
     completed it returns outcount = 0. If there is no active handle in the list it returns outcount
37
     = MPI_UNDEFINED.
38
          MPI_TESTSOME is a local operation, which returns immediately, whereas
39
     MPI_WAITSOME will block until a communication completes, if it was passed a list that
40
     contains at least one active handle. Both calls fulfill a fairness requirement: If a request
41
     for a receive repeatedly appears in a list of requests passed to MPI_WAITSOME or
42
     MPI_TESTSOME, and a matching send has been posted, then the receive will eventually
43
     succeed, unless the send is satisfied by another receive; and similarly for send requests.
44
          Errors that occur during the execution of MPI_TESTSOME are handled as for
45
     MPI_WAITSOME.
46
47
           Advice to users. The use of MPI_TESTSOME is likely to be more efficient than the use
48
           of MPI_TESTANY. The former returns information on all completed communications,
```

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 5that 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 3.15** Client-server code (starvation can occur). 1213 CALL MPI_COMM_SIZE(comm, size, ierr) 14CALL MPI_COMM_RANK(comm, rank, ierr) 15IF (rank .GT. 0) THEN ! client code 16DO 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 20ELSE ! rank=0 -- server code 21DO i=1,size-1 22 CALL MPI_IRECV(a(1,i), n, MPI_REAL, i, tag, & 23comm, request_list(i), ierr) 24 END DO 25DO WHILE(.TRUE.) 26CALL MPI_WAITANY(size-1, request_list, index, status, ierr) 27CALL DO_SERVICE(a(1, index)) ! handle one message 28CALL MPI_IRECV(a(1, index), n, MPI_REAL, index, tag, & 29comm, request_list(index), ierr) 30 END DO 31 END IF 32 33 34 Example 3.16 Same code, using MPI_WAITSOME. 35CALL 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 42ELSE ! rank=0 -- server code 43 DO i=1, size-1 44CALL MPI_IRECV(a(1,i), n, MPI_REAL, i, tag, & 45comm, request_list(i), ierr) 46END DO 47DO WHILE(.TRUE.) 48

```
1
            CALL MPI_WAITSOME(size, request_list, numdone, &
2
                                 indices, statuses, ierr)
3
            DO i=1, numdone
4
                CALL DO_SERVICE(a(1, indices(i)))
5
                CALL MPI_IRECV(a(1, indices(i)), n, MPI_REAL, 0, tag, &
6
                                 comm, request_list(indices(i)), ierr)
7
            END DO
8
         END DO
9
     END IF
10
11
            Non-Destructive Test of status
     3.7.6
12
     This call is useful for accessing the information associated with a request, without freeing
13
     the request (in case the user is expected to access it later). It allows one to layer libraries
14
     more conveniently, since multiple layers of software may access the same completed request
15
     and extract from it the status information.
16
17
18
     MPI_REQUEST_GET_STATUS(request, flag, status)
19
       IN
                                              request (handle)
                 request
20
21
                                              boolean flag, same as from MPI_TEST (logical)
       OUT
                 flag
22
       OUT
                                              status object if flag is true (Status)
                 status
23
24
     C binding
25
     int MPI_Request_get_status(MPI_Request request, int *flag,
26
                     MPI_Status *status)
27
28
     Fortran 2008 binding
29
     MPI_Request_get_status(request, flag, status, ierror)
30
          TYPE(MPI_Request), INTENT(IN) :: request
^{31}
          LOGICAL, INTENT(OUT) :: flag
32
          TYPE(MPI_Status) :: status
33
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
     Fortran binding
35
     MPI_REQUEST_GET_STATUS (REQUEST, FLAG, STATUS, IERROR)
36
          INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
37
          LOGICAL FLAG
38
39
          Sets flag = true if the operation is complete, and, if so, returns in status the request
40
     status. However, unlike test or wait, it does not deallocate or inactivate the request; a
41
     subsequent call to test, wait or free should be executed with that request. It sets flag =
42
     false if the operation is not complete.
43
          One is allowed to call MPI_REQUEST_GET_STATUS with a null or inactive request
44
     argument. In such a case the operation returns with flag = true and empty status.
45
46
47
48
```

3.8 Probe and Cancel

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

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

Cancelling a send request by calling MPI_CANCEL is deprecated. Cancelling a sendrecv request by calling MPI_CANCEL is not allowed.

3.8.1 Probe

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

	_		,	18		
	IN	source	rank of source or MPI_ANY_SOURCE (integer)	19		
	IN	tag	message tag or MPI_ANY_TAG (integer)	20		
	IN	comm	communicator (handle)	21		
	OUT	flag	(logical)	22 23		
	OUT	status	status object (Status)	23 24		
				25		
(C binding					
	0	, probe(int source int tag	MPT Comm comm int *flag	27		

Fortran 2008 binding

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

Fortran binding

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

MPI_IPROBE(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.

1 2			, then the content of the status object can be sub- on $3.2.5$ to find the source, tag and length of the				
3	probed me						
4	-	0	the same communicator, and the source and tag re-				
5	turned in status by MPI_IPROBE will receive the message that was matched by the probe, if						
6	no other intervening receive occurs after the probe, and the send is not successfully cancelled						
7	before the receive. If the receiving process is multithreaded, it is the user's responsibility						
8	to ensure that the last condition holds.						
9	The source $\operatorname{argument}$ of MPI_PROBE can be MPI_ANY_SOURCE, and the tag $\operatorname{argument}$						
10	can be MPI_ANY_TAG, so that one can probe for messages from an arbitrary source and/or						
11			fic communication context must be provided with				
12 13	the comm	8					
14		-	sage immediately after it has been probed for, and veral times before it is received.				
15		0 0 1	source returns $flag = true$, and the status object				
16			= MPI_ANY_TAG, and count = 0; see Section 3.10 .				
17	10000100 000						
18							
19	MPI_PRO	BE(source, tag, comm, status)					
20 21	IN	source	rank of source or MPI_ANY_SOURCE (integer)				
22	IN	tag	message tag or MPI_ANY_TAG (integer)				
23	IN	comm	communicator (handle)				
24	OUT	status	status object (Status)				
25 26	~						
27	C binding						
28	int MPI_P	robe(int source, int tag	, MPI_Comm comm, MPI_Status *status)				
29		3					
30		(anurco the commentation	Fortran 2008 binding				
31	<pre>MPI_Probe(source, tag, comm, status, ierror)</pre>						
32	TYPE(MPI_Comm), INTENT(IN) :: comm						
	TYPE(GER, INTENT(IN) :: source (MPI_Comm), INTENT(IN) ::	, tag				
33	TYPE (TYPE (GER, INTENT(IN) :: source (MPI_Comm), INTENT(IN) :: (MPI_Status) :: status	, tag comm				
34	TYPE (TYPE (GER, INTENT(IN) :: source (MPI_Comm), INTENT(IN) ::	, tag comm				
34 35	TYPE(TYPE(INTEG	GER, INTENT(IN) :: source (MPI_Comm), INTENT(IN) :: (MPI_Status) :: status GER, OPTIONAL, INTENT(OUT) Dinding	, tag comm) :: ierror				
34 35 36	TYPE(TYPE(INTEG Fortran b MPI_PROBE	GER, INTENT(IN) :: source (MPI_Comm), INTENT(IN) :: (MPI_Status) :: status GER, OPTIONAL, INTENT(OUT) Dinding E(SOURCE, TAG, COMM, STATU	, tag comm) :: ierror JS, IERROR)				
34 35	TYPE(TYPE(INTEG Fortran b MPI_PROBE	GER, INTENT(IN) :: source (MPI_Comm), INTENT(IN) :: (MPI_Status) :: status GER, OPTIONAL, INTENT(OUT) Dinding E(SOURCE, TAG, COMM, STATU	, tag comm) :: ierror				
34 35 36 37	TYPE(TYPE(INTEG Fortran h MPI_PROBE INTEG	GER, INTENT(IN) :: source (MPI_Comm), INTENT(IN) :: (MPI_Status) :: status GER, OPTIONAL, INTENT(OUT) Dinding E(SOURCE, TAG, COMM, STATU GER SOURCE, TAG, COMM, STATU	, tag comm) :: ierror JS, IERROR)				
34 35 36 37 38	TYPE(TYPE) INTEG Fortran b MPI_PROBE INTEG MPI_F	GER, INTENT(IN) :: source (MPI_Comm), INTENT(IN) :: (MPI_Status) :: status GER, OPTIONAL, INTENT(OUT) Dinding E(SOURCE, TAG, COMM, STATU GER SOURCE, TAG, COMM, STATU	, tag comm) :: ierror US, IERROR) ATUS(MPI_STATUS_SIZE), IERROR OBE except that it is a blocking call that returns				
34 35 36 37 38 39	TYPE (TYPE (INTEG Fortran h MPI_PROBE INTEG MPI_F only after The M	GER, INTENT(IN) :: source (MPI_Comm), INTENT(IN) :: (MPI_Status) :: status GER, OPTIONAL, INTENT(OUT) Dinding E(SOURCE, TAG, COMM, STATU GER SOURCE, TAG, COMM, STATU PROBE behaves like MPI_IPR a matching message has been (PI implementation of MPI_PR	<pre>, tag comm) :: ierror US, IERROR) ATUS(MPI_STATUS_SIZE), IERROR OBE except that it is a blocking call that returns found. COBE and MPI_IPROBE needs to guarantee progress:</pre>				
34 35 36 37 38 39 40	TYPE (TYPE (INTEG Fortran & MPI_PROBE INTEG MPI_F only after The M if a call to	GER, INTENT(IN) :: source (MPI_Comm), INTENT(IN) :: (MPI_Status) :: status GER, OPTIONAL, INTENT(OUT) oinding C(SOURCE, TAG, COMM, STATU GER SOURCE, TAG, COMM, STATU PROBE behaves like MPI_IPR a matching message has been API implementation of MPI_PR MPI_PROBE has been issued	<pre>, tag comm) :: ierror US, IERROR) ATUS(MPI_STATUS_SIZE), IERROR OBE except that it is a blocking call that returns found. OBE and MPI_IPROBE needs to guarantee progress: l by a process, and a send that matches the probe</pre>				
34 35 36 37 38 39 40 41 42 43	TYPE (TYPE (INTEG Fortran & MPI_PROBE INTEG MPI_F only after The M if a call to has been i	GER, INTENT(IN) :: source (MPI_Comm), INTENT(IN) :: (MPI_Status) :: status GER, OPTIONAL, INTENT(OUT) oinding E(SOURCE, TAG, COMM, STATU GER SOURCE, TAG, COMM, STATU PROBE behaves like MPI_IPR a matching message has been MPI implementation of MPI_PR MPI_PROBE has been issued nitiated by some process, the	<pre>, tag comm) :: ierror US, IERROR) ATUS(MPI_STATUS_SIZE), IERROR OBE except that it is a blocking call that returns found. OBE and MPI_IPROBE needs to guarantee progress: l by a process, and a send that matches the probe en the call to MPI_PROBE will return, unless the</pre>				
34 35 36 37 38 39 40 41 42 43 44	TYPE (TYPE (INTEG Fortran & MPI_PROBE INTEG MPI_F only after The M if a call to has been i message is	GER, INTENT(IN) :: source (MPI_Comm), INTENT(IN) :: (MPI_Status) :: status GER, OPTIONAL, INTENT(OUT) oinding E(SOURCE, TAG, COMM, STATU GER SOURCE, TAG, COMM, STATU PROBE behaves like MPI_IPR a matching message has been MPI_PROBE has been issued nitiated by some process, the received by another concurrent	<pre>, tag comm) :: ierror US, IERROR) ATUS(MPI_STATUS_SIZE), IERROR OBE except that it is a blocking call that returns found. COBE and MPI_IPROBE needs to guarantee progress: l by a process, and a send that matches the probe en the call to MPI_PROBE will return, unless the ent receive operation (that is executed by another</pre>				
34 35 36 37 38 39 40 41 42 43 44 45	TYPE (TYPE (INTEG Fortran & MPI_PROBE INTEG MPI_F only after The M if a call to has been i message is thread at t	GER, INTENT(IN) :: source (MPI_Comm), INTENT(IN) :: (MPI_Status) :: status GER, OPTIONAL, INTENT(OUT) oinding (SOURCE, TAG, COMM, STATU GER SOURCE, TAG, COMM, STATU PROBE behaves like MPI_IPR a matching message has been MPI_PROBE has been issued nitiated by some process, the received by another concurrent the probing process). Similarly	<pre>, tag comm) :: ierror US, IERROR) ATUS(MPI_STATUS_SIZE), IERROR OBE except that it is a blocking call that returns found. OBE and MPI_IPROBE needs to guarantee progress: l by a process, and a send that matches the probe en the call to MPI_PROBE will return, unless the ent receive operation (that is executed by another y, if a process busy waits with MPI_IPROBE and a</pre>				
34 35 36 37 38 39 40 41 42 43 44 45 46	TYPE (TYPE (INTEG Fortran & MPI_PROBE INTEG MPI_F only after The M if a call to has been i message is thread at t	GER, INTENT(IN) :: source (MPI_Comm), INTENT(IN) :: (MPI_Status) :: status GER, OPTIONAL, INTENT(OUT) oinding C(SOURCE, TAG, COMM, STATU GER SOURCE, TAG, COMM, STATU PROBE behaves like MPI_IPR a matching message has been MPI_PROBE has been issued nitiated by some process, the received by another concurrent the probing process). Similarly message has been issued, then	<pre>, tag comm) :: ierror US, IERROR) ATUS(MPI_STATUS_SIZE), IERROR OBE except that it is a blocking call that returns found. OBE and MPI_IPROBE needs to guarantee progress: l by a process, and a send that matches the probe en the call to MPI_PROBE will return, unless the ent receive operation (that is executed by another y, if a process busy waits with MPI_IPROBE and a the call to MPI_IPROBE will eventually return flag</pre>				
34 35 36 37 38 39 40 41 42 43 44 45	TYPE (TYPE (INTEG Fortran & MPI_PROBE INTEG MPI_F only after The M if a call to has been i message is thread at t matching r = true unl	GER, INTENT(IN) :: source (MPI_Comm), INTENT(IN) :: (MPI_Status) :: status GER, OPTIONAL, INTENT(OUT) oinding C(SOURCE, TAG, COMM, STATU GER SOURCE, TAG, COMM, STATU PROBE behaves like MPI_IPR a matching message has been MPI_PROBE has been issued nitiated by some process, the received by another concurrent the probing process). Similarly message has been issued, then	<pre>, tag comm) :: ierror US, IERROR) ATUS(MPI_STATUS_SIZE), IERROR OBE except that it is a blocking call that returns found. OBE and MPI_IPROBE needs to guarantee progress: l by a process, and a send that matches the probe en the call to MPI_PROBE will return, unless the ent receive operation (that is executed by another y, if a process busy waits with MPI_IPROBE and a</pre>				

Example 3.17 Use blocking probe to wait for an incoming message.

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

Each message is received with the right type.

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

```
23
    CALL MPI_COMM_RANK(comm, rank, ierr)
                                                                                      ^{24}
    IF (rank .EQ. 0) THEN
                                                                                      25
       CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
                                                                                      26
    ELSE IF (rank .EQ. 1) THEN
                                                                                      27
       CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
                                                                                      28
    ELSE IF (rank .EQ. 2) THEN
                                                                                      29
       DO i=1,2
                                                                                      30
          CALL MPI_PROBE(MPI_ANY_SOURCE, 0, &
                                                                                      31
                           comm, status, ierr)
                                                                                      32
          IF (status(MPI_SOURCE) .EQ. 0) THEN
                                                                                      33
100
              CALL MPI_RECV(i, 1, MPI_INTEGER, MPI_ANY_SOURCE, &
                                                                                      34
                             0, comm, status, ierr)
                                                                                      35
          ELSE
                                                                                      36
200
              CALL MPI_RECV(x, 1, MPI_REAL, MPI_ANY_SOURCE, &
                                                                                      37
                             0, comm, status, ierr)
                                                                                      38
          END IF
                                                                                      39
       END DO
                                                                                      40
    END IF
                                                                                      41
```

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

Advice to users. In a multithreaded MPI program, MPI_PROBE and MPI_IPROBE might need special care. If a thread probes for a message and then

Unofficial Draft for Comment Only

1

2

6

10

11

12

13

14

15

16

17

18 19

20 21

22

42

43

44

45

46 47

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 [30]. 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 [30, 27].

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.

 $33 \\ 34$

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

35		tobe(source, tag, comm, mag,	message, status)
36	IN	source	rank of source or MPI_ANY_SOURCE (integer)
37	IN	tag	message tag or MPI_ANY_TAG (integer)
38 39	IN	comm	communicator (handle)
40	OUT	flag	flag (logical)
41	OUT	message	returned message (handle)
42 43	OUT	status	status object (Status)
43 44			
45	\mathbf{C} binding	0	
46	int MPI_]	Improbe(int source, int ta	ag, MPI_Comm comm, int *flag,
47		MPI_Message *message	, MPI_Status *status)

⁴⁸ Fortran 2008 binding

1

2

3

4

5

6

7

8

9

10 11

12

13

14

15

16

17

18 19

20 21

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

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

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

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

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

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

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

Rationale. MPI_MESSAGE_NO_PROC was chosen instead of MPI_MESSAGE_PROC_NULL to avoid possible confusion as another null handle constant. (*End of rationale.*) 24

1 MPI_MPROBE(source, tag, comm, message, status) 2 IN rank of source or MPI_ANY_SOURCE (integer) source 3 IN message tag or MPI_ANY_TAG (integer) tag 4 5IN comm communicator (handle) 6 OUT returned message (handle) message 7 OUT status status object (Status) 8 9 C binding 10 int MPI_Mprobe(int source, int tag, MPI_Comm comm, MPI_Message *message, 11 MPI_Status *status) 1213 Fortran 2008 binding 14MPI_Mprobe(source, tag, comm, message, status, ierror) 15INTEGER, INTENT(IN) :: source, tag 16TYPE(MPI_Comm), INTENT(IN) :: comm 17 TYPE(MPI_Message), INTENT(OUT) :: message 18 TYPE(MPI_Status) :: status 19INTEGER, OPTIONAL, INTENT(OUT) :: ierror 20Fortran binding 21MPI_MPROBE(SOURCE, TAG, COMM, MESSAGE, STATUS, IERROR) 22 INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR 2324MPI_MPROBE behaves like MPI_IMPROBE except that it is a blocking call that returns 25only after a matching message has been found. 26The implementation of MPI_MPROBE and MPI_IMPROBE needs to guarantee progress 27in the same way as in the case of MPI_PROBE and MPI_IPROBE. 2829 3.8.3 Matched Receives 30 31 The functions MPI_MRECV and MPI_IMRECV receive messages that have been previously 32 matched by a matching probe (Section 3.8.2). 33 34MPI_MRECV(buf, count, datatype, message, status) 35 36 OUT buf initial address of receive buffer (choice) 37 IN number of elements in receive buffer (non-negative count 38 integer) 39 IN datatype datatype of each receive buffer element (handle) 40 41 INOUT message (handle) message 42OUT status object (Status) status 43 44 C binding 45 int MPI_Mrecv(void *buf, int count, MPI_Datatype datatype, 46 MPI_Message *message, MPI_Status *status) 4748 Fortran 2008 binding

```
1
MPI_Mrecv(buf, count, datatype, message, status, ierror)
                                                                                            2
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Message), INTENT(INOUT) :: message
                                                                                            5
    TYPE(MPI_Status) :: status
                                                                                            6
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
MPI_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR)
                                                                                            10
     <type> BUF(*)
                                                                                            11
    INTEGER COUNT, DATATYPE, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR
                                                                                            12
                                                                                            13
    This call receives a message matched by a matching probe operation (Section 3.8.2).
                                                                                            14
    The receive buffer consists of the storage containing count consecutive elements of the
                                                                                            15
type specified by datatype, starting at address buf. The length of the received message must
                                                                                            16
be less than or equal to the length of the receive buffer. An overflow error occurs if all
                                                                                            17
incoming data does not fit, without truncation, into the receive buffer.
                                                                                            18
    If the message is shorter than the receive buffer, then only those locations corresponding
                                                                                            19
to the (shorter) message are modified.
    On return from this function, the message handle is set to MPI_MESSAGE_NULL. All
                                                                                           20
                                                                                            21
errors that occur during the execution of this operation are handled according to the error
                                                                                            22
handler set for the communicator used in the matching probe call that produced the message
                                                                                            23
handle.
    If MPI_MRECV is called with MPI_MESSAGE_NO_PROC as the message argument, the
                                                                                            ^{24}
                                                                                            25
call returns immediately with the status object set to source = MPI_PROC_NULL,
                                                                                            26
tag = MPI_ANY_TAG, and count = 0, as if a receive from MPI_PROC_NULL was issued (see
Section 3.10). A call to MPI_MRECV with MPI_MESSAGE_NULL is erroneous.
                                                                                            27
                                                                                            28
                                                                                            29
MPI_IMRECV(buf, count, datatype, message, request)
                                                                                            30
                                                                                            31
  OUT
            buf
                                        initial address of receive buffer (choice)
                                                                                            32
  IN
            count
                                        number of elements in receive buffer (non-negative
                                                                                            33
                                        integer)
                                                                                            34
  IN
            datatype
                                        datatype of each receive buffer element (handle)
                                                                                           35
                                                                                            36
  INOUT
            message
                                        message (handle)
                                                                                            37
  OUT
            request
                                        communication request (handle)
                                                                                            38
                                                                                            39
C binding
                                                                                            40
int MPI_Imrecv(void *buf, int count, MPI_Datatype datatype,
                                                                                            41
               MPI_Message *message, MPI_Request *request)
                                                                                            42
                                                                                            43
Fortran 2008 binding
                                                                                            44
MPI_Imrecv(buf, count, datatype, message, request, ierror)
                                                                                            45
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                            46
    INTEGER, INTENT(IN) :: count
                                                                                            47
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                            48
```

Unofficial Draft for Comment Only

```
1
          TYPE(MPI_Message), INTENT(INOUT) :: message
\mathbf{2}
          TYPE(MPI_Request), INTENT(OUT) :: request
3
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     Fortran binding
5
     MPI_IMRECV(BUF, COUNT, DATATYPE, MESSAGE, REQUEST, IERROR)
6
          <type> BUF(*)
7
          INTEGER COUNT, DATATYPE, MESSAGE, REQUEST, IERROR
8
9
          MPI_IMRECV is the nonblocking variant of MPI_MRECV and starts a nonblocking
10
     receive of a matched message. Completion semantics are similar to MPI_IRECV as described
11
     in Section 3.7.2. On return from this function, the message handle is set to
12
     MPI_MESSAGE_NULL.
13
         If MPI_IMRECV is called with MPI_MESSAGE_NO_PROC as the message argument, the
14
     call returns immediately with a request object which, when completed, will yield a status
15
     object set to source = MPI_PROC_NULL, tag = MPI_ANY_TAG, and count = 0, as if a receive
16
     from MPI_PROC_NULL was issued (see Section 3.10). A call to MPI_IMRECV with
17
     MPI_MESSAGE_NULL is erroneous.
18
19
           Advice to implementors. If reception of a matched message is started with
           MPI_IMRECV, then it is possible to cancel the returned request with MPI_CANCEL. If
20
           MPI_CANCEL succeeds, the matched message must be found by a subsequent message
21
           probe (MPI_PROBE, MPI_IPROBE, MPI_MPROBE, or MPI_IMPROBE), received by
22
23
           a subsequent receive operation or cancelled by the sender. See Section 3.8.4 for details
24
           about MPI_CANCEL. The cancellation of operations initiated with MPI_IMRECV may
25
           fail. (End of advice to implementors.)
26
27
     3.8.4 Cancel
28
29
30
     MPI_CANCEL(request)
^{31}
       IN
                                             communication request (handle)
                 request
32
33
34
     C binding
     int MPI_Cancel(MPI_Request *request)
35
36
     Fortran 2008 binding
37
     MPI_Cancel(request, ierror)
38
          TYPE(MPI_Request), INTENT(IN) :: request
39
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     Fortran binding
42
     MPI_CANCEL(REQUEST, IERROR)
          INTEGER REQUEST, IERROR
43
44
          A call to MPI_CANCEL marks for cancellation a pending, nonblocking communica-
45
     tion operation (send or receive). Cancelling a send request by calling MPI_CANCEL is
46
```

deprecated. The cancel call is local. It returns immediately, possibly before the communi-

cation is actually cancelled. It is still necessary to call MPI_REQUEST_FREE, MPI_WAIT or

47

MPI_TEST (or any of the derived operations) with the cancelled request as argument after the call to MPI_CANCEL. If a communication is marked for cancellation, then a MPI_WAIT call for that communication is guaranteed to return, irrespective of the activities of other processes (i.e., MPI_WAIT behaves as a local function); similarly if MPI_TEST is repeatedly called in a busy wait loop for a cancelled communication, then MPI_TEST will eventually be successful.

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

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

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

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

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

MDL TEST CANCELLED(status flor)				
MPI_TEST_CANCELLED(status, flag)				
IN status	status object (Status)	34		
OUT flag	(logical)	35		
0		36		
C binding		37		
int MPI_Test_cancelled(const MPI_S	tatus kstatus int kflag)	38		
int Millest_cancerred(const Milb	tatus *status, int *ilag)	39		
Fortran 2008 binding		40		
MPI_Test_cancelled(status, flag, i	error)	41		
TYPE(MPI_Status), INTENT(IN) :	: status	42		
LOGICAL, INTENT(OUT) :: flag		43		
INTEGER, OPTIONAL, INTENT(OUT)	:: ierror	44		
Fortran binding				
0		46		
	MPI_TEST_CANCELLED(STATUS, FLAG, IERROR)			
INTEGER STATUS(MPI_STATUS_SIZE), IERROR				

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

LOGICAL FLAG

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

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

Advice to implementors. If a send operation uses an "eager" protocol (data is transferred to the receiver before a matching receive is posted), then the cancellation of this send may require communication with the intended receiver in order to free allocated buffers. On some systems this may require an interrupt to the intended receiver. Note that, while communication may be needed to implement

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

3.9 Persistent Communication Requests

Often a communication with the same argument list (with the exception of the buffer con-24tents) is repeatedly executed within the inner loop of a parallel computation. In such a 25situation, it may be possible to optimize the communication by binding the list of com-26munication arguments to a **persistent** communication request once and, then, repeatedly 27using the request to initiate and complete operations. In the case of point-to-point commu-28nication, the persistent request thus created can be thought of as a communication port or 29 a "half-channel." It does not provide the full functionality of a conventional channel, since 30 there is no binding of the send port to the receive port. This construct allows reduction 31 of the overhead for communication between the process and communication controller, but 32 not of the overhead for communication between one communication controller and another. 33 It is not necessary that messages sent with a persistent point-to-point request be received 34 by a receive operation using a persistent point-to-point request, or vice versa. 35

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

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

41 42

40

- 43
- 44
- 45
- 46 47
- 48

1

2

3

4

5

6

7 8

9

10 11

12

13

14

15

16

17

18

19

20 21

MPI_SEND_INIT(buf, count, datatype, dest, tag, comm, request) ¹						
IN	buf	initial address of send buffer (choice)	2			
IN	count	number of elements sent (non-negative integer)	3 4			
IN	datatype	type of each element (handle)	5			
IN	dest	rank of destination (integer)	6 7			
IN	tag	message tag (integer)	8			
IN	comm	communicator (handle)	9			
OUT	request	communication request (handle)	10			
			11 12			
C binding						
int MDT 9	int MPT Send init(const void *huf int count MPT Datatype datatype					

(

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

Fortran 2008 binding

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

Fortran binding

MPI_SEND_INIT(BUF,	COUNT, DAT	ΓΑΤΥΡΕ,	DEST,	TAG,	COMM,	REQUEST,	IERROR)
<type> BUF(*)</type>							
INTEGER COUNT,	DATATYPE,	DEST,	TAG, C	OMM,	REQUEST	, IERROR	

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

MPI_BSEND_INIT(buf, count, datatype, dest, tag, comm, request) 3				
IN	buf	initial address of send buffer (choice)	34	
IN	count	number of elements sent (non-negative integer)	35 36	
IN	datatype	type of each element (handle)	37	
IN	dest	rank of destination (integer)	38	
IN	tag	message tag (integer)	$\frac{39}{40}$	
IN	comm	communicator (handle)	41	
OUT	request	communication request (handle)	42	
			43	

C binding

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

Fortran 2008 binding

Unofficial Draft for Comment Only

 31

```
1
     MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
\mathbf{2}
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
3
         INTEGER, INTENT(IN) :: count, dest, tag
4
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
         TYPE(MPI_Comm), INTENT(IN) :: comm
5
6
         TYPE(MPI_Request), INTENT(OUT) :: request
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
     Fortran binding
9
     MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
10
         <type> BUF(*)
11
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
12
13
         Creates a persistent communication request for a buffered mode send.
14
15
     MPI_SSEND_INIT(buf, count, datatype, dest, tag, comm, request)
16
17
       IN
                buf
                                            initial address of send buffer (choice)
18
       IN
                count
                                            number of elements sent (non-negative integer)
19
       IN
                datatype
                                            type of each element (handle)
20
21
       IN
                dest
                                            rank of destination (integer)
22
                                            message tag (integer)
       IN
                tag
23
                                            communicator (handle)
       IN
                comm
24
25
       OUT
                                            communication request (handle)
                request
26
27
     C binding
28
     int MPI_Ssend_init(const void *buf, int count, MPI_Datatype datatype,
29
                    int dest, int tag, MPI_Comm comm, MPI_Request *request)
30
     Fortran 2008 binding
^{31}
     MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror)
32
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
33
34
         INTEGER, INTENT(IN) :: count, dest, tag
         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
39
     Fortran binding
40
     MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
41
         <type> BUF(*)
42
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
43
         Creates a persistent communication object for a synchronous mode send operation.
44
45
46
47
48
```

IN	buf	initial address of send buffer (choice)
	count	number of elements sent (non-negative integer)
IN		
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)
bindi	ng	
nt MPI		void *buf, int count, MPI_Datatype datatype,
		t tag, MPI_Comm comm, MPI_Request *request)
	2008 binding	detetune dest ten some nemest issued
		, datatype, dest, tag, comm, request, ierror)), INTENT(IN), ASYNCHRONOUS :: buf
	EGER, INTENT(IN) :	
		NTENT(IN) :: datatype
	E(MPI_Comm), INTEN	
יסעיד		
	-	TENT(OUT) :: request
	-	TENT(OUT) :: request TENT(OUT) :: ierror
INTI ortran	EGER, OPTIONAL, IN	TENT(OUT) :: ierror
INT ortran 'I_RSE	EGER, OPTIONAL, IN binding ND_INIT(BUF, COUNT	-
INT ortran PI_RSE <tyj< td=""><td>EGER, OPTIONAL, IN binding ND_INIT(BUF, COUNT pe> BUF(*)</td><td>TENT(OUT) :: ierror , DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)</td></tyj<>	EGER, OPTIONAL, IN binding ND_INIT(BUF, COUNT pe> BUF(*)	TENT(OUT) :: ierror , DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
INT) ortran I_RSE <tyj INT)</tyj 	EGER, OPTIONAL, IN binding ND_INIT(BUF, COUNT pe> BUF(*) EGER COUNT, DATATYI	TENT(OUT) :: ierror , DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR
INT ortran PI_RSE <tyj INT</tyj 	EGER, OPTIONAL, IN binding ND_INIT(BUF, COUNT pe> BUF(*) EGER COUNT, DATATYI	TENT(OUT) :: ierror , DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
INT ortran PI_RSE <tyj INT Crea</tyj 	EGER, OPTIONAL, IN binding ND_INIT(BUF, COUNT pe> BUF(*) EGER COUNT, DATATYP ates a persistent comm	TENT(OUT) :: ierror , DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR nunication object for a ready mode send operation.
INT ortran PI_RSE <ty] INT Crea</ty] 	EGER, OPTIONAL, IN binding ND_INIT(BUF, COUNT pe> BUF(*) EGER COUNT, DATATY ates a persistent comm	TENT(OUT) :: ierror , DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR nunication object for a ready mode send operation.
INT ortran >I_RSE >I_RSE INT Crea PI_REC OUT	EGER, OPTIONAL, IN binding ND_INIT(BUF, COUNT pe> BUF(*) EGER COUNT, DATATY ates a persistent comm CV_INIT(buf, count, da buf	<pre>TENT(OUT) :: ierror , DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR nunication object for a ready mode send operation. atatype, source, tag, comm, request)</pre>
INT ortran PI_RSE (ty) INT Crea IPI_REC OUT IN	EGER, OPTIONAL, IN binding ND_INIT(BUF, COUNT pe> BUF(*) EGER COUNT, DATATYN ates a persistent comm CV_INIT(buf, count, da buf count	<pre>TENT(OUT) :: ierror , DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR nunication object for a ready mode send operation. atatype, source, tag, comm, request)</pre>
INT ortran PI_RSE (ty) INT Crea IPI_REC OUT IN IN	EGER, OPTIONAL, IN binding ND_INIT(BUF, COUNT pe> BUF(*) EGER COUNT, DATATYI ates a persistent comm CV_INIT(buf, count, da buf count datatype	<pre>TENT(OUT) :: ierror , DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR nunication object for a ready mode send operation. atatype, source, tag, comm, request)</pre>
INT ortran PI_RSE (ty) INT Crea IPI_REC OUT IN	EGER, OPTIONAL, IN binding ND_INIT(BUF, COUNT pe> BUF(*) EGER COUNT, DATATYN ates a persistent comm CV_INIT(buf, count, da buf count	<pre>TENT(OUT) :: ierror , DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR nunication object for a ready mode send operation. atatype, source, tag, comm, request)</pre>
INT ortran PI_RSE (ty) INT Crea IPI_REC OUT IN IN	EGER, OPTIONAL, IN binding ND_INIT(BUF, COUNT pe> BUF(*) EGER COUNT, DATATYI ates a persistent comm CV_INIT(buf, count, da buf count datatype	<pre>TENT(OUT) :: ierror , DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR nunication object for a ready mode send operation. atatype, source, tag, comm, request)</pre>
INT ortran PI_RSE (ty) INT Crea IPI_REC OUT IN IN IN	EGER, OPTIONAL, IN binding ND_INIT(BUF, COUNT pe> BUF(*) EGER COUNT, DATATYI ates a persistent comm CV_INIT(buf, count, da buf count datatype source	<pre>TENT(OUT) :: ierror , DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR hunication object for a ready mode send operation. atatype, source, tag, comm, request)</pre>
INT ortran PI_RSE (ty) INT Crea IPI_REC OUT IN IN IN IN	EGER, OPTIONAL, IN binding ND_INIT(BUF, COUNT pe> BUF(*) EGER COUNT, DATATY ates a persistent comm CV_INIT(buf, count, da buf count datatype source tag	<pre>TENT(OUT) :: ierror , DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR nunication object for a ready mode send operation. atatype, source, tag, comm, request) initial address of receive buffer (choice) number of elements received (non-negative integer) type of each element (handle) rank of source or MPI_ANY_SOURCE (integer) message tag or MPI_ANY_TAG (integer)</pre>
INT PT PT PT ST PT ST ST ST ST ST ST ST ST ST S	EGER, OPTIONAL, IN binding ND_INIT(BUF, COUNT pe> BUF(*) EGER COUNT, DATATYN ates a persistent comm CV_INIT(buf, count, da buf count datatype source tag comm	<pre>TENT(OUT) :: ierror , DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR nunication object for a ready mode send operation. atatype, source, tag, comm, request) initial address of receive buffer (choice) number of elements received (non-negative integer) type of each element (handle) rank of source or MPI_ANY_SOURCE (integer) message tag or MPI_ANY_TAG (integer) communicator (handle)</pre>
INT ortran PI_RSE (ty) INT Crea OUT IN IN IN IN IN IN OUT	EGER, OPTIONAL, IN binding ND_INIT(BUF, COUNT pe> BUF(*) EGER COUNT, DATATYN ates a persistent comm CV_INIT(buf, count, da buf count datatype source tag comm request ng	TENT(OUT) :: ierror , DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR nunication object for a ready mode send operation. atatype, source, tag, comm, request) initial address of receive buffer (choice) number of elements received (non-negative integer) type of each element (handle) rank of source or MPI_ANY_SOURCE (integer) message tag or MPI_ANY_TAG (integer) communicator (handle) communication request (handle)
INT ortran PI_RSE (ty) INT Crea IPI_REC OUT IN IN IN IN IN IN SUT	EGER, OPTIONAL, IN binding ND_INIT(BUF, COUNT pe> BUF(*) EGER COUNT, DATATYN ates a persistent comm CV_INIT(buf, count, da buf count datatype source tag comm request ng _Recv_init(void *bu	<pre>TENT(OUT) :: ierror , DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR nunication object for a ready mode send operation. atatype, source, tag, comm, request)</pre>
INT ortran PI_RSE (ty) INT Crea OUT IN IN IN IN IN IN OUT	EGER, OPTIONAL, IN binding ND_INIT(BUF, COUNT pe> BUF(*) EGER COUNT, DATATYN ates a persistent comm CV_INIT(buf, count, da buf count datatype source tag comm request ng _Recv_init(void *bu	TENT(OUT) :: ierror , DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR nunication object for a ready mode send operation. atatype, source, tag, comm, request) initial address of receive buffer (choice) number of elements received (non-negative integer) type of each element (handle) rank of source or MPI_ANY_SOURCE (integer) message tag or MPI_ANY_TAG (integer) communicator (handle) communication request (handle)

Unofficial Draft for Comment Only

```
1
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
2
          INTEGER, INTENT(IN) :: count, source, tag
3
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
4
          TYPE(MPI_Comm), INTENT(IN) :: comm
5
          TYPE(MPI_Request), INTENT(OUT) :: request
6
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
     Fortran binding
8
     MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
9
          <type> BUF(*)
10
          INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
11
12
          Creates a persistent communication request for a receive operation. The argument buf
13
     is marked as OUT because the user gives permission to write on the receive buffer by passing
14
     the argument to MPI_RECV_INIT.
15
          A persistent communication request is inactive after it was created — no active com-
16
     munication is attached to the request.
17
          A communication that uses a persistent request is initiated by the function
18
     MPI_START.
19
20
     MPI_START(request)
21
22
                                             communication request (handle)
       INOUT
                 request
23
^{24}
     C binding
25
     int MPI_Start(MPI_Request *request)
26
     Fortran 2008 binding
27
     MPI_Start(request, ierror)
28
          TYPE(MPI_Request), INTENT(INOUT) :: request
29
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
^{31}
     Fortran binding
32
     MPI_START(REQUEST, IERROR)
33
          INTEGER REQUEST, IERROR
34
          The argument, request, is a handle returned by one of the previous five calls. The
35
     associated request should be inactive. The request becomes active once the call is made.
36
         If the request is for a send with ready mode, then a matching receive should be posted
37
     before the call is made. The communication buffer should not be modified after the call,
38
     and until the operation completes.
39
          The call is local, with similar semantics to the nonblocking communication operations
40
41
     described in Section 3.7. That is, a call to MPI_START with a request created by
42
     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
43
     in the same manner as a call to MPI_IBSEND; and so on.
44
45
46
47
48
```

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

C binding

int MPI_Startall(int count, MPI_Request array_of_requests[])

Fortran 2008 binding

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

Fortran binding

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

Start all communications associated with requests in array_of_requests. A call to MPI_STARTALL(count, array_of_requests) has the same effect as calls to

MPI_START (&array_of_requests[i]), executed for i=0,..., count-1, in some arbitrary order. A communication started with a call to MPI_START or MPI_STARTALL is completed by a call to MPI_WAIT, MPI_TEST, or one of the derived functions described in Section 3.7.5. The request becomes inactive after successful completion of such call. The request is not deallocated and it can be activated anew by an MPI_START or MPI_STARTALL call.

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

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

Create (Start Complete)* Free

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

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

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 18.1.10–18.1.20. (End of advice to users.)

1

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

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 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

Datatypes

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

4.1 Derived Datatypes

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

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

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

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

1	a general datatype is an opaque object that specifies two things.
•	A sequence of basic datatypes
•	A sequence of integer (byte) displacements

Unofficial Draft for Comment Only

 24

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

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

be the associated type signature. This type map, together with a base address buf, specifies a communication buffer: the communication buffer that consists of n entries, where the *i*-th entry is at address buf + $disp_i$ and has type $type_i$. A message assembled from such a communication buffer will consist of n values, of the types defined by Typesig.

Most datatype constructors have replication count or block length arguments. Allowed values are 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 4.1.11, the case where the second argument count has value > 1.

The basic datatypes presented in Section 3.2.2 are particular cases of a general datatype, and are predefined. Thus, MPI_INT is a predefined handle to a datatype with type map $\{(int, 0)\}$, 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

32

33

34 35 36

37

39

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

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

$$extent(Typemap) = ub(Typemap) - lb(Typemap).$$
(4.1)

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 4.1 Section 4.1.

Unofficial Draft for Comment Only

Let

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

Rationale. The definition of extent is motivated by the assumption that the amount of padding added at the end of each structure in an array of structures is the least needed to fulfill alignment constraints. More explicit control of the extent is provided in Section 4.1.6. Such explicit control is needed in cases where the assumption does not hold, for example, where union types are used. In Fortran, structures can be expressed with several language features, e.g., common blocks, SEQUENCE derived types, or BIND(C) derived types. The compiler may use different alignments, and therefore, it is recommended to use MPI_TYPE_CREATE_RESIZED for arrays of structures if an alignment may cause an alignment-gap at the end of a structure as described in Section 4.1.6 and in Section 18.1.15. (End of rationale.)

4.1.1 Type Constructors with Explicit Addresses

In Fortran, the functions MPI_TYPE_CREATE_HVECTOR,
MPI_TYPE_CREATE_HINDEXED, MPI_TYPE_CREATE_HINDEXED_BLOCK,
MPI_TYPE_CREATE_STRUCT, and MPI_GET_ADDRESS accept arguments of type
INTEGER(KIND=MPI_ADDRESS_KIND), wherever arguments of type MPI_Aint are used in C.
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).

4.1.2 Datatype Constructors

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

		33
MPI_TYPE_CONTIGUOUS(count, oldtype,	newtype)	34
IN count r	replication count (non-negative integer)	35
IN oldtype o	bld datatype (handle)	$36 \\ 37$
OUT newtype n	new datatype (handle)	38
		39
C binding		40
int MPI_Type_contiguous(int count, N	MPI_Datatype oldtype,	41
MPI_Datatype *newtype)		42
		43
Fortran 2008 binding		44
MPI_Type_contiguous(count, oldtype,	newtype, ierror)	45
INTEGER, INTENT(IN) :: count		46
TYPE(MPI_Datatype), INTENT(IN)		47
TYPE(MPI_Datatype), INTENT(OUT)	:: newtype	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

29 30

31

1	INTEG	ER, OPTIONAL, INTENT(OUT	[) :: ierror
2 3	Fortran h	0	
4 5		CONTIGUOUS(COUNT, OLDTY) ER COUNT, OLDTYPE, NEWT	
6 7 8			by concatenating count copies of generated as the size of the concatenated copies.
9 10	-	4.2 Let oldtype have type in The type map of the dataty	nap {(double, 0), (char, 8)}, with extent 16, and let pe returned by newtype is
11 12	$\{(dor)$	uble, 0), (char, 8), (double, 1)	$6), (char, 24), (double, 32), (char, 40)\};$
13	i.e., altern	ating double and char eleme	ents, with displacements $0, 8, 16, 24, 32, 40$.
14 15	In ger	neral, assume that the type n	nap of oldtype is
16 17	$\{(typ)\}$	$(pe_0, disp_0), \ldots, (type_{n-1}, disp_n)$	$_{n-1})\},$
18 19	with exten	t ex . Then newtype has a ty	pe map with $count \cdot n$ entries defined by:
19 20	$\{(type_0)\}$	$(disp_0), \ldots, (type_{n-1}, disp_{n-1})$	1), $(type_0, disp_0 + ex), \dots, (type_{n-1}, disp_{n-1} + ex),$
21 22	$\ldots, (ty$	$pe_0, disp_0 + ex \cdot (count - 1))$	$\ldots, (type_{n-1}, disp_{n-1} + ex \cdot (count - 1))\}.$
24 25 26 27 28	cation of a obtained b	a datatype into locations th	TOR is a more general constructor that allows repli- at consist of equally spaced blocks. Each block is umber of copies of the old datatype. The spacing ent of the old datatype.
29	MPI_TYPI	E_VECTOR(count, blocklengt	h, stride, oldtype, newtype)
$30 \\ 31$	IN	count	number of blocks (non-negative integer)
32 33	IN	blocklength	number of elements in each block (non-negative integer)
34 35 36	IN	stride	number of elements between start of each block (integer)
37	IN	oldtype	old datatype (handle)
38	OUT	newtype	new datatype (handle)
39 40	C binding	a	
41		•	nt blocklength, int stride,
42 43		MPI_Datatype oldtyp	e, MPI_Datatype *newtype)
44	Fortran 2	2008 binding	
45		-	th, stride, oldtype, newtype, ierror)
46 47		ER, INTENT(IN) :: count (MPI_Datatype), INTENT(I)	
48		(MPI_Datatype), INTENT(O	

INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1
Fortran binding MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR	2 3 4 5 6
Example 4.3 Assume, again, that oldtype has type map {(double, 0), (char, 8)}, with extent 16. A call to MPI_TYPE_VECTOR(2, 3, 4, oldtype, newtype) will create the datatype with type map,	7 8 9
$\{(\texttt{double},0),(\texttt{char},8),(\texttt{double},16),(\texttt{char},24),(\texttt{double},32),(\texttt{char},40),$	10 11
$(\texttt{double}, 64), (\texttt{char}, 72), (\texttt{double}, 80), (\texttt{char}, 88), (\texttt{double}, 96), (\texttt{char}, 104) \}.$	12 13
That is, two blocks with three copies each of the old type, with a stride of 4 elements $(4 \cdot 16 \text{ bytes})$ between the the start of each block.	14 15 16
Example 4.4 A call to MPI_TYPE_VECTOR(3, 1, -2, oldtype, newtype) will create the datatype,	17 18 19
$\{(\texttt{double}, 0), (\texttt{char}, 8), (\texttt{double}, -32), (\texttt{char}, -24), (\texttt{double}, -64), (\texttt{char}, -56)\}.$	20
In general, assume that oldtype has type map,	21 22
$\{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\},\$	23 24
with extent ex . Let bl be the blocklength. The newly created datatype has a type map with count \cdot bl \cdot n entries:	24 25 26
$\{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1}),$	27 28
$(type_0, disp_0 + ex), \ldots, (type_{n-1}, disp_{n-1} + ex), \ldots,$	29 30
$(type_0, disp_0 + (bl - 1) \cdot ex), \dots, (type_{n-1}, disp_{n-1} + (bl - 1) \cdot ex),$	31 32
$(type_0, disp_0 + stride \cdot ex), \dots, (type_{n-1}, disp_{n-1} + stride \cdot ex), \dots,$	33
$(type_0, disp_0 + (stride + bl - 1) \cdot ex), \dots, (type_{n-1}, disp_{n-1} + (stride + bl - 1) \cdot ex), \dots,$	$\frac{34}{35}$
$(type_0, disp_0 + stride \cdot (count - 1) \cdot ex), \ldots,$	36 37
$(type_{n-1}, disp_{n-1} + stride \cdot (count - 1) \cdot ex), \dots,$	38 39
$(type_0, disp_0 + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots,$	40 41
$(type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex)\}.$	42
	43 44
A call to MPI_TYPE_CONTIGUOUS(count, oldtype, newtype) is equivalent to a call to MPI_TYPE_VECTOR(count, 1, 1, oldtype, newtype), or to a call to MPI_TYPE_VECTOR(1,	45
count, n, oldtype, newtype), n arbitrary.	46 47
	48

```
1
      Hvector The function MPI_TYPE_CREATE_HVECTOR is identical to
\mathbf{2}
      MPI_TYPE_VECTOR, except that stride is given in bytes, rather than in elements. The
3
      use for both types of vector constructors is illustrated in Section 4.1.14. (H stands for
4
      "heterogeneous").
5
6
      MPI_TYPE_CREATE_HVECTOR(count, blocklength, stride, oldtype, newtype)
7
8
        IN
                                                  number of blocks (non-negative integer)
                   count
9
        IN
                   blocklength
                                                  number of elements in each block (non-negative
10
                                                  integer)
11
        IN
                   stride
                                                  number of bytes between start of each block (integer)
12
13
        IN
                   oldtype
                                                  old datatype (handle)
14
        OUT
                                                  new datatype (handle)
                   newtype
15
16
      C binding
17
      int MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride,
18
                       MPI_Datatype oldtype, MPI_Datatype *newtype)
19
20
      Fortran 2008 binding
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
      Fortran binding
29
      MPI_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,
30
                       IERROR)
31
           INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR
32
           INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE
33
34
          Assume that oldtype has type map,
35
            \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\},\
36
37
      with extent ex. Let bl be the blocklength. The newly created datatype has a type map with
38
      \operatorname{count} \cdot \operatorname{bl} \cdot n entries:
39
40
            \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1}), 
41
42
            (type_0, disp_0 + ex), \dots, (type_{n-1}, disp_{n-1} + ex), \dots,
43
            (type_0, disp_0 + (bl - 1) \cdot ex), \dots, (type_{n-1}, disp_{n-1} + (bl - 1) \cdot ex),
44
45
            (type_0, disp_0 + \mathsf{stride}), \ldots, (type_{n-1}, disp_{n-1} + \mathsf{stride}), \ldots,
46
47
            (type_0, disp_0 + stride + (bl - 1) \cdot ex), \ldots,
48
```

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

Indexed The function MPI_TYPE_INDEXED allows replication of an old datatype into a sequence of blocks (each block is a concatenation of the old datatype), where each block can contain a different number of copies and have a different displacement. All block displacements are multiples of the old type extent.

MPI_TYPE_INDEXED(count, array_of_blocklengths, array_of_displacements, oldtype,

		15
ype)		16
	number of blocks – also number of entries in	17
	array_of_displacements and array_of_blocklengths	18
	(non-negative integer)	19
_blocklengths	number of elements per block (array of non-negative	20
- 0		21
		22
_displacements		23
	(array of integers)	24
	old datatype (handle)	25
	new datatype (handle)	26
		27
		28
vod(int count co	anat int array of blocklongths[]	29
		30
•		31
Datatype *newtyp	e)	32
ing		33
count, array_of_1	blocklengths, array_of_displacements,	34
		35
	t int array_of_d Datatype *newtyp ing count, array_of_1	<pre>number of blocks - also number of entries in array_of_displacements and array_of_blocklengths (non-negative integer) _blocklengths number of elements per block (array of non-negative integers) _displacements displacement for each block, in multiples of oldtype (array of integers) old datatype (handle) new datatype (handle) new datatype (handle)</pre>

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

Fortran binding

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

1	Example	4.5 Let oldtype have type may	$p \{(double, 0), (char, 8)\}, with extent 16. Let B =$
2	. ,		FYPE_INDEXED(2, B, D, oldtype, newtype)
4		with type map,	
5			$80), ({\tt char}, 88), ({\tt double}, 96), ({\tt char}, 104),$
6	(doul	$ple, 0), (char, 8) \}.$	
8	That is, the displacement		ting at displacement 64, and one copy starting at
9 10	In gen	eral, assume that $oldtype$ has	type map,
11	$\{(typ$	$(type_0, disp_0), \ldots, (type_{n-1}, disp_n)$	$_{-1})\},$
12 13			ocklengths argument and D be the ewly created datatype has $n \cdot \sum_{i=0}^{\text{count}-1} B[i]$ entries:
14 15	$\{(typ$	$pe_0, disp_0 + D[0] \cdot ex), \dots, (typ_0)$	$e_{n-1}, disp_{n-1} + D[0] \cdot ex), \dots,$
16	(type	$c_0, disp_0 + (D[0] + B[0] - 1) \cdot e_0$	$x),\ldots,$
17 18	(type	$a_{n-1}, disp_{n-1} + (D[0] + B[0] - $	$1) \cdot ex), \ldots,$
19	(type	$\mathbf{C}_0, disp_0 + D[count-1] \cdot ex), \dots,$	$(type_{n-1}, disp_{n-1} + D[count-1] \cdot ex), \dots,$
20 21	(type	$\mathbf{x}_0, disp_0 + (D[count-1] + B[cou])$	$nt extsf{-1}] - 1) \cdot ex), \dots,$
22	(type	$a_{n-1}, disp_{n-1} + (D[count-1] + E)$	$3[\operatorname{count-1}] - 1) \cdot ex)\}.$
23 24	A call	to MPI_TYPE_VECTOR(coun	t, blocklength, stride, oldtype, newtype) is $equivalent$
25			, B, D, oldtype, newtype) where
26	D[j] =	$j \cdot \text{stride}, \ j = 0, \dots, \text{count} - j$	1,
27 28	and		
29	B[j] =	= blocklength, $j = 0, \ldots,$ count	:-1.
30 31	Hindexed	The function MPI TYPE CR	EATE_HINDEXED is identical to
32			a displacements in array_of_displacements are spec-
33	ified in byt	es, rather than in multiples of	f the oldtype extent.
34 35			
36 37	MPI_TYPE	E_CREATE_HINDEXED(count, oldtype, newtype)	array_of_blocklengths, array_of_displacements,
38	IN	count	number of blocks – also number of entries in
39 40			array_of_displacements and array_of_blocklengths (non-negative integer)
41	IN	array_of_blocklengths	number of elements in each block (array of
42			non-negative integers)
43 44	IN	array_of_displacements	byte displacement of each block (array of integers)
45	IN	oldtype	old datatype (handle)
46	OUT	newtype	new datatype (handle)
47 48	C binding	y 5	

1 int MPI_Type_create_hindexed(int count, const int array_of_blocklengths[], 2 const MPI_Aint array_of_displacements[], MPI_Datatype oldtype, MPI_Datatype *newtype) 4 Fortran 2008 binding 5 MPI_Type_create_hindexed(count, array_of_blocklengths, 6 array_of_displacements, oldtype, newtype, ierror) 7 INTEGER, INTENT(IN) :: count, array_of_blocklengths(count) INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: 9 array_of_displacements(count) 10 TYPE(MPI_Datatype), INTENT(IN) :: oldtype 11 TYPE(MPI_Datatype), INTENT(OUT) :: newtype 12INTEGER, OPTIONAL, INTENT(OUT) :: ierror 13 14Fortran binding 15MPI_TYPE_CREATE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, 16ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR) 17 INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), OLDTYPE, NEWTYPE, IERROR 18 INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*) 19 Assume that oldtype has type map, 2021 $\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\},\$ 22 with extent ex. Let B be the array_of_blocklengths argument and D be the 23array_of_displacements argument. The newly created datatype has a type map with $n \cdot$ 24 $\sum_{i=0}^{\text{count}-1} B[i]$ entries: 2526 $\{(type_0, disp_0 + D[0]), \dots, (type_{n-1}, disp_{n-1} + D[0]), \dots, \}$ 27 $(type_0, disp_0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex), \ldots,$ 28 29 30 $(type_{n-1}, disp_{n-1} + D[0] + (B[0] - 1) \cdot ex), \dots,$ 31 32 $(type_0, disp_0 + \mathsf{D}[\mathsf{count-1}]), \dots, (type_{n-1}, disp_{n-1} + \mathsf{D}[\mathsf{count-1}]), \dots,$ 33 34 $(type_0, disp_0 + \mathsf{D}[\mathsf{count-1}] + (\mathsf{B}[\mathsf{count-1}] - 1) \cdot ex), \ldots,$ 35 $(type_{n-1}, disp_{n-1} + \mathsf{D}[\mathsf{count-1}] + (\mathsf{B}[\mathsf{count-1}] - 1) \cdot ex)\}.$ 36 37 38

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.

Unofficial Draft for Comment Only

39

40

41

```
1
     MPI_TYPE_CREATE_INDEXED_BLOCK(count, blocklength, array_of_displacements,
\mathbf{2}
                    oldtype, newtype)
3
       IN
                                              length of array of displacements (non-negative
                 count
4
                                              integer)
5
                 blocklength
       IN
                                             size of block (non-negative integer)
6
7
       IN
                 array_of_displacements
                                             array of displacements (array of integers)
8
       IN
                 oldtype
                                             old datatype (handle)
9
       OUT
                                             new datatype (handle)
                 newtype
10
11
     C binding
12
     int MPI_Type_create_indexed_block(int count, int blocklength,
13
                     const int array_of_displacements[], MPI_Datatype oldtype,
14
15
                    MPI_Datatype *newtype)
16
     Fortran 2008 binding
17
     MPI_Type_create_indexed_block(count, blocklength, array_of_displacements,
18
                     oldtype, newtype, ierror)
19
          INTEGER, INTENT(IN) :: count, blocklength,
20
                     array_of_displacements(count)
21
          TYPE(MPI_Datatype), INTENT(IN) :: oldtype
22
          TYPE(MPI_Datatype), INTENT(OUT) :: newtype
23
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
25
     Fortran binding
26
     MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,
27
                    OLDTYPE, NEWTYPE, IERROR)
          INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDTYPE,
28
                     NEWTYPE, IERROR
29
30
^{31}
     Hindexed_block The function MPI_TYPE_CREATE_HINDEXED_BLOCK is identical to
32
     MPI_TYPE_CREATE_INDEXED_BLOCK, except that block displacements in
33
     array_of_displacements are specified in bytes, rather than in multiples of the oldtype extent.
34
35
36
     MPI_TYPE_CREATE_HINDEXED_BLOCK(count, blocklength, array_of_displacements,
37
                     oldtype, newtype)
38
       IN
                 count
                                              length of array of displacements (non-negative
39
                                             integer)
40
41
       IN
                 blocklength
                                             size of block (non-negative integer)
42
       IN
                 array_of_displacements
                                             byte displacement of each block (array of integers)
43
       IN
                 oldtype
                                             old datatype (handle)
44
45
       OUT
                 newtype
                                             new datatype (handle)
46
47
     C binding
48
```

int MP]	[_Type_create_hindexed_blo	ck(int count, int blocklength,	1
		y_of_displacements[], MPI_Datatype oldtype,	2
	MPI_Datatype *newty	pe)	3
Fortrai	n 2008 binding		4 5
	_	ount, blocklength, array_of_displacements,	6
	oldtype, newtype, i	error)	7
	TEGER, INTENT(IN) :: count	-	8
INT	TEGER(KIND=MPI_ADDRESS_KIN		9
	array_of_displacem		10
	PE(MPI_Datatype), INTENT(I	• -	11
	PE(MPI_Datatype), INTENT(O TEGER, OPTIONAL, INTENT(OU		12
TNI	LEGER, OPIIONAL, INIENI(00		13
	n binding		14
MPI_TYF		OUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,	15 16
T 110	OLDTYPE, NEWTYPE, I		10
	TEGER COUNT, BLOCKLENGTH,	DLDIYPE, NEWIYPE, IERROR D) ARRAY_OF_DISPLACEMENTS(*)	18
TNI	LEGER (KIND-MPI_ADDRESS_KIN	D) ARRAI_OF_DISPLACEMENTS(*)	19
			20
		is the most general type constructor. It further	21
-		DEXED in that it allows each block to consist of repli-	22
cations	of different datatypes.		23
			24
MPI_TY	PE_CREATE_STRUCT(count.	array_of_blocklengths, array_of_displacements,	25
-	array_of_types, newtyp		26
IN	count	number of blocks also number of entries in arrays	27 28
		array_of_types, array_of_displacements, and	28 29
		array_of_blocklengths (non-negative integer)	30
IN	array_of_blocklengths	number of elements in each block (array of	31
	anay_or_blocklengths	non-negative integers)	32
IN	array_of_displacements	byte displacement of each block (array of integers)	33
			34
IN	array_of_types	types of elements in each block (array of handles)	35 36
OUT	newtype	new datatype (handle)	37
<u> </u>			38
C bind	U U	ount count int owners of blocklangths[]	39
int MPI	• •	<pre>ount, const int array_of_blocklengths[], y_of_displacements[],</pre>	40
		array_of_types[], MPI_Datatype *newtype)	41
		array_or_types[], in r_batatype *newtype)	42
	n 2008 binding		43
мьт_дай	pe_create_struct(count, ar	•	44
ד אדי		ents, array_of_types, newtype, ierror)	45
	TEGER, INTENI(IN) :: count TEGER(KIND=MPI_ADDRESS_KIN	, array_of_blocklengths(count)	46
T 11]	array_of_displacem		47 48
	array_or_urspracem		40

Unofficial Draft for Comment Only

1 2 3	TYPE(MPI_Datatype), INTENT(IN) :: array_of_types(count) TYPE(MPI_Datatype), INTENT(OUT) :: newtype INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4 5 7 8 9 10	<pre>Fortran binding MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,</pre>
11 12	Example 4.6 Let type1 have type map,
13 14	$\{(\texttt{double}, 0), (\texttt{char}, 8)\},\$
15 16 17 18 19	with extent 16. Let $B = (2, 1, 3)$, $D = (0, 16, 26)$, and $T = (MPI_FLOAT, type1, MPI_CHAR)$. Then a call to MPI_TYPE_CREATE_STRUCT(3, B, D, T, newtype) returns a datatype with type map, {(float, 0), (float, 4), (double, 16), (char, 24), (char, 26), (char, 27), (char, 28)}.
20 21 22	That is, two copies of MPI_FLOAT starting at 0, followed by one copy of type1 starting at 16, followed by three copies of MPI_CHAR, starting at 26. (We assume that a float occupies four bytes.)
23 24	In general, let T be the array_of_types argument, where T[i] is a handle to,
25 26	$typemap_i = \{(type_0^i, disp_0^i), \dots, (type_{n_i-1}^i, disp_{n_i-1}^i)\},\$
27 28 29	with extent ex_i . Let B be the array_of_blocklength argument and D be the array_of_displacements argument. Let c be the count argument. Then the newly created datatype has a type map with $\sum_{i=0}^{c-1} B[i] \cdot n_i$ entries:
30 31	$\{(type_0^0, disp_0^0 + D[0]), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0]), \dots, \}$
32 33	$(type_0^0, disp_0^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots,$
34 35	$(type_0^{C-1}, disp_0^{C-1} + D[c-1]), \dots, (type_{n_{C-1}-1}^{C-1}, disp_{n_{C-1}-1}^{C-1} + D[c-1]), \dots,$
36 37	$(type_0^{C-1}, disp_0^{C-1} + D[c-1] + (B[c-1] - 1) \cdot ex_{C-1}), \dots,$
38	$(type_{n_{C-1}-1}^{C-1}, disp_{n_{C-1}-1}^{C-1} + D[c-1] + (B[c-1]-1) \cdot ex_{C-1})\}.$
39 40	A call to MPI_TYPE_CREATE_HINDEXED(count, B, D, oldtype, newtype) is equivalent
41 42 43	to a call to MPI_TYPE_CREATE_STRUCT(count, B, D, T, newtype), where each entry of T is equal to oldtype.
$44 \\ 45$	
46	
47 48	

4.1. DERIVED DATATYPES

4.1.3 Subarray Datatype Constructor

MPI_TYPE_CREATE_SUBARRAY(ndims, array_of_sizes, array_of_subsizes, array_of_starts, order, oldtype, newtype)

	,		6
IN	ndims	number of array dimensions (positive integer)	7
IN	array_of_sizes	number of elements of type $oldtype$ in each dimension	8
		of the full array (array of positive integers)	9
IN	array_of_subsizes	number of elements of type oldtype in each dimension	10
	-	of the subarray (array of positive integers)	11
IN	array_of_starts	starting coordinates of the subarray in each	12
		dimension (array of non-negative integers)	13
			14
IN	order	array storage order flag (state)	15
IN	oldtype	old datatype (handle)	16
OUT	newtype	new datatype (handle)	17
	51		18

C binding

int MPI_Type_	create_subarray(int ndims, const int array_of_sizes[],
	<pre>const int array_of_subsizes[], const int array_of_starts[],</pre>
	int order, MPI_Datatype oldtype, MPI_Datatype *newtype)

Fortran 2008 binding

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

Fortran binding

MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES, ARRAY_OF_SUBSIZES, ARRAY_OF_STARTS, ORDER, OLDTYPE, NEWTYPE, IERROR) INTEGER NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*), ARRAY_OF_STARTS(*), ORDER, OLDTYPE, NEWTYPE, IERROR

The subarray type constructor creates an MPI datatype describing an *n*-dimensional subarray of an n-dimensional array. The subarray may be situated anywhere within the full array, and may be of any nonzero size up to the size of the larger array as long as it is confined within this array. This type constructor facilitates creating filetypes to access arrays distributed in blocks among processes to a single file that contains the global array, see MPI I/O, especially Section 13.1.1.

This type constructor can handle arrays with an arbitrary number of dimensions and works for both C and Fortran ordered matrices (i.e., row-major or column-major). Note that a C program may use Fortran order and a Fortran program may use C order.

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

Unofficial Draft for Comment Only

```
1
                     The number of elements of type oldtype in each dimension of the n-dimensional ar-
\mathbf{2}
            ray and the requested subarray are specified by array_of_sizes and array_of_subsizes, re-
3
            spectively. For any dimension i, it is erroneous to specify array_of_subsizes[i] < 1 or
 4
            array_of_subsizes[i] > array_of_sizes[i].
5
                      The array_of_starts contains the starting coordinates of each dimension of the subarray.
6
            Arrays are assumed to be indexed starting from zero. For any dimension i, it is erroneous to
\overline{7}
            specify array_of_starts[i] < 0 or array_of_starts[i] > (array_of_sizes[i] - array_of_subsizes[i]).
 8
                        Advice to users. In a Fortran program with arrays indexed starting from 1, if the
9
                        starting coordinate of a particular dimension of the subarray is n, then the entry in
10
                        array_of_starts for that dimension is n-1. (End of advice to users.)
11
12
                      The order argument specifies the storage order for the subarray as well as the full array.
13
            It must be set to one of the following:
14
15
            MPI_ORDER_C The ordering used by C arrays, (i.e., row-major order)
16
17
            MPI_ORDER_FORTRAN The ordering used by Fortran arrays, (i.e., column-major order)
18
                      A ndims-dimensional subarray (newtype) with no extra padding can be defined by the
19
            function Subarray() as follows:
20
21
                        newtype = Subarray(ndims, {size_0, size_1, \ldots, size_{ndims-1}},
22
                                                    \{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\},\
23
                                                     \{start_0, start_1, \dots, start_{ndims-1}\}, oldtype\}
^{24}
25
                     Let the typemap of oldtype have the form:
26
                        \{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\}
27
28
            where type_i is a predefined MPI datatype, and let e_x be the extent of oldtype. Then we define
29
            the Subarray() function recursively using the following three equations. Equation 4.2 defines
30
            the base step. Equation 4.3 defines the recursion step when order = MPI_ORDER_FORTRAN,
^{31}
            and Equation 4.4 defines the recursion step when order = MPI_ORDER_C. These equations
32
            use the conceptual datatypes lb_marker and ub_marker; see Section 4.1.6 for details.
33
34
35
                        Subarray(1, \{size_0\}, \{subsize_0\}, \{start_0\}, \{start_
                                                                                                                                                                                                (4.2)
36
                                          \{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\}\}
37
38
                                 \{(\mathsf{lb}_\mathsf{marker}, 0),
                            =
39
                                    (type_0, disp_0 + start_0 \times ex), \ldots, (type_{n-1}, disp_{n-1} + start_0 \times ex),
40
                                     (tupe_0, disp_0 + (start_0 + 1) \times ex), \ldots, (tupe_{n-1}), \ldots, (tupe_{n-1})
41
                                                    disp_{n-1} + (start_0 + 1) \times ex), \ldots
42
                                     (tupe_0, disp_0 + (start_0 + subsize_0 - 1) \times ex), \ldots,
43
44
                                                    (type_{n-1}, disp_{n-1} + (start_0 + subsize_0 - 1) \times ex),
45
                                     (ub_marker, size_0 \times ex)
46
47
                        Subarray(ndims, {size_0, size_1, \ldots, size_{ndims-1}},
48
                                                                                                                                                                                                (4.3)
```

$\{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\},\$	1
$\{start_0, start_1, \dots, start_{ndims-1}\}, oldtype)$	2
= Subarray($ndims - 1, \{size_1, size_2, \dots, size_{ndims-1}\},\$	3
$\{subsize_1, subsize_2, \dots, subsize_{ndims-1}\},\$	4 5
$\{start_1, start_2, \dots, start_{ndims-1}\},\$	6
	7
$Subarray(1, \{size_0\}, \{subsize_0\}, \{start_0\}, oldtype))$	
	8
Subarray($ndims$, { $size_0, size_1, \dots, size_{ndims-1}$ }, (4.4)	9
	10
$\{subsize_0, subsize_1, \dots, subsize_{ndims-1}\},\$	11
$\{start_0, start_1, \dots, start_{ndims-1}\}, oldtype)$	12
= Subarray($ndims - 1, \{size_0, size_1, \dots, size_{ndims-2}\},\$	13
	14
$\{subsize_0, subsize_1, \dots, subsize_{ndims-2}\},$	15
$\{start_0, start_1, \dots, start_{ndims-2}\},\$	16
$Subarray(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, oldtype))$	17
	18

For an example use of MPI_TYPE_CREATE_SUBARRAY in the context of I/O see Section 13.9.2.

4.1.4 Distributed Array Datatype Constructor

The distributed array type constructor supports HPF-like [44] data distributions. However, unlike in HPF, the storage order may be specified for C arrays as well as for Fortran arrays.

Advice to users. One can create an HPF-like file view using this type constructor as follows. Complementary filetypes are created by having every process of a group call this constructor with identical arguments (with the exception of rank which should be set appropriately). These filetypes (along with identical disp and etype) are then used to define the view (via MPI_FILE_SET_VIEW), see MPI I/O, especially Section 13.1.1 and Section 13.3. Using this view, a collective data access operation (with identical offsets) will yield an HPF-like distribution pattern. (*End of advice to users.*)

 24

 $25 \\ 26$

 31

```
1
     MPI_TYPE_CREATE_DARRAY(size, rank, ndims, array_of_gsizes, array_of_distribs,
\mathbf{2}
                     array_of_dargs, array_of_psizes, order, oldtype, newtype)
3
       IN
                 size
                                              size of process group (positive integer)
4
       IN
                 rank
                                              rank in process group (non-negative integer)
5
6
       IN
                 ndims
                                              number of array dimensions as well as process grid
7
                                              dimensions (positive integer)
8
       IN
                 array_of_gsizes
                                              number of elements of type oldtype in each dimension
9
                                              of global array (array of positive integers)
10
       IN
                 array_of_distribs
                                              distribution of array in each dimension (array of
11
                                              states)
12
13
       IN
                 array_of_dargs
                                              distribution argument in each dimension (array of
14
                                              positive integers)
15
       IN
                 array_of_psizes
                                              size of process grid in each dimension (array of
16
                                              positive integers)
17
       IN
                 order
                                              array storage order flag (state)
18
                                              old datatype (handle)
19
       IN
                 oldtype
20
       OUT
                 newtype
                                              new datatype (handle)
21
22
     C binding
23
     int MPI_Type_create_darray(int size, int rank, int ndims,
24
                     const int array_of_gsizes[], const int array_of_distribs[],
25
                     const int array_of_dargs[], const int array_of_psizes[],
26
                     int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
27
     Fortran 2008 binding
28
     MPI_Type_create_darray(size, rank, ndims, array_of_gsizes,
29
30
                     array_of_distribs, array_of_dargs, array_of_psizes, order,
^{31}
                     oldtype, newtype, ierror)
          INTEGER, INTENT(IN) :: size, rank, ndims, array_of_gsizes(ndims),
32
33
                      array_of_distribs(ndims), array_of_dargs(ndims),
34
                      array_of_psizes(ndims), order
          TYPE(MPI_Datatype), INTENT(IN) :: oldtype
35
          TYPE(MPI_Datatype), INTENT(OUT) :: newtype
36
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
     Fortran binding
39
     MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS, ARRAY_OF_GSIZES,
40
                     ARRAY_OF_DISTRIBS, ARRAY_OF_DARGS, ARRAY_OF_PSIZES, ORDER,
41
                     OLDTYPE, NEWTYPE, IERROR)
42
          INTEGER SIZE, RANK, NDIMS, ARRAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*),
43
                      ARRAY_OF_DARGS(*), ARRAY_OF_PSIZES(*), ORDER, OLDTYPE,
44
                      NEWTYPE, IERROR
45
46
          MPI_TYPE_CREATE_DARRAY can be used to generate the datatypes corresponding
47
     to the distribution of an ndims-dimensional array of oldtype elements onto an
48
     ndims-dimensional grid of logical processes. Unused dimensions of array_of_psizes should be
```

CHAPTER 4. DATATYPES

set to 1. (See Example 4.7.) For a call to MPI_TYPE_CREATE_DARRAY to be correct, the equation $\prod_{i=0}^{ndims-1} array_of_psizes[i] = size$ must be satisfied. The ordering of processes in the process grid is assumed to be row-major, as in the case of virtual Cartesian process topologies.

Advice to users. For both Fortran and C arrays, the ordering of processes in the process grid is assumed to be row-major. This is consistent with the ordering used in virtual Cartesian process topologies in MPI. To create such virtual process topologies, or to find the coordinates of a process in the process grid, etc., users may use the corresponding process topology functions, see Chapter 7. (*End of advice to users.*)

Each dimension of the array can be distributed in one of three ways:

- MPI_DISTRIBUTE_BLOCK Block distribution
- MPI_DISTRIBUTE_CYCLIC Cyclic distribution
- MPI_DISTRIBUTE_NONE Dimension not distributed.

The constant MPI_DISTRIBUTE_DFLT_DARG specifies a default distribution argument. The distribution argument for a dimension that is not distributed is ignored. For any dimension i in which the distribution is MPI_DISTRIBUTE_BLOCK, it is erroneous to specify array_of_dargs[i] * array_of_psizes[i] < array_of_gsizes[i].

For example, the HPF layout ARRAY(CYCLIC(15)) corresponds to MPI_DISTRIBUTE_CYCLIC with a distribution argument of 15, and the HPF layout AR-RAY(BLOCK) corresponds to MPI_DISTRIBUTE_BLOCK with a distribution argument of MPI_DISTRIBUTE_DFLT_DARG.

The order argument is used as in MPI_TYPE_CREATE_SUBARRAY to specify the storage order. Therefore, arrays described by this type constructor may be stored in Fortran (column-major) or C (row-major) order. Valid values for order are MPI_ORDER_FORTRAN and MPI_ORDER_C.

This routine creates a new MPI datatype with a typemap defined in terms of a function called "cyclic()" (see below).

Without loss of generality, it suffices to define the typemap for the MPI_DISTRIBUTE_CYCLIC case where MPI_DISTRIBUTE_DFLT_DARG is not used.

MPI_DISTRIBUTE_BLOCK and MPI_DISTRIBUTE_NONE can be reduced to the MPI_DISTRIBUTE_CYCLIC case for dimension i as follows.

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

```
(array_of_gsizes[i] + array_of_psizes[i] - 1)/array_of_psizes[i].
```

If array_of_dargs[i] is not MPI_DISTRIBUTE_DFLT_DARG, then MPI_DISTRIBUTE_BLOCK and MPI_DISTRIBUTE_CYCLIC are equivalent.

MPI_DISTRIBUTE_NONE is equivalent to MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] set to array_of_gsizes[i].

Finally, MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] equal to44MPI_DISTRIBUTE_DFLT_DARG is equivalent to MPI_DISTRIBUTE_CYCLIC with45array_of_dargs[i] set to 1.46

For MPI_ORDER_FORTRAN, an ndims-dimensional distributed array (newtype) is defined ⁴⁷ by the following code fragment: ⁴⁸

1

2

3 4

5

6

7

8

9

10 11

12

13 14

15 16

17

18

19

20

21

22

23

 24

25

26

27

28

29

30

31

32

33

34

35

36

37 38

39

40 41

42

```
1
          oldtypes[0] = oldtype;
\mathbf{2}
          for (i = 0; i < ndims; i++) {</pre>
3
                oldtypes[i+1] = cyclic(array_of_dargs[i],
4
                                            array_of_gsizes[i],
5
                                            r[i],
6
                                            array_of_psizes[i],
7
                                            oldtypes[i]);
8
          }
9
          newtype = oldtypes[ndims];
10
11
          For MPI_ORDER_C, the code is:
12
          oldtypes[0] = oldtype;
13
          for (i = 0; i < ndims; i++) {</pre>
14
                oldtypes[i + 1] = cyclic(array_of_dargs[ndims - i - 1],
15
                                               array_of_gsizes[ndims - i - 1],
16
                                              r[ndims - i - 1],
17
                                               array_of_psizes[ndims - i - 1],
18
                                               oldtypes[i]);
19
          }
20
          newtype = oldtypes[ndims];
21
22
23
      where r[i] is the position of the process (with rank rank) in the process grid at dimension i.
24
      The values of r[i] are given by the following code fragment:
25
26
          t_rank = rank;
27
          t_size = 1;
28
          for (i = 0; i < ndims; i++)
29
                t_size *= array_of_psizes[i];
30
          for (i = 0; i < ndims; i++) {</pre>
31
                t_size = t_size / array_of_psizes[i];
32
               r[i] = t_rank / t_size;
33
               t_rank = t_rank % t_size;
34
          }
35
36
      Let the typemap of oldtype have the form:
37
38
            \{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\}
39
      where type_i is a predefined MPI datatype, and let ex be the extent of
40
      oldtype. The following function uses the conceptual datatypes lb_marker and ub_marker, see
41
      Section 4.1.6 for details.
42
          Given the above, the function cyclic() is defined as follows:
43
44
           cyclic(darg, gsize, r, psize, oldtype)
45
              = {(lb_marker, 0),
46
                  (type_0, disp_0 + r \times darg \times ex), \ldots,
47
48
                         (type_{n-1}, disp_{n-1} + r \times darg \times ex),
```

$(type_0, disp_0 + (r \times darg + 1) \times ex), \dots,$	1
$(type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex),$	2
	3
$(type_0, disp_0 + ((r+1) \times darg - 1) \times ex), \ldots,$	4 5
$(type_{n-1}, disp_{n-1} + ((r+1) \times darq - 1) \times ex),$	6
$(-\partial \mathbf{r} \cdot \mathbf{n} - 1) \cdots \mathbf{r} \cdot \mathbf{n} - 1 \cdot ((\mathbf{r} \cdot \mathbf{r}) - \mathbf{n} \cdot 1))$	7
$(type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex), \ldots,$	8
$(type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex),$	9
$(type_{n-1}, utsp_{n-1} + t) \times dut g \times cx + psize \times dut g \times cx),$ $(type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex), \dots,$	10
	11 12
$(type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex + psize \times darg \times ex),$	12
	14
$(type_0, disp_0 + ((r+1) \times darg - 1) \times ex + psize \times darg \times ex), \dots,$	15
$(type_{n-1}, disp_{n-1} + ((r+1) \times darg - 1) \times ex + psize \times darg \times ex),$	16
	17
$(type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex \times (count - 1)), \dots,$	18
$(type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex \times (count - 1)),$	19 20
$(type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex \times (count - 1)), \dots,$	20
$(type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex$	22
$(vgpe_{n-1}, uvsp_{n-1} + (r \times uurg + 1) \times cx) + psize \times darg \times ex \times (count - 1)),$	23
$+psize \times uarg \times ex \times (count - 1)),$	24
	25
$(type_0, disp_0 + (r \times darg + darg_{last} - 1) \times ex$	26
$+psize imes darg imes ex imes (count-1)), \ldots,$	27 28
$(type_{n-1}, disp_{n-1} + (r \times darg + darg_{last} - 1) \times ex$	20
+psize imes darg imes ex imes (count-1)),	30
$(ub_marker, gsize * ex) \}$	31
where <i>count</i> is defined by this code fragment:	32
	33
<pre>nblocks = (gsize + (darg - 1)) / darg;</pre>	34
<pre>count = nblocks / psize; left_over = nblocks - count * psize;</pre>	35 36
if (r < left_over)	37
count = count + 1;	38
	39
Here, <i>nblocks</i> is the number of blocks that must be distributed among the processors.	40
Finally, $darg_{last}$ is defined by this code fragment:	41
if ((num_in_last_cyclic = gsize % (psize * darg)) == 0)	42
<pre>darg_last = darg;</pre>	43

```
darg_last = darg;
                                                                                  44
else {
                                                                                  45
    darg_last = num_in_last_cyclic - darg * r;
    if (darg_last > darg)
                                                                                  46
        darg_last = darg;
                                                                                  47
    if (darg_last <= 0)</pre>
                                                                                  48
```

Unofficial Draft for Comment Only

```
1
                  darg_last = darg;
\mathbf{2}
              }
3
4
     Example 4.7 Consider generating the filetypes corresponding to the HPF distribution:
5
6
            <oldtype> FILEARRAY(100, 200, 300)
7
     !HPF$ PROCESSORS PROCESSES(2, 3)
8
     !HPF$ DISTRIBUTE FILEARRAY(CYCLIC(10), *, BLOCK) ONTO PROCESSES
9
     This can be achieved by the following Fortran code, assuming there will be six processes
10
     attached to the run:
11
12
     ndims = 3
13
     array_of_gsizes(1) = 100
14
     array_of_distribs(1) = MPI_DISTRIBUTE_CYCLIC
15
     array_of_dargs(1) = 10
16
     array_of_gsizes(2) = 200
17
     array_of_distribs(2) = MPI_DISTRIBUTE_NONE
18
     \operatorname{array_of_dargs}(2) = 0
19
     array_of_gsizes(3) = 300
20
     array_of_distribs(3) = MPI_DISTRIBUTE_BLOCK
21
     array_of_dargs(3) = MPI_DISTRIBUTE_DFLT_DARG
22
     array_of_psizes(1) = 2
23
     array_of_psizes(2) = 1
24
     array_of_psizes(3) = 3
25
     call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
26
     call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
27
     call MPI_TYPE_CREATE_DARRAY(size, rank, ndims, array_of_gsizes, &
28
           array_of_distribs, array_of_dargs, array_of_psizes,
                                                                            &
29
           MPI_ORDER_FORTRAN, oldtype, newtype, ierr)
30
```

³¹ ₃₂ 4.1.5 Ad

4.1.5 Address and Size Functions

The displacements in a general datatype are relative to some initial buffer address. Absolute addresses can be substituted for these displacements: we treat them as displacements relative to "address zero," the start of the address space. This initial address zero is indicated by the constant MPI_BOTTOM. Thus, a datatype can specify the absolute address of the entries in the communication buffer, in which case the buf argument is passed the value MPI_BOTTOM. Note that in Fortran MPI_BOTTOM is not usable for initialization or assignment, see Section 2.5.4.

The address of a location in memory can be found by invoking the function MPI_GET_ADDRESS. The **relative displacement** between two absolute addresses can

⁴² be calculated with the function MPI_AINT_DIFF. A new absolute address as sum of an ⁴³ absolute base address and a relative displacement can be calculated with the function

⁴⁴ MPI_AINT_ADD. To ensure portability, arithmetic on absolute addresses should not be ⁴⁵ performed with the intrinsic operators "-" and "+". See also Sections 2.5.6 and 4.1.12 on ⁴⁶ pages 16 and 123.

47 48

MPI_GET_ADDRESS(location, address)

location

address

Rationale.

rationale.)

IN

OUT

C binding

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 location in caller memory (choice) address of location (integer) int MPI_Get_address(const void *location, MPI_Aint *address)

Fortran 2008 binding

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

Fortran binding

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

INTEGER IERROR

Returns the (byte) address of location.

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

Example 4.8 Using MPI_GET_ADDRESS for an array.

```
REAL A(100,100)
INTEGER(KIND=MPI_ADDRESS_KIND) I1, I2, DIFF
CALL MPI_GET_ADDRESS(A(1,1), I1, IERROR)
CALL MPI_GET_ADDRESS(A(10,10), I2, IERROR)
DIFF = MPI_AINT_DIFF(I2, I1)
! The value of DIFF is 909*sizeofreal; the values of I1 and I2 are
! implementation dependent.
```

Advice to users. C users may be tempted to avoid the usage of 43 44MPI_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 4546that the value of a pointer (or the pointer cast to int) be the absolute address of the 47object pointed at — although this is commonly the case. Furthermore, referencing 48 may not have a unique definition on machines with a segmented address space. The

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

56 7

9

10 11

1213

14

15

16

17

18

19

2021

22

23

 24

25

26

2728

29

30

3132 33

34 35

36

37

38

39

40

41

1	us	e of MPI_GET_A	DDRESS to "reference" C variables guarantees portability to such
2			End of advice to users.)
3		.	
4		dvice to users.	To prevent problems with the argument copying and register
5 6	-	8.1.20. (End of ad	by Fortran compilers, please note the hints in Sections $18.1.10-$
7	10	5.1.20. (End of ad	wice to users.)
8	То	ensure portability	y, arithmetic on MPI addresses must be performed using the
9	MPI_AII	NT_ADD and MF	PI_AINT_DIFF functions.
10			
11	ΜΡΙ ΔΙΙ	NT_ADD(base, di	sn)
12			
13	IN	base	base address (integer)
14 15	IN	disp	displacement (integer)
16			
17	C bind	•	
18	MP1_A1r	it MPI_Aint_add	A(MPI_Aint base, MPI_Aint disp)
19	Fortrar	n 2008 binding	
20			RESS_KIND) MPI_Aint_add(base, disp)
21	INT	TEGER(KIND=MPI_	ADDRESS_KIND), INTENT(IN) :: base, disp
22	Fortrar	n binding	
23 24	INTEGEF	R(KIND=MPI_ADDR	RESS_KIND) MPI_AINT_ADD(BASE, DISP)
25	INT	TEGER(KIND=MPI_	ADDRESS_KIND) BASE, DISP
26	MP	AINT_ADD pro	oduces a new MPI_Aint value that is equivalent to the sum of
27		-	ents, where base represents a base address returned by a call to
28	MPI_GE	T_ADDRESS and	d disp represents a signed integer displacement. The resulting ad-
29			process that generated $base,$ and it must correspond to a location
30			nced by $base$, as described in Section 4.1.12. The addition is per-
31			results in the correct MPI_Aint representation of the output address,
32 33	as if the	e process that ong	ginally produced base had called:
34	MPI_Get	_address((char	*) base + disp, &result);
35			
36			
37	MPI_AII	NT_DIFF(addr1, a	addr2)
38	IN	addr1	minuend address (integer)
39	IN	addr2	
40	IIN	auurz	subtrahend address (integer)
41 42	C bind	ina	
43		•	f(MPI_Aint addr1, MPI_Aint addr2)
44			
45		n 2008 binding	DECC KIND) MDI Aist diff(add-1 - dd-0)
46			RESS_KIND) MPI_Aint_diff(addr1, addr2) _ADDRESS_KIND), INTENT(IN) :: addr1, addr2
47			ADDIGDO_NIND, INIENI(IN/ AUUII, AUUIZ
48	Fortrai	n binding	

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

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

The following auxiliary functions provide useful information on derived datatypes.

MPI_TYPE_SIZE(datatype, size) IN datatype datatype (handle) OUT size datatype size (integer) C binding int MPI_Type_size(MPI_Datatype datatype, int *size) Fortran 2008 binding MPI_Type_size(datatype, size, ierror) TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(OUT) :: size INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR) INTEGER DATATYPE, SIZE, IERROR MPI_TYPE_SIZE_X(datatype, size) IN datatype datatype (handle) OUT datatype size (integer) size C binding int MPI_Type_size_x(MPI_Datatype datatype, MPI_Count *size) Fortran 2008 binding MPI_Type_size_x(datatype, size, ierror) TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_TYPE_SIZE_X(DATATYPE, SIZE, IERROR) INTEGER DATATYPE, IERROR

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

1 $\mathbf{2}$

3

4

5

6

7

8 9

10

INTEGER(KIND=MPI_COUNT_KIND) SIZE

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

Lower-Bound and Upper-Bound Markers 4.1.6

11It is often convenient to define explicitly the lower bound and upper bound of a type map, 12and override the definition given on page 112. This allows one to define a datatype that has 13"holes" at its beginning or its end, or a datatype with entries that extend above the upper 14bound or below the lower bound. Examples of such usage are provided in Section 4.1.14. 15Also, the user may want to overide the alignment rules that are used to compute upper 16bounds and extents. E.g., a C compiler may allow the user to overide default alignment 17rules for some of the structures within a program. The user has to specify explicitly the 18 bounds of the datatypes that match these structures.

19To achieve this, we add two additional conceptual datatypes, **lb_marker** and 20**ub_marker**, that represent the lower bound and upper bound of a datatype. These con-21ceptual datatypes occupy no space $(extent(lb_marker) = extent(ub_marker) = 0)$. They do 22not affect the size or count of a datatype, and do not affect the content of a message created 23with this datatype. However, they do affect the definition of the extent of a datatype and, 24 therefore, affect the outcome of a replication of this datatype by a datatype constructor. 25

26**Example 4.9** A call to MPI_TYPE_CREATE_RESIZED(MPI_INT, -3, 9, type1) creates a 27new datatype that has an extent of 9 (from -3 to 5, 5 included), and contains an integer 28at displacement 0. This is the datatype defined by the typemap $\{(\mathsf{lb}_\mathsf{marker}, -3), (int, 0), \}$ 29(ub_marker, 6)}. If this type is replicated twice by a call to MPI_TYPE_CONTIGUOUS(2, 30 type1, type2) then the newly created type can be described by the typemap {(lb_marker, 31 -3), (int, 0), (int,9), (ub_marker, 15)}. (An entry of type ub_marker can be deleted if there 32 is another entry of type ub_marker with a higher displacement; an entry of type lb_marker 33 can be deleted if there is another entry of type lb_marker with a lower displacement.) 34In general, if

35 36 37

38

39 40 41

43

4445

48

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

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

$$lb(Typemap) = \begin{cases} \min_{j} disp_{j} & \text{if no entry has type} \\ \min_{j} \{ disp_{j} \text{ such that } type_{j} = \mathsf{lb_marker} \} & \text{otherwise} \end{cases}$$

Similarly, the **upper bound** of *Typemap* is defined to be 42

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

46 Then 47

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

Unofficial Draft for Comment Only

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.)

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

Example: Instead of

TYPE, BIND(C) :: my_data
REAL, DIMENSION(3) :: x
! there may be a gap of the size of one REAL
! if the alignment of a DOUBLE PRECISION is
! two times the size of a REAL
DOUBLE PRECISION :: p
END TYPE
one should define
TYPE, BIND(C) :: my_data
REAL, DIMENSION(3) :: x
REAL :: gap1
DOUBLE PRECISION :: p
END TYPE

 $\overline{7}$

 24

1 2 3 4 5 6	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.		
7 8 9 10 11	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 18.1.15. (End of advice to users.)		
12 13 14	4.1.7 Ex	xtent and Bounds of Datatype	s
15 16	MPI_TYP	PE_GET_EXTENT(datatype, lb,	extent)
17	IN	datatype	datatype to get information on (handle)
18 19	OUT	lb	lower bound of datatype (integer)
19 20	OUT	extent	extent of datatype (integer)
21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38	Fortran MPI_Type TYPE INTE INTE Fortran MPI_TYPE INTE INTE	Type_get_extent(MPI_Datat; MPI_Aint *extent) 2008 binding e_get_extent(datatype, lb, (MPI_Datatype), INTENT(IN GER(KIND=MPI_ADDRESS_KIND GER, OPTIONAL, INTENT(OUT) :: datatype), INTENT(OUT) :: lb, extent) :: ierror EXTENT, IERROR)) LB, EXTENT
39	IN	datatype	datatype to get information on (handle)
40	OUT	lb	lower bound of datatype (integer)
41 42	OUT	extent	extent of datatype (integer)
43 44 45 46	C bindir int MPI_	0	atype datatype, MPI_Count *1b,
47 48		2008 binding get_extent_x(datatype, 1	b, extent, ierror)

TYPE(MPI_Datatype), INTENT(IN) :: datatype	
<pre>INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: I</pre>	lb, extent
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	

Fortran binding

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

Returns the lower bound and the extent of datatype (as defined in Equation 4.1). For both functions, if either OUT parameter cannot express the value to be returned (e.g., if the parameter is too small to hold the output value), it is set to MPI_UNDEFINED.

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

MPI_TYPE_CREATE_RESIZED(oldtype, lb, extent, newtype)

IN old	ltype	input datatype (handle)
IN lb		new lower bound of datatype (integer)
IN ext	ent	new extent of datatype (integer)
OUT nev	wtype	output datatype (handle)

C binding

<pre>int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb,</pre>	
MPI_Aint extent, MPI_Datatype *newtype)	

Fortran 2008 binding

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

Fortran binding

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

Returns in newtype a handle to a new datatype that is identical to oldtype, except that the lower bound of this new datatype is set to be lb, and its upper bound is set to be lb+ extent. Any previous lb and ub markers are erased, and a new pair of lower bound and upper bound markers are put in the positions indicated by the lb and extent arguments. This affects the behavior of the datatype when used in communication operations, with count > 1, and when used in the construction of new derived datatypes.

```
4.1.8 True Extent of Datatypes
\mathbf{2}
     Suppose we implement gather (see also Section 5.5) as a spanning tree implemented on
3
     top of point-to-point routines. Since the receive buffer is only valid on the root pro-
4
     cess, one will need to allocate some temporary space for receiving data on intermedi-
5
     ate nodes. However, the datatype extent cannot be used as an estimate of the amount
6
     of space that needs to be allocated, if the user has modified the extent, for example
7
     by using MPI_TYPE_CREATE_RESIZED. The functions MPI_TYPE_GET_TRUE_EXTENT
8
     and MPI_TYPE_GET_TRUE_EXTENT_X are provided which return the true extent of the
9
     datatype.
10
11
12
     MPI_TYPE_GET_TRUE_EXTENT(datatype, true_lb, true_extent)
13
       IN
                datatype
                                            datatype to get information on (handle)
14
                true_lb
                                            true lower bound of datatype (integer)
       OUT
15
16
       OUT
                                            true size of datatype (integer)
                true_extent
17
18
     C binding
19
     int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb,
20
                    MPI_Aint *true_extent)
21
22
     Fortran 2008 binding
23
     MPI_Type_get_true_extent(datatype, true_lb, true_extent, ierror)
^{24}
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: true_lb, true_extent
25
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     Fortran binding
28
     MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
29
         INTEGER DATATYPE, IERROR
30
         INTEGER(KIND=MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT
^{31}
32
33
     MPI_TYPE_GET_TRUE_EXTENT_X(datatype, true_lb, true_extent)
34
35
       IN
                datatype
                                            datatype to get information on (handle)
36
       OUT
                true_lb
                                            true lower bound of datatype (integer)
37
       OUT
                true_extent
                                            true size of datatype (integer)
38
39
     C binding
40
41
     int MPI_Type_get_true_extent_x(MPI_Datatype datatype, MPI_Count *true_lb,
42
                    MPI_Count *true_extent)
43
     Fortran 2008 binding
44
     MPI_Type_get_true_extent_x(datatype, true_lb, true_extent, ierror)
45
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
46
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: true_lb, true_extent
```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

1

47

Fortran binding

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

true_lb returns the offset of the lowest unit of store which is addressed by the datatype, i.e., the lower bound of the corresponding typemap, ignoring explicit lower bound markers. true_extent returns the true size of the datatype, i.e., the extent of the corresponding typemap, ignoring explicit lower bound and upper bound markers, and performing no rounding for alignment. If the typemap associated with datatype is

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

Then

$$true_lb(Typemap) = min_i \{ disp_i : type_i \neq lb_marker, ub_marker \},$$

$$true_ub(Typemap) = max_i \{ disp_i + sizeof(type_i) : type_i \neq \mathsf{lb_marker}, \mathsf{ub_marker} \},\$$

and

$$true_extent(Typemap) = true_ub(Typemap) - true_lb(typemap).$$

(Readers should compare this with the definitions in Section 4.1.6 and Section 4.1.7, which describe the function MPI_TYPE_GET_EXTENT.)

The true_extent is the minimum number of bytes of memory necessary to hold a datatype, uncompressed.

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

Commit and Free 4.1.9

A datatype object has to be **committed** before it can be used in a communication. As an argument in datatype constructors, uncommitted and also committed datatypes can be used. There is no need to commit basic datatypes. They are "pre-committed."

MPI_TYPE_COMMIT(datatype)	35
INOUT datatype datat	ype that is committed (handle) 36
	37 37
C binding	38
int MPI_Type_commit(MPI_Datatype *datat	39 39
int in i_iype_commit(in i_batatype *datat	40 40
Fortran 2008 binding	41
<pre>MPI_Type_commit(datatype, ierror)</pre>	42
TYPE(MPI_Datatype), INTENT(INOUT) :	: datatype 43
INTEGER, OPTIONAL, INTENT(OUT) :: i	error 44
Fortron hinding	45
Fortran binding MPI_TYPE_COMMIT(DATATYPE, IERROR)	46
	47
INTEGER DATATYPE, IERROR	48

 $\mathbf{2}$

 24

 The commit operation commits the datatype, that is, the formal description of a communication buffer, not the content of that buffer. Thus, after a datatype has been committed, it can be repeatedly reused to communicate the changing content of a buffer or, indeed,
 the content of different buffers, with different starting addresses.

Advice to implementors. The system may "compile" at commit time an internal representation for the datatype that facilitates communication, e.g., change from a compacted representation to a flat representation of the datatype, and select the most convenient transfer mechanism. (*End of advice to implementors.*)

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

¹⁴ **Example 4.10** The following code fragment gives examples of using MPI_TYPE_COMMIT.

```
15
     INTEGER type1, type2
16
     CALL MPI_TYPE_CONTIGUOUS(5, MPI_REAL, type1, ierr)
17
                     ! new type object created
18
     CALL MPI_TYPE_COMMIT(type1, ierr)
19
                     ! now type1 can be used for communication
20
     type2 = type1
21
                     ! type2 can be used for communication
22
                     ! (it is a handle to same object as type1)
23
     CALL MPI_TYPE_VECTOR(3, 5, 4, MPI_REAL, type1, ierr)
^{24}
                     ! new uncommitted type object created
25
     CALL MPI_TYPE_COMMIT(type1, ierr)
26
                    ! now type1 can be used anew for communication
27
28
29
     MPI_TYPE_FREE(datatype)
30
^{31}
       INOUT
                datatype
                                           datatype that is freed (handle)
32
33
     C binding
34
     int MPI_Type_free(MPI_Datatype *datatype)
35
     Fortran 2008 binding
36
     MPI_Type_free(datatype, ierror)
37
         TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     Fortran binding
41
     MPI_TYPE_FREE(DATATYPE, IERROR)
42
         INTEGER DATATYPE, IERROR
43
         Marks the datatype object associated with datatype for deallocation and sets datatype
44
45
```

to MPI_DATATYPE_NULL. Any communication that is currently using this datatype will complete normally. Freeing a datatype does not affect any other datatype that was built from the freed datatype. The system behaves as if input datatype arguments to derived datatype constructors are passed by value.

Unofficial Draft for Comment Only

5

6

7

8

9 10

11

4.1. DERIVED DATATYPES

Advice to implementors. The implementation may keep a reference count of active communications that use the datatype, in order to decide when to free it. Also, one may implement constructors of derived datatypes so that they keep pointers to their datatype arguments, rather then copying them. In this case, one needs to keep track of active datatype definition references in order to know when a datatype object can be freed. (*End of advice to implementors.*)

4.1.10 Duplicating a Datatype

MPI_TYPE_DUP is a type constructor which duplicates the existing oldtype with associated key values. For each key value, the respective copy callback function determines the attribute value associated with this key in the new communicator; one particular action that a copy callback may take is to delete the attribute from the new datatype. Returns in newtype a new datatype with exactly the same properties as oldtype and any copied cached information, see Section 6.7.4. The new datatype has identical upper bound and lower bound and yields the same net result when fully decoded with the functions in Section 4.1.13. The newtype has the same committed state as the old oldtype.

4.1.11 Use of General Datatypes in Communication

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

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

 $\mathbf{2}$

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 $addr_{i,j} = \text{buf} + extent \cdot i + disp_j$ 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,

17 18 19

16

 $\{(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 4.11 This example shows that type matching is defined in terms of the basic types that a derived type consists of.

```
32
     . . .
33
     CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, type2, ...)
34
     CALL MPI_TYPE_CONTIGUOUS(4, MPI_REAL, type4, ...)
35
     CALL MPI_TYPE_CONTIGUOUS(2, type2, type22, ...)
36
     . . .
37
     CALL MPI_SEND(a, 4, MPI_REAL, ...)
38
     CALL MPI_SEND(a, 2, type2, ...)
39
     CALL MPI_SEND(a, 1, type22, ...)
40
     CALL MPI_SEND(a, 1, type4, ...)
41
     . . .
42
     CALL MPI_RECV(a, 4, MPI_REAL, ...)
43
     CALL MPI_RECV(a, 2, type2, ...)
^{44}
     CALL MPI_RECV(a, 1, type22, ...)
45
     CALL MPI_RECV(a, 1, type4, ...)
46
     Each of the sends matches any of the receives.
47
48
```

1

 $\mathbf{2}$

3

4

5

6 7

8

9

10

A datatype may specify overlapping entries. The use of such a datatype in a receive operation is erroneous. (This is erroneous even if the actual message received is short enough not to write any entry more than once.)

Suppose that MPI_RECV(buf, count, datatype, dest, tag, comm, status) is executed, where datatype has type map,

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

The received message need not fill all the receive buffer, nor does it need to fill a number of locations which is a multiple of n. Any number, k, of basic elements can be received, where $0 \le k \le \text{count} \cdot n$. The number of basic elements received can be retrieved from status using the query functions MPI_GET_ELEMENTS or MPI_GET_ELEMENTS_X.

MPI_TYPE_GET_ELEMENTS(status, datatype, count)

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

C binding

```
Fortran 2008 binding
MPI_Type_get_elements(status, datatype, count, ierror)
   TYPE(MPI_Status), INTENT(IN) :: status
   TYPE(MPI_Datatype), INTENT(IN) :: datatype
   INTEGER, INTENT(OUT) :: count
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

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

MPI_TYPE_GET_ELEMENTS_X(status, datatype, count)

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

```
C binding
```

Fortran 2008 binding
MPI_Type_get_elements_x(status, datatype, count, ierror)
 TYPE(MPI_Status), INTENT(IN) :: status
 TYPE(MPI_Datatype), INTENT(IN) :: datatype

Unofficial Draft for Comment Only

 $\mathbf{2}$

 24

1	INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: count
2	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3	Fortran binding
4	MPI_TYPE_GET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)
5	INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR
6	
7	INTEGER(KIND=MPI_COUNT_KIND) COUNT
8	The datatype argument should match the argument provided by the receive call that
9	set the status variable. For both functions, if the OUT parameter cannot express the value
10	to be returned (e.g., if the parameter is too small to hold the output value), it is set to
11	MPI_UNDEFINED.
12	The previously defined function MPI_GET_COUNT (Section 3.2.5), has a different be-
13	havior. It returns the number of "top-level entries" received, i.e. the number of "copies" of
14	type datatype. In the previous example, MPI_GET_COUNT may return any integer value
15	k, where $0 \le k \le \text{count}$. If MPI_GET_COUNT returns k, then the number of basic elements
16	received (and the value returned by MPI_GET_ELEMENTS or MPI_GET_ELEMENTS_X) is
17	$n \cdot k$. If the number of basic elements received is not a multiple of n, that is, if the receive
18	operation has not received an integral number of datatype "copies," then MPI_GET_COUNT
19	sets the value of count to MPI_UNDEFINED.
20	
21	Example 4.12 Usage of MPI_GET_COUNT and MPI_GET_ELEMENTS.
22	
23	CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, Type2, ierr)
24	CALL MPI_TYPE_COMMIT(Type2, ierr)
25	····
26	CALL MPI_COMM_RANK(comm, rank, ierr)
27	IF (rank.EQ.0) THEN
28	CALL MPI_SEND(a, 2, MPI_REAL, 1, 0, comm, ierr)
29 30	CALL MPI_SEND(a, 3, MPI_REAL, 1, 0, comm, ierr)
31	ELSE IF (rank.EQ.1) THEN
32	CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
33	CALL MPI_GET_COUNT(stat, Type2, i, ierr) ! returns i=1
34	CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr) ! returns i=2
35	CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
36	CALL MPI_GET_COUNT(stat, Type2, i, ierr)
37	CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr) ! returns i=3
38	END IF
39	
40	The functions MPI_GET_ELEMENTS and MPI_GET_ELEMENTS_X can also be used
41	after a probe to find the number of elements in the probed message. Note that the
42	MPI_GET_COUNT, MPI_GET_ELEMENTS, and MPI_GET_ELEMENTS_X return the same
43	values when they are used with basic datatypes as long as the limits of their respective
44	count arguments are not exceeded.
45	<i>Rationale.</i> The extension given to the definition of MPI_GET_COUNT seems natural:
46	one would expect this function to return the value of the count argument, when the
47	receive buffer is filled. Sometimes datatype represents a basic unit of data one wants
48	to transfer, for example, a record in an array of records (structures). One should be

Unofficial Draft for Comment Only

able to find out how many components were received without bothering to divide by the number of elements in each component. However, on other occasions, datatype is used to define a complex layout of data in the receiver memory, and does not represent a basic unit of data for transfers. In such cases, one needs to use the function MPI_GET_ELEMENTS or MPI_GET_ELEMENTS_X. (*End of rationale.*)

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

4.1.12 Correct Use of Addresses

Successively declared variables in C or Fortran are not necessarily stored at contiguous locations. Thus, care must be exercised that displacements do not cross from one variable to another. Also, in machines with a segmented address space, addresses are not unique and address arithmetic has some peculiar properties. Thus, the use of **addresses**, that is, displacements relative to the start address MPI_BOTTOM, has to be restricted.

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

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

A correct program uses only valid addresses to identify the locations of entries in communication buffers. Furthermore, if u and v are two valid addresses, then the (integer) difference u - v can be computed only if both u and v are in the same sequential storage. No other arithmetic operations can be meaningfully 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 detect45erroneous, "out of bound" displacements — unless those overflow the user address46space — since the MPI call may not know the extent of the arrays and records in the47host program.(End of advice to users.)48

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\overline{7}$

 24

 31

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

4.1.13 Decoding a Datatype

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

18 MPI_TYPE_GET_ENVELOPE(datatype, num_integers, num_addresses, num_datatypes, 19 combiner) 20IN datatype to access (handle) datatype 2122 OUT number of input integers used in call constructing num_integers 23combiner (non-negative integer) 24OUT number of input addresses used in call constructing num_addresses 25combiner (non-negative integer) 26OUT num_datatypes number of input datatypes used in call constructing 27combiner (non-negative integer) 2829 OUT combiner combiner (state) 30 31 C binding 32 int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers, 33 int *num_addresses, int *num_datatypes, int *combiner) 34 Fortran 2008 binding 35 MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes, 36 combiner, ierror) 37 TYPE(MPI_Datatype), INTENT(IN) :: datatype 38 INTEGER, INTENT(OUT) :: num_integers, num_addresses, num_datatypes, 39 combiner 40 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 41 42Fortran binding 43 MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, 44 COMBINER, IERROR) 45INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER, 46 IERROR 47 48

1

2

3

4

5

6 7

8

9

10

11

12

13

14

15

For the given datatype, MPI_TYPE_GET_ENVELOPE returns information on the number and type of input arguments used in the call that created the datatype. The number-of-arguments values returned can be used to provide sufficiently large arrays in the decoding routine MPI_TYPE_GET_CONTENTS. This call and the meaning of the returned values is described below. The combiner reflects the MPI datatype constructor call that was used in creating datatype.

Rationale. By requiring that the **combiner** reflect the constructor used in the creation of the **datatype**, the decoded information can be used to effectively recreate the calling sequence used in the original creation. This is the most useful information and was felt to be reasonable even though it constrains implementations to remember the original constructor sequence even if the internal representation is different.

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

The list in Table 4.1 has the values that can be returned in combiner on the left and the call associated with them on the right.

MPI_COMBINER_NAMED	a named predefined datatype	21
MPI_COMBINER_DUP	MPI_TYPE_DUP	22
MPI_COMBINER_CONTIGUOUS	MPI_TYPE_CONTIGUOUS	23
MPI_COMBINER_VECTOR	MPI_TYPE_VECTOR	24
MPI_COMBINER_HVECTOR	MPI_TYPE_CREATE_HVECTOR	25
MPI_COMBINER_INDEXED	MPI_TYPE_INDEXED	26
MPI_COMBINER_HINDEXED	MPI_TYPE_CREATE_HINDEXED	27
MPI_COMBINER_INDEXED_BLOCK	MPI_TYPE_CREATE_INDEXED_BLOCK	28
MPI_COMBINER_HINDEXED_BLOCK	MPI_TYPE_CREATE_HINDEXED_BLOCK	29
MPI_COMBINER_STRUCT	MPI_TYPE_CREATE_STRUCT	30
MPI_COMBINER_SUBARRAY	MPI_TYPE_CREATE_SUBARRAY	31
MPI_COMBINER_DARRAY	MPI_TYPE_CREATE_DARRAY	32
MPI_COMBINER_F90_REAL	MPI_TYPE_CREATE_F90_REAL	33
MPI_COMBINER_F90_COMPLEX	MPI_TYPE_CREATE_F90_COMPLEX	34
MPI_COMBINER_F90_INTEGER	MPI_TYPE_CREATE_F90_INTEGER	35
MPI_COMBINER_RESIZED	MPI_TYPE_CREATE_RESIZED	36
		37
Table 4.1: combiner values return	rned from MPI_TYPE_GET_ENVELOPE	38
Table 4.1. Combiner values letur		39
If combiner is MPI_COMBINER_NAME	D then datatype is a named predefined datatype.	40
	reation call for a datatype can be obtained using	41
MPI_TYPE_GET_CONTENTS.		42

 $\mathbf{5}$

```
1
     MPI_TYPE_GET_CONTENTS(datatype, max_integers, max_addresses, max_datatypes,
\mathbf{2}
                    array_of_integers, array_of_addresses, array_of_datatypes)
3
       IN
                 datatype
                                             datatype to access (handle)
4
       IN
                 max_integers
                                             number of elements in array_of_integers
5
                                             (non-negative integer)
6
7
       IN
                 max_addresses
                                             number of elements in array_of_addresses
8
                                             (non-negative integer)
9
       IN
                 max_datatypes
                                             number of elements in array_of_datatypes
10
                                             (non-negative integer)
11
       OUT
                 array_of_integers
                                             contains integer arguments used in constructing
12
                                             datatype (array of integers)
13
       OUT
                                             contains address arguments used in constructing
14
                 array_of_addresses
                                             datatype (array of integers)
15
16
       OUT
                                             contains datatype arguments used in constructing
                 array_of_datatypes
17
                                             datatype (array of handles)
18
19
     C binding
20
     int MPI_Type_get_contents(MPI_Datatype datatype, int max_integers,
21
                    int max_addresses, int max_datatypes, int array_of_integers[],
22
                    MPI_Aint array_of_addresses[],
23
                    MPI_Datatype array_of_datatypes[])
^{24}
25
     Fortran 2008 binding
26
     MPI_Type_get_contents(datatype, max_integers, max_addresses, max_datatypes,
27
                    array_of_integers, array_of_addresses, array_of_datatypes,
                    ierror)
28
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
          INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes
30
          INTEGER, INTENT(OUT) :: array_of_integers(max_integers)
31
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::
32
33
                     array_of_addresses(max_addresses)
34
         TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes)
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
     Fortran binding
37
     MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,
38
                    ARRAY_OF_INTEGERS, ARRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES,
39
                    IERROR)
40
          INTEGER DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,
41
                     ARRAY_OF_INTEGERS(*), ARRAY_OF_DATATYPES(*), IERROR
42
          INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_ADDRESSES(*)
43
44
          datatype must be a predefined unnamed or a derived datatype; the call is erroneous if
45
     datatype is a predefined named datatype.
46
          The values given for max_integers, max_addresses, and max_datatypes must be at least as
47
     large as the value returned in num_integers, num_addresses, and num_datatypes, respectively,
48
     in the call MPI_TYPE_GET_ENVELOPE for the same datatype argument.
```

CHAPTER 4. DATATYPES

Rationale. The arguments max_integers, max_addresses, and max_datatypes allow for error checking in the call. (*End of rationale.*)

The datatypes returned in array_of_datatypes are handles to datatype objects that are equivalent to the datatypes used in the original construction call. If these were derived datatypes, then the returned datatypes are new datatype objects, and the user is responsible for freeing these datatypes with MPI_TYPE_FREE. If these were predefined datatypes, then the returned datatype is equal to that (constant) predefined datatype and cannot be freed.

The committed state of returned derived datatypes is undefined, i.e., the datatypes may or may not be committed. Furthermore, the content of attributes of returned datatypes is undefined.

Note that MPI_TYPE_GET_CONTENTS can be invoked with a datatype argument that was constructed using MPI_TYPE_CREATE_F90_REAL, MPI_TYPE_CREATE_F90_INTEGER, or MPI_TYPE_CREATE_F90_COMPLEX (an unnamed predefined datatype). In such a case, an empty array_of_datatypes is returned.

Rationale. The definition of datatype equivalence implies that equivalent predefined datatypes are equal. By requiring the same handle for named predefined datatypes, it is possible to use the == or .EQ. comparison operator to determine the datatype involved. (*End of rationale.*)

Advice to implementors. The datatypes returned in array_of_datatypes must appear to the user as if each is an equivalent copy of the datatype used in the type constructor call. Whether this is done by creating a new datatype or via another mechanism such as a reference count mechanism is up to the implementation as long as the semantics are preserved. (*End of advice to implementors.*)

Rationale. The committed state and attributes of the returned datatype is deliberately left vague. The datatype used in the original construction may have been modified since its use in the constructor call. Attributes can be added, removed, or modified as well as having the datatype committed. The semantics given allow for a reference count implementation without having to track these changes. (*End of rationale.*)

In the deprecated datatype constructor calls, the address arguments in Fortran are of type INTEGER. In the preferred calls, the address arguments are of type INTEGER(KIND=MPI_ADDRESS_KIND). The call MPI_TYPE_GET_CONTENTS returns all addresses in an argument of type INTEGER(KIND=MPI_ADDRESS_KIND). This is true even if the deprecated calls were used. Thus, the location of values returned can be thought of as being returned by the C bindings. It can also be determined by examining the preferred calls for datatype constructors for the deprecated calls that involve addresses.

Rationale. By having all address arguments returned in the array_of_addresses argument, the result from a C and Fortran decoding of a datatype gives the result in the same argument. It is assumed that an integer of type INTEGER(KIND=MPI_ADDRESS_KIND) will be at least as large as the INTEGER argument used in datatype construction with the old MPI-1 calls so no loss of information will occur. (End of rationale.)

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

45

46

47

1 The following defines what values are placed in each entry of the returned arrays $\mathbf{2}$ depending on the datatype constructor used for datatype. It also specifies the size of the 3 arrays needed which is the values returned by MPI_TYPE_GET_ENVELOPE. In Fortran, 4 the following calls were made: 5PARAMETER (LARGE = 1000) 6 INTEGER TYPE, NI, NA, ND, COMBINER, I(LARGE), D(LARGE), IERROR 7 INTEGER (KIND=MPI_ADDRESS_KIND) A(LARGE) 8 ! CONSTRUCT DATATYPE TYPE (NOT SHOWN) 9 CALL MPI_TYPE_GET_ENVELOPE(TYPE, NI, NA, ND, COMBINER, IERROR) 10 IF ((NI .GT. LARGE) .OR. (NA .GT. LARGE) .OR. (ND .GT. LARGE)) THEN 11 WRITE (*, *) "NI, NA, OR ND = ", NI, NA, ND, & 12" RETURNED BY MPI_TYPE_GET_ENVELOPE IS LARGER THAN LARGE = ", LARGE 13 CALL MPI_ABORT(MPI_COMM_WORLD, 99, IERROR) 14ENDIF 15CALL MPI_TYPE_GET_CONTENTS(TYPE, NI, NA, ND, I, A, D, IERROR) 1617 or in C the analogous calls of: 18 19#define LARGE 1000 20int ni, na, nd, combiner, i[LARGE]; 21MPI_Aint a[LARGE]; 22MPI_Datatype type, d[LARGE]; 23/* construct datatype type (not shown) */ 24 MPI_Type_get_envelope(type, &ni, &na, &nd, &combiner); 25if ((ni > LARGE) || (na > LARGE) || (nd > LARGE)) { 26fprintf(stderr, "ni, na, or nd = %d %d %d returned by ", ni, na, nd); 27fprintf(stderr, "MPI_Type_get_envelope is larger than LARGE = %d\n", 28LARGE); 29 MPI_Abort(MPI_COMM_WORLD, 99); 30 }; 31MPI_Type_get_contents(type, ni, na, nd, i, a, d); 32 In the descriptions that follow, the lower case name of arguments is used. 33 If combiner is MPI_COMBINER_NAMED then it is erroneous to call 34 MPI_TYPE_GET_CONTENTS. 35If combiner is MPI_COMBINER_DUP then 36 37 Constructor argument С Fortran location 38 oldtype d[0]D(1)39 40and ni = 0, na = 0, nd = 1. 41 If combiner is MPI_COMBINER_CONTIGUOUS then 4243 Constructor argument С Fortran location 44 i[0]I(1)count 45d[0]D(1)oldtype 46and ni = 1, na = 0, nd = 1. 47 If combiner is MPI_COMBINER_VECTOR then 48

and ni =

-	Constructor argument	С	Fortran location
_	count	i[0]	I(1)
	blocklength	i[1]	I(2)
	stride	i[2]	I(3)
_	oldtype	d[0]	D(1)
and $ni = 3$, $na = 0$, n If combiner is MF	d = 1. PI_COMBINER_HVECTOR	then	
_	Constructor argument	С	Fortran location

		0.0			
	count		i[0]	I(1)	
	blocklength		i[1]	I(2)	
	stride		a[0]	A(1)	
	oldtype		d[0]	D(1)	
i = 2, na $= 1$, combiner is N		R_INDEXED t	hen		
Constructo	r argument	\mathbf{C}		Fortran	location
count		i[0]		I	(1)
array_of_b	locklengths	i[1] to i[i[0]] i[i[0]+1] to i[2*i[0]]		I(2) to I	I(I(1)+1)
array_of_d	isplacements			I(I(1)+2) to	I(2*I(1)+1)
oldtype		d[0		D	(1)

and $ni = 2^*count+1$, na = 0, nd = 1.

If combiner is $MPI_COMBINER_HINDEXED$ then

Constructor argument	С	Fortran location
count	i[0]	I(1)
$array_of_blocklengths$	i[1] to $i[i[0]]$	I(2) to $I(I(1)+1)$
array_of_displacements	a[0] to a[i[0]-1]	A(1) to $A(I(1))$
oldtype	d[0]	$\mathrm{D}(1)$

and ni = count+1, na = count, nd = 1.

If combiner is MPI_COMBINER_INDEXED_BLOCK then

Constructor argument	С	Fortran location
count	i[0]	I(1)
blocklength	i[1]	I(2)
array_of_displacements	i[2] to $i[i[0]+1]$	I(3) to $I(I(1)+2)$
oldtype	d[0]	D(1)

and ni = count+2, na = 0, nd = 1.

If combiner is $MPI_COMBINER_HINDEXED_BLOCK$ then

Constructor argument	С	Fortran location
count	i[0]	I(1)
blocklength	i[1]	I(2)
$array_of_displacements$	a[0] to $a[i[0]-1]$	A(1) to $A(I(1))$
oldtype	d[0]	D(1)

Unofficial Draft for Comment Only

 24

 44

_	Constructor	roumont	С		Fortran loca	ation
-	Constructor a count	rgument	i[0]		$\frac{\text{Fortran locs}}{\text{I}(1)}$	au1011
	array_of_bloc	klongthe	i[1] to i[[[0]]	I(1) I(2) to I(I(1)	$) \perp 1)$
	array_of_disp	0			A(1) to A(1)	, ,
	array_of_type		d[0] to $d[i]$		D(1) to $D(1)$	
-	<u>airaj _0r_0j po</u>			[0] 1]	D(1) 00 D(<u>()</u>
and $ni = count$,	,				
If combine	r is MPI_COME	BINER_SUB	ARRAY the	n		
Constru	istor argumon	+	С		Fortran l	oastion
ndims	uctor argumen	U	i[0]		I(1	
array_c	of sizes	;[1]	1[0] to $i[i[0]]$		I(1) I(2) to I(/
-	of_subsizes		1] to $i[2*i[0]]$		I(2) to $I(1)$	
v	of_starts		+1 to $i[3*i]$		$2^{I(1)+2}$ to $2^{I(1)+2}$ to	
order	-200103		3*i[0]+1]	[0]] 1(2	I(1)+2 to I(3*I(1)	
oldtype		٦[و	d[0]		D(1)	· -
	·		alol		- D (-	-)
and $ni = 3*ndi$	ms+2, na = 0	, nd = 1.				
If combine	r is MPI_COME	BINER_DAR	RAY then			
	ctor argument		C			location
size			ilO			(1)
,			i[0]			
rank			i[1]		I	(2)
ndims			i[1] i[2]		I (I	$(2) \\ (3)$
ndims array_of			i[1] i[2] to $i[i[2]+2]$	u ol	I I I(4) to I	(2) (3) I(I(3)+3)
ndims array_of array_of	_distribs	i[i[2]+3]	$i[1] \\ i[2] \\ to i[i[2]+2] \\ to i[2*i[2]-2] $	-	I(4) to I $I(4) to I$ $I(I(3)+4) to$	(2) (3) I(I(3)+3) D I(2*I(3)+3)
ndims array_of array_of array_of	_distribs _dargs	i[i[2]+3] i[2*i[2]+3]	$i[1] \\ i[2] \\ to i[i[2]+2] \\ to i[2^*i[2]-3] to i[3^*i[2] \\ i[3] to i[3^*i[2] \\ i[3] to i[3^*i[2] \\ i[3] to i[3^*i[2] \\ i[3] to i[3^*i[3] \\ i[3] to i[3^*i[3^*i[3] \\ i[3^*i[3] \\ i[3^*i[3^*i[3] \\ i[3^*i[3^*i[3] \\ i[3^*i[3^*i[3^*i[3] \\ i[3^*i[3^*i[3^*i[3] \\ i[3^*i[3^*i[3^*i[3^*i[3] \\ i[3^*i[3^*i[3^*i[3^*i[3^*i[3^*i[3^*i[3^*$]+2] I	I(4) to II(1(3)+4) to(2*I(3)+4) to	(2) (3) I(I(3)+3) D I(2*I(3)+3) to $I(3*I(3)-3)$
ndims array_of array_of array_of array_of	_distribs _dargs	i[i[2]+3] i[2*i[2]+3] i[3*i[2]+3]	i[1] i[2] to i[i[2]+2] to i[2*i[2]- 3] to i[3*i[2] 3] to i[4*i[2]]+2] I	I(4) to II(1(3)+4) to(2*I(3)+4) to(3*I(3)+4) to	(2) (3) I(I(3)+3) to $I(2*I(3)+3)$ to $I(2*I(3)-3)$ to $I(3*I(3)-3)$ to $I(4*I(3)-3)$
ndims array_of array_of array_of array_of order	_distribs _dargs	i[i[2]+3] i[2*i[2]+3] i[3*i[2]+3]	$ i[1] \\ i[2] \\ to i[i[2]+2] \\ to i[2^*i[2]-3] \\ to i[3^*i[2]-3] \\ to i[4^*i[2]+3] $]+2] I	I(4) to II(4) to II(I(3)+4) to(2*I(3)+4) to(3*I(3)+4) toI(4*I)	(2) (3) I(I(3)+3) D I(2*I(3)+4) to I(3*I(3)- to I(4*I(3)- (3)+4)
ndims array_of array_of array_of array_of	_distribs _dargs	i[i[2]+3] i[2*i[2]+3] i[3*i[2]+3]	i[1] i[2] to i[i[2]+2] to i[2*i[2]- 3] to i[3*i[2] 3] to i[4*i[2]]+2] I	I(4) to II(4) to II(I(3)+4) to(2*I(3)+4) to(3*I(3)+4) toI(4*I)	(2) (3) I(I(3)+3) to $I(2*I(3)+3)$ to $I(2*I(3)-3)$ to $I(3*I(3)-3)$ to $I(4*I(3)-3)$
ndims array_of array_of array_of order oldtype	_distribs _dargs _psizes	i[i[2]+3] i[2*i[2]+3] i[3*i[2]+3] i[3*i[2]+3] i[3*i[2]+3] i[4]	$ i[1] \\ i[2] \\ to i[i[2]+2] \\ to i[2^*i[2]-3] \\ to i[3^*i[2]-3] \\ to i[4^*i[2]+3] $]+2] I	I(4) to II(4) to II(I(3)+4) to(2*I(3)+4) to(3*I(3)+4) toI(4*I)	(2) (3) I(I(3)+3) D I(2*I(3)+4) to I(3*I(3)- to I(4*I(3)- (3)+4)
ndims array_of array_of array_of array_of order oldtype and ni = 4*ndi	_distribs _dargs _psizes ms+4, na = 0	i[i[2]+3] $i[2*i[2]+3]$ $i[3*i[2]+3]$ $i[3*i[2]+3]$ $i[4]$ $i[4]$ $nd = 1.$	$\begin{array}{c} {\rm i}[1]\\ {\rm i}[2]\\ {\rm to}\; {\rm i}[{\rm i}[2]{+}2]\\ {\rm to}\; {\rm i}[2^{*}{\rm i}[2]{-}3]\\ {\rm to}\; {\rm i}[3^{*}{\rm i}[2]{-}3]\\ {\rm j}\; {\rm to}\; {\rm i}[4^{*}{\rm i}[2]{+}3]\\ {\rm d}[0]\\ \end{array}$]+2] I	I(4) to II(4) to II(I(3)+4) to(2*I(3)+4) to(3*I(3)+4) toI(4*I)	(2) (3) I(I(3)+3) D I(2*I(3)+4) to I(3*I(3)- to I(4*I(3)- (3)+4)
ndims array_of array_of array_of array_of order oldtype and ni = 4*ndi	_distribs _dargs _psizes	i[i[2]+3] $i[2*i[2]+3]$ $i[3*i[2]+3]$ $i[3*i[2]+3]$ $i[4]$ $i[4]$ $nd = 1.$	$\begin{array}{c} {\rm i}[1]\\ {\rm i}[2]\\ {\rm to}\; {\rm i}[{\rm i}[2]{+}2]\\ {\rm to}\; {\rm i}[2^{*}{\rm i}[2]{-}3]\\ {\rm to}\; {\rm i}[3^{*}{\rm i}[2]{-}3]\\ {\rm j}\; {\rm to}\; {\rm i}[4^{*}{\rm i}[2]{+}3]\\ {\rm d}[0]\\ \end{array}$]+2] I	I(4) to II(4) to II(I(3)+4) to(2*I(3)+4) to(3*I(3)+4) toI(4*I)	(2) (3) I(I(3)+3) D I(2*I(3)+4) to I(3*I(3)- to I(4*I(3)- (3)+4)
ndims array_of array_of array_of array_of order oldtype and ni = 4*ndi	_distribs _dargs _psizes ms+4, na = 0 r is MPI_COME	i[i[2]+3] $i[2*i[2]+3]$ $i[3*i[2]+3]$ $i[3*i[2]+3]$ $i[4]$ $i[4]$ $nd = 1.$	i[1] i[2] to $i[i[2]+2]$ to $i[2^*i[2]-3]$ to $i[3^*i[2]-3]$ to $i[4^*i[2]+3]$ d[0] REAL then]+2] I]+2] I	I(4) to II(4) to II(I(3)+4) to(2*I(3)+4) to(3*I(3)+4) toI(4*I)	(2) (3) I(I(3)+3) D I(2*I(3)+4) to I(3*I(3)- to I(4*I(3)- (3)+4)
ndims array_of array_of array_of array_of order oldtype and ni = 4*ndi	_distribs _dargs _psizes ms+4, na = 0 r is MPI_COME	i[i[2]+3] i[2*i[2]+3] i[2*i[2]+3] i[3*i[2]+3] i[4] , nd = 1. BINER_F90_	i[1] i[2] to $i[i[2]+2]$ to $i[2^*i[2]-3]$ to $i[3^*i[2]-3]$ to $i[4^*i[2]+3]$ d[0] REAL then]+2] I]+2] I Fortra	I(4) to I I(4) to I I(1(3)+4) to (2*I(3)+4) to (3*I(3)+4) to I(4*I) D	(2) (3) I(I(3)+3) D I(2*I(3)+4) to I(3*I(3)- to I(4*I(3)- (3)+4)
ndims array_of array_of array_of array_of order oldtype and ni = 4*ndi	_distribs _dargs _psizes ms+4, na = 0 r is MPI_COME 	i[i[2]+3] i[2*i[2]+3] i[2*i[2]+3] i[3*i[2]+3] i[4] , nd = 1. BINER_F90_	i[1] i[2] to i[i[2]+2] to i[2*i[2]- 3] to i[3*i[2] 3] to i[4*i[2]+3] d[0] REAL then nent C]+2] I]+2] I Fortra	$\frac{I}{I(4) \text{ to } I}$ $I(4) \text{ to } I$ $I(I(3)+4) \text{ to } I$ $(2*I(3)+4) \text{ to } I$ $(3*I(3)+4) \text{ to } I$ $I(4*I)$ D $n \text{ location}$	(2) (3) I(I(3)+3) D I(2*I(3)+4) to I(3*I(3)- to I(4*I(3)- (3)+4)
ndims array_of array_of array_of order oldtype and ni = 4*ndi If combine	_distribs _dargs _psizes ms+4, na = 0 r is MPI_COME $\underline{Constru-p}$ r	i[i[2]+3] i[2*i[2]+3] i[2*i[2]+3] i[3*i[2]+3] i[4] , nd = 1. BINER_F90_	$\begin{array}{c} i[1] \\ i[2] \\ \text{to } i[i[2]+2] \\ \text{to } i[2^*i[2]-3] \\ \text{to } i[3^*i[2]-3] \\ \text{to } i[3^*i[2]-3] \\ \text{d} i[2]+3] \\ \text{d} [0] \\ \end{array}$ REAL then $\begin{array}{c} \text{ment } C \\ \hline i[0] \\ \end{array}$]+2] I]+2] I Fortra	I(4) I(4) I(4) I(4) I(4) I(4) I(4) I(4)	(2) (3) I(I(3)+3) D I(2*I(3)+4) to I(3*I(3)- to I(4*I(3)- (3)+4)
ndims array_of array_of array_of array_of order oldtype and ni = 4 *ndi If combine and ni = 2, na	_distribs _dargs _psizes ms+4, na = 0 r is MPI_COME $\overline{Constru-p}$ r = 0, nd = 0.	i[i[2]+3] i[2*i[2]+3 i[2*i[2]+3 i[2*i[2]+3 i[2*i[2]+3 i[2]+3 i[2]+3 i[2]+3 i[2]+3 i[2]+3 i[2]+3]	$i[1] \\i[2] \\to i[i[2]+2] \\to i[2*i[2]-3] \\to i[3*i[2]-3] \\to i[3*i[2]-3] \\d[0] \\REAL then \\nent C \\i[0] \\i[1]$]+2] I]+2] I Fortra	I(4) I(4) I(4) I(4) I(4) I(4) I(4) I(4)	(2) (3) I(I(3)+3) D I(2*I(3)+4) to I(3*I(3)- to I(4*I(3)- (3)+4)
ndims array_of array_of array_of array_of order oldtype and ni = 4 *ndi If combine and ni = 2, na	_distribs _dargs _psizes ms+4, na = 0 r is MPI_COME $\underline{Constru-p}$ r	i[i[2]+3] i[2*i[2]+3 i[2*i[2]+3 i[2*i[2]+3 i[2*i[2]+3 i[2]+3 i[2]+3 i[2]+3 i[2]+3 i[2]+3 i[2]+3]	$i[1] \\i[2] \\to i[i[2]+2] \\to i[2*i[2]-3] \\to i[3*i[2]-3] \\to i[3*i[2]-3] \\d[0] \\REAL then \\nent C \\i[0] \\i[1]$]+2] I]+2] I Fortra	I(4) I(4) I(4) I(4) I(4) I(4) I(4) I(4)	(2) (3) I(I(3)+3) D I(2*I(3)+4) to I(3*I(3)- to I(4*I(3)- (3)+4)
ndims array_of array_of array_of array_of order oldtype and ni = 4 *ndi If combine and ni = 2, na	_distribs _dargs _psizes ms+4, na = 0 r is MPI_COME $\underline{Constru-p}$ r = 0, nd = 0. r is MPI_COME	i[i[2]+3] i[2*i[2]+3 i[2*i[2]+3 i[3*i[2]+3 i[3*i[2]+3 i[4 , nd = 1. BINER_F90_ actor argun	$i[1] \\i[2] \\to i[i[2]+2] \\to i[2*i[2]-3] \\to i[3*i[2]-3] \\to i[3*i[2]-3] \\d[0] \\REAL then \\nent C \\i[0] \\i[1] \\COMPLEX \\$]+2] I]+2] I Fortra	$\frac{I}{I(4) \text{ to } I}$ $I(4) \text{ to } I$ $I(I(3)+4) \text{ to } (2*I(3)+4) \text{ to } (3*I(3)+4) \text{ to } (3*I(3)+4) \text{ to } I$ $I(4*I)$ $\frac{I}{I(4*I)}$ $\frac{I}{I(1)}$ $I(2)$	(2) (3) I(I(3)+3) D I(2*I(3)+4) to I(3*I(3)- to I(4*I(3)- (3)+4)
ndims array_of array_of array_of array_of order oldtype and ni = 4 *ndi If combine and ni = 2, na	_distribs _dargs _psizes ms+4, na = 0 r is MPI_COME $\overline{Constru-}$ r = 0, nd = 0. r is MPI_COME $\overline{Constru-}$	i[i[2]+3] i[2*i[2]+3 i[2*i[2]+3 i[2*i[2]+3 i[2*i[2]+3 i[2]+3 i[2]+3 i[2]+3 i[2]+3 i[2]+3 i[2]+3]	i[1] i[2] to i[i[2]+2] to i[2*i[2]-3] to i[3*i[2] 3] to i[3*i[2] 3] to i[4*i[2]+3] d[0] REAL then nent C i[0] i[1] COMPLEX]+2] I]+2] I Fortra then Fortra	I(4) I(4) I(4) I(4) I(4) I(4) I(4) I(4)	(2) (3) I(I(3)+3) D I(2*I(3)+4) to I(3*I(3)- to I(4*I(3)- (3)+4)
ndims array_of array_of array_of array_of order oldtype and ni = 4 *ndi If combine and ni = 2, na	_distribs _dargs _psizes ms+4, na = 0 r is MPI_COME $\underline{Constru-p}$ r = 0, nd = 0. r is MPI_COME	i[i[2]+3] i[2*i[2]+3 i[2*i[2]+3 i[3*i[2]+3 i[3*i[2]+3 i[4 , nd = 1. BINER_F90_ actor argun	$i[1] \\i[2] \\to i[i[2]+2] \\to i[2*i[2]-3] \\to i[3*i[2]-3] \\to i[3*i[2]-3] \\d[0] \\REAL then \\nent C \\i[0] \\i[1] \\COMPLEX \\$]+2] I]+2] I Fortra then Fortra	$\frac{I}{I(4) \text{ to } I}$ $I(4) \text{ to } I$ $I(I(3)+4) \text{ to } (2*I(3)+4) \text{ to } (3*I(3)+4) \text{ to } (3*I(3)+4) \text{ to } I$ $I(4*I)$ $\frac{1}{I(4*I)}$ $\frac{1}{I(1)}$ $I(2)$	(2) (3) I(I(3)+3) D I(2*I(3)+4) to I(3*I(3)- to I(4*I(3)- (3)+4)

	Constructor argument	С	Fortran locatio	m
	r	i[0]	I(1)	
-				
ni = 1, na = 0 If combiner is	0, nd = 0. MPI_COMBINER_RESIZED	then		
n combiner is		unen		
	Constructor argument	С	Fortran locati	on
	lb	a[0]	A(1)	
	extent	a[1]	A(2)	
	oldtype	d[0]	D(1)	
ni = 0 $na = 0$	2 nd = 1			
ni = 0, na = 2	2, 110 = 1.			
Examples				
ollowing exa	mples illustrate the use of	derive	d datatypes.	₩
nnle 4 13 S	end and receive a section	of a 3I) arrav	
		01 a 01	o array.	
100,100,	100), e(9,9,9)			•
CR oneslic	e, twoslice, threeslic	e, my	rank, ierr	
	PI_ADDRESS_KIND) 1b, s	sizeof	real	
GER status(MPI_STATUS_SIZE)			
	$a = \pm \frac{1}{2} = \frac{1}{2} (1 \cdot \frac{17}{2} \cdot 0) = \frac{2}{2} \cdot \frac{11}{2}$	0.10		
	ection a(1:17:2, 3:11, in e(:,:,:).	2:10		
id Store it	III e(.,.,.).	X		
MPI_COMM_R	ANK(MPI_COMM_WORLD, my	rank,	ierr)	
	,	,		
MPI_TYPE_G	ET_EXTENT(MPI_REAL, 1t	, siz	eofreal, ierr)	
	pe for a 1D section	_		
L MPI_TYPE_V	ECTOR(9, 1, 2, MPI_REA	L, on	eslice, ierr)	
	pe for a 2D section REATE_HVECTOR(9, 1, 10)()*ai~	enfreel once	icc
MP1_ITPE_C.			eoireai, onesi e, ierr)	.ICe
	U 1	OBITC	, 1911 <i>)</i>	
eate datatv	pe for the entire sect	ion		
• .	REATE_HVECTOR(9, 1, 10		*sizeofreal, t	wo
	threesli	.ce, i	err)	
	OMMIT(threeslice, ierr			
MPI_SENDRE	CV(a(1,3,2), 1, threes		v	
	MPI_REAL, myrank, (), MPI	_COMM_WORLD, s	ta
1. 414 (Converting (strictly) lower to	• • • •	, C ,	

Example 4.14 Copy the (strictly) lower triangular part of a matrix.

Unofficial Draft for Comment Only

```
1
     REAL a(100,100), b(100,100)
\mathbf{2}
     INTEGER disp(100), blocklen(100), ltype, myrank, ierr
3
     INTEGER status(MPI_STATUS_SIZE)
4
\mathbf{5}
     ! copy lower triangular part of array a
6
     ! onto lower triangular part of array b
7
8
     CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
9
10
     ! compute start and size of each column
^{11}
     DO i=1,100
12
        disp(i) = 100*(i-1) + i
13
        blocklen(i) = 100-i
14
     END DO
15
16
     ! create datatype for lower triangular part
17
     CALL MPI_TYPE_INDEXED(100, blocklen, disp, MPI_REAL, ltype, ierr)
18
19
     CALL MPI_TYPE_COMMIT(ltype, ierr)
20
     CALL MPI_SENDRECV(a, 1, ltype, myrank, 0, b, 1, &
21
                        ltype, myrank, 0, MPI_COMM_WORLD, status, ierr)
22
23
     Example 4.15 Transpose a matrix.
24
     REAL a(100,100), b(100,100)
25
     INTEGER row, xpose, myrank, ierr
26
     INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal
27
     INTEGER status(MPI_STATUS_SIZE)
28
29
     ! transpose matrix a onto b
30
31
     CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
32
33
     CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lb, sizeofreal, ierr)
34
35
     ! create datatype for one row
36
     CALL MPI_TYPE_VECTOR(100, 1, 100, MPI_REAL, row, ierr)
37
38
     ! create datatype for matrix in row-major order
39
     CALL MPI_TYPE_CREATE_HVECTOR(100, 1, sizeofreal, row, xpose, ierr)
40
41
     CALL MPI_TYPE_COMMIT(xpose, ierr)
42
43
     ! send matrix in row-major order and receive in column major order
44
     CALL MPI_SENDRECV(a, 1, xpose, myrank, 0, b, 100*100, &
45
                        MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
46
47
48
     Example 4.16 Another approach to the transpose problem:
```

4.1. DERIVED DATATYPES

```
1
REAL a(100,100), b(100,100)
                                                                                      \mathbf{2}
INTEGER row, row1
                                                                                      3
INTEGER (KIND=MPI_ADDRESS_KIND) disp(2), lb, sizeofreal
INTEGER myrank, ierr
                                                                                      4
INTEGER status(MPI_STATUS_SIZE)
                                                                                      5
                                                                                      6
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
                                                                                      9
! transpose matrix a onto b
                                                                                      10
                                                                                      11
CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lb, sizeofreal, ierr)
                                                                                      12
! create datatype for one row
                                                                                      13
                                                                                      14
CALL MPI_TYPE_VECTOR(100, 1, 100, MPI_REAL, row, ierr)
                                                                                      15
                                                                                      16
! create datatype for one row, with the extent of one real number
                                                                                      17
1b = 0
CALL MPI_TYPE_CREATE_RESIZED(row, lb, sizeofreal, row1, ierr)
                                                                                      18
                                                                                      19
CALL MPI_TYPE_COMMIT(row1, ierr)
                                                                                      20
                                                                                      21
! send 100 rows and receive in column major order
                                                                                      22
CALL MPI_SENDRECV(a, 100, row1, myrank, 0, b, 100*100, &
                                                                                      23
                                                                                      ^{24}
                   MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
                                                                                      25
                                                                                      26
Example 4.17 Use of MPI datatypes to manipulate an array of structures.
                                                                                      27
                                                                                      28
struct Partstruct
                                                                                      29
{
                                                                                      30
           type; /* particle type */
   int
                                                                                      31
   double d[6]; /* particle coordinates */
                                                                                      32
                   /* some additional information */
   char
          b[7];
                                                                                      33
};
                                                                                      34
                                                                                      35
struct Partstruct
                      particle[1000];
                                                                                      36
                                                                                      37
int
              i, dest, tag;
                                                                                      38
MPI_Comm
              comm;
                                                                                      39
                                                                                      40
                                                                                      41
/* build datatype describing structure */
                                                                                      42
MPI_Datatype Particlestruct, Particletype;
                                                                                      43
                                                                                      44
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
              blocklen[3] = \{1, 6, 7\};
                                                                                      45
int
                                                                                      46
MPI_Aint
              disp[3];
                                                                                      47
MPI_Aint
              base, lb, sizeofentry;
                                                                                      48
```

```
1
\mathbf{2}
     /* compute displacements of structure components */
3
4
     MPI_Get_address(particle, disp);
5
     MPI_Get_address(particle[0].d, disp+1);
6
     MPI_Get_address(particle[0].b, disp+2);
7
     base = disp[0];
8
     for (i=0; i < 3; i++) disp[i] = MPI_Aint_diff(disp[i], base);</pre>
9
10
     MPI_Type_create_struct(3, blocklen, disp, type, &Particlestruct);
11
12
     /* Since the compiler may pad the structure, it is best to explicitly
13
        set the extent of the MPI datatype for a structure element using
14
        MPI_Type_create_resized */
15
16
     /* compute extent of the structure */
17
     MPI_Get_address(particle+1, &sizeofentry);
18
     sizeofentry = MPI_Aint_diff(sizeofentry, base);
19
20
     /* build datatype describing structure */
21
     MPI_Type_create_resized(Particlestruct, 0, sizeofentry, &Particletype);
22
23
^{24}
     /* 4.1: send the entire array */
25
26
     MPI_Type_commit(&Particletype);
27
     MPI_Send(particle, 1000, Particletype, dest, tag, comm);
28
29
     /* 4.2: send only the entries of type zero particles,
30
^{31}
             preceded by the number of such entries */
32
33
     MPI_Datatype Zparticles;
                                  /* datatype describing all particles
34
                                     with type zero (needs to be recomputed
35
                                     if types change) */
36
     MPI_Datatype Ztype;
37
38
                   zdisp[1000];
     int
39
                   zblock[1000], j, k;
     int
40
                   zzblock[2] = \{1,1\};
     int
41
                   zzdisp[2];
     MPI_Aint
42
     MPI_Datatype zztype[2];
43
44
     /* compute displacements of type zero particles */
45
     j = 0;
46
     for (i=0; i < 1000; i++)</pre>
47
        if (particle[i].type == 0)
48
           {
```

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

Unofficial Draft for Comment Only

```
1
     /* an alternative solution to 4.3 */
\mathbf{2}
3
     MPI_Datatype Twodouble;
4
\mathbf{5}
     MPI_Type_contiguous(2, MPI_DOUBLE, &Twodouble);
6
7
     MPI_Datatype Onepair;
                               /* datatype for one pair of coordinates, with
8
                                  the extent of one particle entry */
9
10
     MPI_Type_create_resized(Twodouble, 0, sizeofentry, &Onepair );
^{11}
     MPI_Type_commit(&Onepair);
     MPI_Send(particle[0].d, 1000, Onepair, dest, tag, comm);
12
13
14
15
     Example 4.18 The same manipulations as in the previous example, but use absolute
16
     addresses in datatypes.
17
18
     struct Partstruct
19
     {
20
         int
                 type;
21
         double d[6];
22
         char
                 b[7];
23
     };
24
25
     struct Partstruct particle[1000];
26
27
     /* build datatype describing first array entry */
28
29
     MPI_Datatype Particletype;
30
     MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
31
                   block[3] = \{1, 6, 7\};
     int
32
     MPI_Aint
                   disp[3];
33
34
     MPI_Get_address(particle, disp);
35
     MPI_Get_address(particle[0].d, disp+1);
36
     MPI_Get_address(particle[0].b, disp+2);
37
     MPI_Type_create_struct(3, block, disp, type, &Particletype);
38
39
     /* Particletype describes first array entry -- using absolute
40
        addresses */
41
42
     /* 5.1: send the entire array */
43
44
     MPI_Type_commit(&Particletype);
45
     MPI_Send(MPI_BOTTOM, 1000, Particletype, dest, tag, comm);
46
47
48
```

```
1
/* 5.2: send the entries of type zero,
                                                                                        \mathbf{2}
         preceded by the number of such entries */
                                                                                        3
MPI_Datatype Zparticles, Ztype;
                                                                                        4
                                                                                        5
                                                                                        6
int
              zdisp[1000];
int
              zblock[1000], i, j, k;
                                                                                        7
              zzblock[2] = {1,1};
                                                                                         8
int
                                                                                        9
MPI_Datatype zztype[2];
                                                                                        10
              zzdisp[2];
MPI_Aint
                                                                                        11
j=0;
                                                                                        12
for (i=0; i < 1000; i++)
                                                                                        13
    if (particle[i].type == 0)
                                                                                        14
                                                                                        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++;
             i = k;
                                                                                        20
                                                                                        21
         }
MPI_Type_indexed(j, zblock, zdisp, Particletype, &Zparticles);
                                                                                        22
/* Zparticles describe particles with type zero, using
                                                                                        23
                                                                                        ^{24}
   their absolute addresses*/
                                                                                        25
                                                                                        26
/* prepend particle count */
MPI_Get_address(&j, zzdisp);
                                                                                        27
zzdisp[1] = (MPI_Aint)0;
                                                                                        28
                                                                                        29
zztype[0] = MPI_INT;
                                                                                        30
zztype[1] = Zparticles;
MPI_Type_create_struct(2, zzblock, zzdisp, zztype, &Ztype);
                                                                                        31
                                                                                        32
                                                                                        33
MPI_Type_commit(&Ztype);
                                                                                        34
MPI_Send(MPI_BOTTOM, 1, Ztype, dest, tag, comm);
                                                                                        35
                                                                                        36
                                                                                        37
Example 4.19 Handling of unions.
                                                                                        38
                                                                                        39
union {
                                                                                        40
   int
            ival;
                                                                                        41
   float
            fval;
                                                                                        42
       } u[1000];
                                                                                        43
                                                                                        44
int
         utype;
                                                                                        45
                                                                                        46
/* All entries of u have identical type; variable
                                                                                        47
   utype keeps track of their current type */
                                                                                        48
```

```
1
\mathbf{2}
     MPI_Datatype
                     mpi_utype[2];
3
     MPI_Aint
                     i, extent;
4
5
     /* compute an MPI datatype for each possible union type;
6
        assume values are left-aligned in union storage. */
7
8
     MPI_Get_address(u, &i);
9
     MPI_Get_address(u+1, &extent);
10
     extent = MPI_Aint_diff(extent, i);
11
12
     MPI_Type_create_resized(MPI_INT, 0, extent, &mpi_utype[0]);
13
14
     MPI_Type_create_resized(MPI_FLOAT, 0, extent, &mpi_utype[1]);
15
16
     for(i=0; i<2; i++) MPI_Type_commit(&mpi_utype[i]);</pre>
17
18
     /* actual communication */
19
     MPI_Send(u, 1000, mpi_utype[utype], dest, tag, comm);
20
21
     Example 4.20 This example shows how a datatype can be decoded. The routine
22
     printdatatype prints out the elements of the datatype. Note the use of MPI_Type_free for
23
     datatypes that are not predefined.
24
25
     /*
26
       Example of decoding a datatype.
27
28
       Returns 0 if the datatype is predefined, 1 otherwise
29
      */
30
     #include <stdio.h>
^{31}
     #include <stdlib.h>
32
     #include "mpi.h"
33
     int printdatatype(MPI_Datatype datatype)
34
     {
35
         int *array_of_ints;
36
         MPI_Aint *array_of_adds;
37
         MPI_Datatype *array_of_dtypes;
38
         int num_ints, num_adds, num_dtypes, combiner;
39
         int i;
40
41
         MPI_Type_get_envelope(datatype,
42
                                 &num_ints, &num_adds, &num_dtypes, &combiner);
         switch (combiner) {
43
44
         case MPI_COMBINER_NAMED:
45
             printf("Datatype is named:");
46
             /* To print the specific type, we can match against the
47
                 predefined forms. We can NOT use a switch statement here
48
                 We could also use MPI_TYPE_GET_NAME if we prefered to use
```

```
names that the user may have changed.
     */
    if
             (datatype == MPI_INT)
                                       printf("MPI_INT\n");
    else if (datatype == MPI_DOUBLE) printf("MPI_DOUBLE\n");
    ... else test for other types ...
    return 0;
    break;
case MPI_COMBINER_STRUCT:
case MPI_COMBINER_STRUCT_INTEGER:
                                                                                10
    printf("Datatype is struct containing");
                                                                                11
    array_of_ints
                     = (int *)malloc(num_ints * sizeof(int));
    array_of_adds
                                                                                12
                (MPI_Aint *) malloc(num_adds * sizeof(MPI_Aint));
                                                                                13
    array_of_dtypes = (MPI_Datatype *)
                                                                                14
                                                                                15
        malloc(num_dtypes * sizeof(MPI_Datatype));
                                                                                16
    MPI_Type_get_contents(datatype, num_ints, num_adds, num_dtypes,
                                                                                17
                        array_of_ints, array_of_adds, array_of_dtypes);
                                                                                18
    printf(" %d datatypes:\n", array_of_ints[0]);
                                                                                19
    for (i=0; i<array_of_ints[0]; i++) {</pre>
        printf("blocklength %d, displacement %ld, type:\n",
                                                                                20
                                                                                21
                 array_of_ints[i+1], (long)array_of_adds[i]);
        if (printdatatype(array_of_dtypes[i])) {
                                                                                22
                                                                                23
            /* Note that we free the type ONLY if it
                                                                                24
                is not predefined */
                                                                                25
            MPI_Type_free(&array_of_dtypes[i]);
                                                                                26
        }
    }
                                                                                27
                                                                                28
    free(array_of_ints);
                                                                                29
    free(array_of_adds);
    free(array_of_dtypes);
                                                                                30
                                                                                31
    break;
                                                                                32
    ... other combiner values ...
                                                                                33
default:
                                                                                34
    printf("Unrecognized combiner type\n");
}
                                                                                35
                                                                                36
return 1;
                                                                                37
                                                                                38
```

Pack and Unpack 4.2

}

Some existing communication libraries provide pack/unpack functions for sending noncontiguous data. In these, the user explicitly packs data into a contiguous buffer before sending it, and unpacks it from a contiguous buffer after receiving it. Derived datatypes, which are described in Section 4.1, allow one, in most cases, to avoid explicit packing and unpacking. The user specifies the layout of the data to be sent or received, and the communication library directly accesses a noncontiguous buffer. The pack/unpack routines are provided for compatibility with previous libraries. Also, they provide some functionality that is not

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5 6

9

39

40 41

42

43

44

45

46

47

```
1
     otherwise available in MPI. For instance, a message can be received in several parts, where
\mathbf{2}
      the receive operation done on a later part may depend on the content of a former part.
3
      Another use is that outgoing messages may be explicitly buffered in user supplied space,
4
      thus overriding the system buffering policy. Finally, the availability of pack and unpack
5
      operations facilitates the development of additional communication libraries layered on top
6
      of MPI.
7
8
      MPI_PACK(inbuf, incount, datatype, outbuf, outsize, position, comm)
9
10
                 inbuf
       IN
                                              input buffer start (choice)
11
                                              number of input data items (non-negative integer)
       IN
                 incount
12
                                              datatype of each input data item (handle)
       IN
                 datatype
13
14
                                              output buffer start (choice)
       OUT
                 outbuf
15
       IN
                 outsize
                                              output buffer size, in bytes (non-negative integer)
16
       INOUT
                 position
                                              current position in buffer, in bytes (integer)
17
18
       IN
                                              communicator for packed message (handle)
                 comm
19
20
      C binding
21
      int MPI_Pack(const void *inbuf, int incount, MPI_Datatype datatype,
22
                     void *outbuf, int outsize, int *position, MPI_Comm comm)
23
     Fortran 2008 binding
^{24}
     MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror)
25
          TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
26
          INTEGER, INTENT(IN) :: incount, outsize
27
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
28
          TYPE(*), DIMENSION(..) :: outbuf
29
          INTEGER, INTENT(INOUT) :: position
30
          TYPE(MPI_Comm), INTENT(IN) :: comm
^{31}
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33
      Fortran binding
34
      MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)
35
          <type> INBUF(*), OUTBUF(*)
36
          INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR
37
          Packs the message in the send buffer specified by inbuf, incount, datatype into the buffer
38
      space specified by outbuf and outsize. The input buffer can be any communication buffer
39
     allowed in MPI_SEND. The output buffer is a contiguous storage area containing outsize
40
      bytes, starting at the address outbuf (length is counted in bytes, not elements, as if it were
41
      a communication buffer for a message of type MPI_PACKED).
42
          The input value of position is the first location in the output buffer to be used for
43
```

packing. position is incremented by the size of the packed message, and the output value
 of position is the first location in the output buffer following the locations occupied by the
 packed message. The comm argument is the communicator that will be subsequently used
 for sending the packed message.

MPI_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm)					
IN	inbuf input buffer start (choice)				
IN	insize	size of input buffer, in bytes (non-negative integer)			
INOUT	position	current position in bytes (integer)			
OUT	outbuf	output buffer start (choice)			
IN	outcount	number of items to be unpacked (integer)			
IN	datatype	datatype of each output data item (handle)			
IN	comm	communicator for packed message (handle)			

C binding

Fortran 2008 binding

MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm,
ierror)
TYPE(*), DIMENSION(), INTENT(IN) :: inbuf
INTEGER, INTENT(IN) :: insize, outcount
INTEGER, INTENT(INOUT) :: position
TYPE(*), DIMENSION() :: outbuf
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM,
IERROR)

<type> INBUF(*), OUTBUF(*)

```
INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR
```

Unpacks a message into the receive buffer specified by outbuf, outcount, datatype from the buffer space specified by inbuf and insize. The output buffer can be any communication buffer allowed in MPI_RECV. The input buffer is a contiguous storage area containing insize bytes, starting at address inbuf. The input value of position is the first location in the input buffer occupied by the packed message. position is incremented by the size of the packed message, so that the output value of position is the first location in the input buffer after the locations occupied by the message that was unpacked. comm is the communicator used to receive the packed message.

41 Note the difference between MPI_RECV and MPI_UNPACK: in Advice to users. 42MPI_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 43 44the 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 4546increment in **position**. The reason for this change is that the "incoming message size" 47is not predetermined since the user decides how much to unpack; nor is it easy to 48 determine the "message size" from the number of items to be unpacked. In fact, in a

Unofficial Draft for Comment Only

13

14

15 16

29

30

31 32

33

34

35

36

37

38

39

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.)

¹⁰ Several messages can be successively packed into one **packing unit**. This is effected ¹¹ by several successive **related** calls to MPI_PACK, where the first call provides **position** = 0, ¹² and each successive call inputs the value of **position** that was output by the previous call, ¹³ and the same values for **outbuf**, **outcount** and **comm**. This packing unit now contains the ¹⁴ equivalent information that would have been stored in a message by one send call with a ¹⁵ 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

²⁵ MPI_UNPACK, where the first call provides position = 0, and each successive call inputs the ²⁶ value of position that was output by the previous call, and the same values for inbuf, insize ²⁷ and comm.

The concatenation of two packing units is not necessarily a packing unit; nor is a substring of a packing unit necessarily a packing unit. Thus, one cannot concatenate two packing units and then unpack the result as one packing unit; nor can one unpack a substring of a packing unit as a separate packing unit. Each packing unit, that was created by a related sequence of pack calls, or by a regular send, must be unpacked as a unit, by a sequence of related unpack calls.

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

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

- $41 \\ 42$
- 43

34

35

36

37

38 39

40

44

45

46

47

48

1

2

MPI_PA	CK_SIZE(incount, datatype, com	m, size)	1		
IN	incount	count argument to packing call (non-negative integer)	2		
IN	datatype	datatype argument to packing call (handle)	3 4		
IN	comm	communicator argument to packing call (handle)	5		
OUT	size	upper bound on size of packed message, in bytes	6		
		(non-negative integer)	7		
			8		
C bindi	ng		9 10		
int MPI	<pre>int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,</pre>				
	int *size)		12		
Fortran	2008 binding		13		
MPI_Pac	k_size(incount, datatype, o	comm, size, ierror)	14		
	EGER, INTENT(IN) :: incount		15		
	E(MPI_Datatype), INTENT(IN)		16 17		
	E(MPI_Comm), INTENT(IN) :: EGER, INTENT(OUT) :: size	comm	18		
	EGER, OPTIONAL, INTENT(OUT)) :: ierror	19		
			20		
	binding K_SIZE(INCOUNT, DATATYPE, (TOMM STZE TERROR)	21		
	EGER INCOUNT, DATATYPE, CON		22		
			23		
A call to MPI_PACK_SIZE(incount, datatype, comm, size) returns in size an upper bound on the increment in position that is effected by a call to MPI_PACK(inbuf, incount, datatype,			24 25		
	outbuf, outcount, position, comm). If the packed size of the datatype cannot be expressed				
		SIZE sets the value of size to MPI_UNDEFINED.	27		
-	_	oper bound, rather than an exact bound, since the	28		
	exact amount of space needed to pack the message may depend on the context (e.g.,				
		unit may take more space). (End of rationale.)	30		
			31 32		
Examp	e 4.21 An example using MPI.	_PACK.	33		
int	<pre>position, i, j, a[2];</pre>		34		
char	buff[1000];		35		
			36		
	m_rank(MPI_COMM_WORLD, &my	rank);	37		
Ũ	ank == 0)		38 39		
{	SENDER CODE */		40		
7 -1-			41		
pos	ition = 0;		42		
MPI	_Pack(&i, 1, MPI_INT, buff	, 1000, &position, MPI_COMM_WORLD);	43		
	-	, 1000, &position, MPI_COMM_WORLD);	44		
	_Send(buff, position, MPI_1	PACKED, 1, 0, MPI_COMM_WORLD);	45 46		
} else /					
		<pre>MPI_COMM_WORLD, MPI_STATUS_IGNORE);</pre>	48		

Unofficial Draft for Comment Only

```
1
     Example 4.22 An elaborate example.
\mathbf{2}
     int
           position, i;
3
     float a[1000];
4
     char buff[1000];
5
6
     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
7
     if (myrank == 0)
8
     {
9
         /* SENDER CODE */
10
11
         int len[2];
12
         MPI_Aint disp[2];
13
         MPI_Datatype type[2], newtype;
14
15
         /* build datatype for i followed by a[0]...a[i-1]
16
17
         len[0] = 1;
18
         len[1] = i;
19
         MPI_Get_address(&i, disp);
20
         MPI_Get_address(a, disp+1);
21
         type[0] = MPI_INT;
22
         type[1] = MPI_FLOAT;
23
         MPI_Type_create_struct(2, len, disp, type, &newtype);
24
         MPI_Type_commit(&newtype);
25
26
         /* Pack i followed by a[0]...a[i-1]*/
27
28
         position = 0;
29
         MPI_Pack(MPI_BOTTOM, 1, newtype, buff, 1000, &position, MPI_COMM_WORLD);
30
31
         /* Send */
32
33
         MPI_Send(buff, position, MPI_PACKED, 1, 0,
34
                   MPI_COMM_WORLD);
35
36
     /* ****
37
        One can replace the last three lines with
38
        MPI_Send(MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
39
        **** */
40
     }
41
     else if (myrank == 1)
42
     {
43
         /* RECEIVER CODE */
44
45
         MPI_Status status;
46
47
         /* Receive */
48
```

```
\mathbf{2}
    MPI_Recv(buff, 1000, MPI_PACKED, 0, 0, MPI_COMM_WORLD, &status);
                                                                                       3
    /* Unpack i */
                                                                                       \mathbf{4}
                                                                                       5
    position = 0;
                                                                                       6
    MPI_Unpack(buff, 1000, &position, &i, 1, MPI_INT, MPI_COMM_WORLD);
    /* Unpack a[0]...a[i-1] */
                                                                                       9
                                                                                      10
    MPI_Unpack(buff, 1000, &position, a, i, MPI_FLOAT, MPI_COMM_WORLD);
}
                                                                                      11
                                                                                      12
                                                                                      13
Example 4.23 Each process sends a count, followed by count characters to the root; the
                                                                                      14
root concatenates all characters into one string.
                                                                                      15
                                                                                      16
int count, gsize, counts[64], totalcount, k1, k2, k,
                                                                                      17
     displs[64], position, concat_pos;
                                                                                      18
char chr[100], *lbuf, *rbuf, *cbuf;
                                                                                      19
MPI_Comm_size(comm, &gsize);
                                                                                      20
                                                                                      21
MPI_Comm_rank(comm, &myrank);
                                                                                      22
                                                                                      23
      /* allocate local pack buffer */
                                                                                      24
MPI_Pack_size(1, MPI_INT, comm, &k1);
                                                                                      25
MPI_Pack_size(count, MPI_CHAR, comm, &k2);
                                                                                      26
k = k1 + k2;
lbuf = (char *)malloc(k);
                                                                                      27
                                                                                      28
                                                                                      29
      /* pack count, followed by count characters */
                                                                                      30
position = 0;
                                                                                      31
MPI_Pack(&count, 1, MPI_INT, lbuf, k, &position, comm);
MPI_Pack(chr, count, MPI_CHAR, lbuf, k, &position, comm);
                                                                                      32
                                                                                      33
                                                                                      34
if (myrank != root) {
    /* gather at root sizes of all packed messages */
                                                                                      35
                                                                                      36
    MPI_Gather(&position, 1, MPI_INT, NULL, 0,
                                                                                      37
                MPI_DATATYPE_NULL, root, comm);
                                                                                      38
                                                                                      39
    /* gather at root packed messages */
                                                                                      40
    MPI_Gatherv(lbuf, position, MPI_PACKED, NULL,
                                                                                      41
                 NULL, NULL, MPI_DATATYPE_NULL, root, comm);
                                                                                      42
            /* root code */
} else {
                                                                                      43
                                                                                      44
    /* gather sizes of all packed messages */
    MPI_Gather(&position, 1, MPI_INT, counts, 1,
                                                                                      45
                MPI_INT, root, comm);
                                                                                      46
                                                                                      47
```

```
Unofficial Draft for Comment Only
```

/* gather all packed messages */

1

```
1
          displs[0] = 0;
2
          for (i=1; i < gsize; i++)
3
              displs[i] = displs[i-1] + counts[i-1];
4
          totalcount = displs[gsize-1] + counts[gsize-1];
5
          rbuf = (char *)malloc(totalcount);
6
          cbuf = (char *)malloc(totalcount);
7
          MPI_Gatherv(lbuf, position, MPI_PACKED, rbuf,
8
                       counts, displs, MPI_PACKED, root, comm);
9
10
          /* unpack all messages and concatenate strings */
11
          concat_pos = 0;
12
          for (i=0; i < gsize; i++) {</pre>
13
              position = 0;
14
              MPI_Unpack(rbuf+displs[i], totalcount-displs[i],
15
                           &position, &count, 1, MPI_INT, comm);
16
              MPI_Unpack(rbuf+displs[i], totalcount-displs[i],
17
                           &position, cbuf+concat_pos, count, MPI_CHAR, comm);
18
              concat_pos += count;
19
          }
20
          cbuf[concat_pos] = '\0';
21
     }
22
23
     4.3
            Canonical MPI_PACK and MPI_UNPACK
24
25
     These functions read/write data to/from the buffer in the "external32" data format specified
26
     in Section 13.5.2, and calculate the size needed for packing. Their first arguments specify
27
     the data format, for future extensibility, but currently the only valid value of the datarep
28
     argument is "external32."
29
30
           Advice to users. These functions could be used, for example, to send typed data in a
31
           portable format from one MPI implementation to another. (End of advice to users.)
32
          The buffer will contain exactly the packed data, without headers. MPI_BYTE should
33
     be used to send and receive data that is packed using MPI_PACK_EXTERNAL.
34
35
           Rationale. MPI_PACK_EXTERNAL specifies that there is no header on the message
36
           and further specifies the exact format of the data. Since MPI_PACK may (and is
37
           allowed to) use a header, the datatype MPI_PACKED cannot be used for data packed
38
           with MPI_PACK_EXTERNAL. (End of rationale.)
39
40
41
42
43
44
45
46
47
48
```

MPI_PACK_EXTERNAL(datarep, inbuf, incount, datatype, outbuf, outsize, position)				
IN	datarep	data representation (string)		
IN	inbuf	input buffer start (choice)		
IN	incount	number of input data items (integer)		
IN	datatype	datatype of each input data item (handle)		
OUT	outbuf	output buffer start (choice)		
IN	outsize	output buffer size, in bytes (integer)		
INOUT	position	current position in buffer, in bytes (integer)		

C binding

Fortran 2008 binding

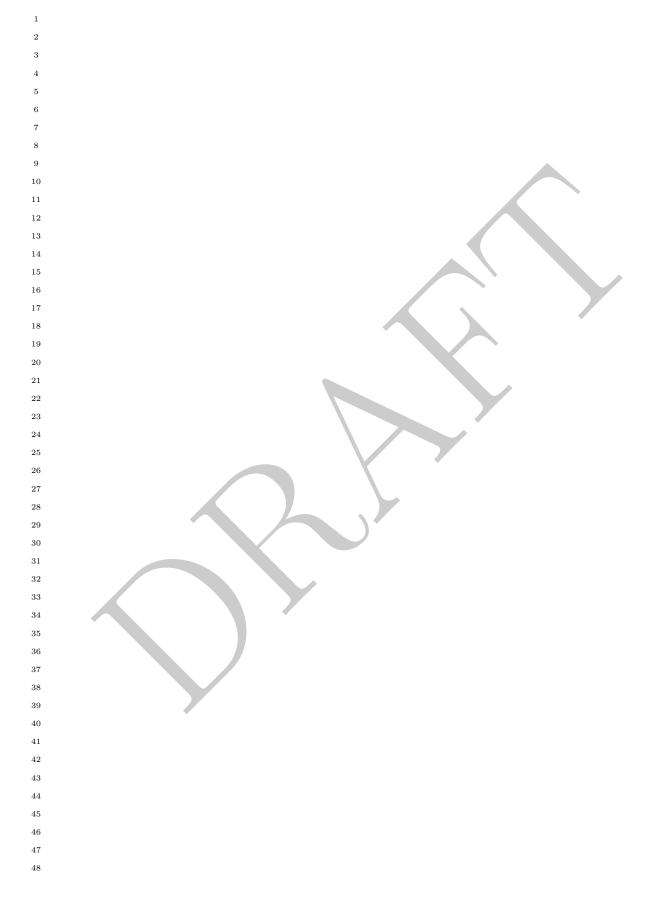
MPI_Pack_external(datarep, inbuf, incount, datatype, outbuf, outsize,
position, ierror)
CHARACTER(LEN=*), INTENT(IN) :: datarep
TYPE(*), DIMENSION(), INTENT(IN) :: inbuf
INTEGER, INTENT(IN) :: incount
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(*), DIMENSION() :: outbuf
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: outsize
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,						
POSITION, IERROR)						
CHARACTER*(*) DATAREP						
<type> INBUF(*), OUTBUF(*)</type>						
INTEGER INCOUNT, DATATYPE, IERROR						
INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, POSITION						

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

Fortran 2008 binding MPI_Pack_external_size(datarep, incount, datatype, size, ierror) CHARACTER(LEN=*), INTENT(IN) :: datarep INTEGER, INTENT(IN) :: incount TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR) CHARACTER*(*) DATAREP INTEGER INCOUNT, DATATYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SIZE



Chapter 5

Collective Communication

5.1 Introduction and Overview

Collective communication is defined as communication that involves a group or groups of processes. The functions of this type provided by MPI are the following:

- MPI_BARRIER, MPI_IBARRIER: Barrier synchronization across all members of a group (Section 5.3 and Section 5.12.1).
- MPI_BCAST, MPI_IBCAST: Broadcast from one member to all members of a group (Section 5.4 and Section 5.12.2). This is shown as "broadcast" in Figure 5.1.
- MPI_GATHER, MPI_IGATHER, MPI_GATHERV, MPI_IGATHERV: Gather data from all members of a group to one member (Section 5.5 and Section 5.12.3). This is shown as "gather" in Figure 5.1.
- MPI_SCATTER, MPI_ISCATTER, MPI_SCATTERV, MPI_ISCATTERV: Scatter data from one member to all members of a group (Section 5.6 and Section 5.12.4). This is shown as "scatter" in Figure 5.1.
- MPI_ALLGATHER, MPI_IALLGATHER, MPI_ALLGATHERV, MPI_IALLGATHERV: A variation on Gather where all members of a group receive the result (Section 5.7 and Section 5.12.5). This is shown as "allgather" in Figure 5.1.
- MPI_ALLTOALL, MPI_IALLTOALL, MPI_ALLTOALLV, MPI_IALLTOALLV, MPI_ALLTOALLW, MPI_IALLTOALLW, MPI_IALLTOALLW: Scatter/Gather data from all members to all members of a group (also called complete exchange) (Section 5.8 and Section 5.12.6). This is shown as "complete exchange" in Figure 5.1.
- MPI_ALLREDUCE, MPI_IALLREDUCE, MPI_REDUCE, MPI_IREDUCE: Global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to all members of a group (Section 5.9.6 and Section 5.12.8) and a variation where the result is returned to only one member (Section 5.9 and Section 5.12.7).
- MPI_REDUCE_SCATTER_BLOCK, MPI_IREDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, MPI_IREDUCE_SCATTER: A combined reduction and scatter operation (Section 5.10, Section 5.12.9, and Section 5.12.10).

4

• MPI_SCAN, MPI_ISCAN, MPI_EXSCAN, MPI_IEXSCAN: Scan across all members of a group (also called prefix) (Section 5.11, Section 5.11.2, Section 5.12.11, and Section 5.12.12).

One of the key arguments in a call to a collective routine is a communicator that 5defines the group or groups of participating processes and provides a context for the oper-6 ation. This is discussed further in Section 5.2. 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 4. 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 4 for 13 information concerning communication buffers, general datatypes and type matching rules, 14and to Chapter 6 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 4.1) 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 5.14.

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

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

(End of rationale.)

Advice to users. It is dangerous to rely on synchronization side-effects of the 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.

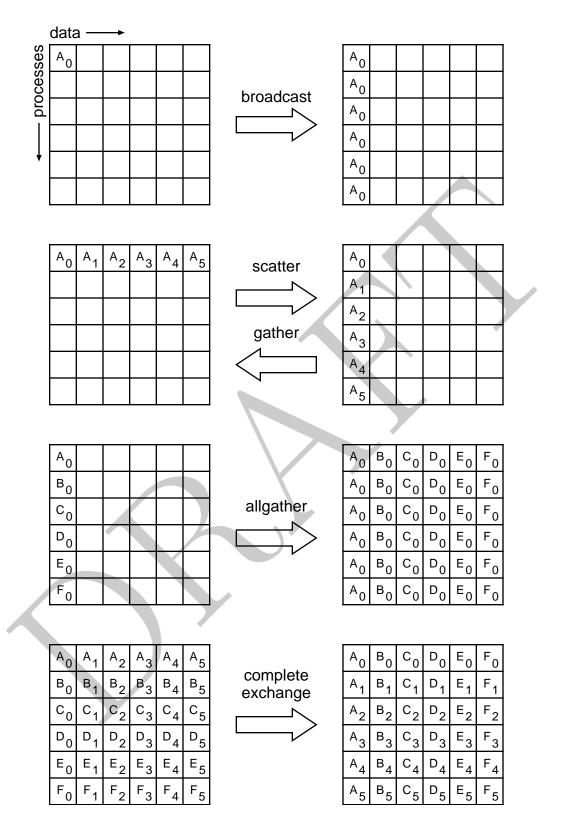


Figure 5.1: Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially just the first process contains the data A_0 , but after the broadcast all processes contain it.

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 5.14. (End of advice to users.)

Advice to implementors. While vendors may write optimized collective routines matched to their architectures, a complete library of the collective communication routines can be written entirely using the MPI point-to-point communication functions and a few auxiliary functions. If implementing on top of point-to-point, a hidden, special communicator might be created for the collective operation so as to avoid interference with any on-going point-to-point communication at the time of the collective call. This is discussed further in Section 5.14. (End of advice to implementors.)

Many of the descriptions of the collective routines provide illustrations in terms of 13 blocking MPI point-to-point routines. These are intended solely to indicate what data is sent or received by what process. Many of these examples are not correct MPI programs; 1516for purposes of simplicity, they often assume infinite buffering.

5.2 Communicator Argument

20The key concept of the collective functions is to have a group or groups of participating 21processes. The routines do not have group identifiers as explicit arguments. Instead, there 22 is a communicator argument. Groups and communicators are discussed in full detail in 23Chapter 6. For the purposes of this chapter, it is sufficient to know that there are two types 24 of communicators: intra-communicators and inter-communicators. An intracommunicator 25can be thought of as an identifier for a single group of processes linked with a context. An 26intercommunicator identifies two distinct groups of processes linked with a context. 27

28Specifics for Intracommunicator Collective Operations 5.2.1 29

All processes in the group identified by the intracommunicator must call the collective 30 routine. 31

In many cases, collective communication can occur "in place" for intracommunicators, 32 with the output buffer being identical to the input buffer. This is specified by providing 33 34a special argument value, MPI_IN_PLACE, instead of the send buffer or the receive buffer argument, depending on the operation performed. 35

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 44 collective calls becomes a send-and-receive buffer. For this reason, a Fortran binding 45 that includes INTENT must mark these as INOUT, not OUT. 46

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

6 7 8

9

10

11

12

1

2

3

4

5

14

1718

5.2.2 Applying Collective Operations to Intercommunicators
To understand how collective operations apply to intercommunicators, we can view most MPI intracommunicator collective operations as fitting one of the following categories (see, for instance, [59]):
All-To-All All processes contribute to the result. All processes receive the result.
 MPI_ALLGATHER, MPI_IALLGATHER, MPI_ALLGATHERV, MPI_IALLGATHERV 9
• MPI_ALLTOALL, MPI_IALLTOALL, MPI_ALLTOALLV, MPI_IALLTOALLV, MPI_ALLTOALLW, MPI_IALLTOALLW, MPI_IALLTOALLW
• MPI_ALLREDUCE, MPI_IALLREDUCE, MPI_REDUCE_SCATTER_BLOCK, MPI_IREDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, MPI_IREDUCE_SCATTER
• MPI_BARRIER, MPI_IBARRIER
All-To-One All processes contribute to the result. One process receives the result.
MPI_GATHER, MPI_IGATHER, MPI_GATHERV, MPI_IGATHERV MPI_REDUCE, MPI_IREDUCE
One-To-All One process contributes to the result. All processes receive the result.
MPI_BCAST, MPI_IBCAST MPI_SCATTER, MPI_SCATTERV, MPI_ISCATTERV
Other Collective operations that do not fit into one of the above categories.
• MPI_SCAN, MPI_ISCAN, MPI_EXSCAN, MPI_IEXSCAN
The data movement patterns of MPI_SCAN, MPI_ISCAN, MPI_EXSCAN, and MPI_IEXSCAN do not fit this taxonomy. The application of collective communication to intercommunicators is best described in terms of two groups. For example, an all-to-all MPI_ALLGATHER operation can be described as collecting data from all members of one group with the result appearing in all members of the other group (see Figure 5.2). As another example, a one-to-all MPI_BCAST operation sends data from one member of one group to all members of the other group. Collective computation operations such as MPI_REDUCE_SCATTER have a similar interpretation (see Figure 5.3). For intracommunicators, these two groups are the same. For intercommunicators, these two groups are distinct. For the all-to-all operations, each such operation is described in two phases, so that it has a symmetric, full-duplex behavior. The following collective operations also apply to intercommunicators:
• MPI_BARRIER, MPI_IBARRIER
• MPI_BCAST, MPI_IBCAST
• MPI_GATHER, MPI_IGATHER, MPI_GATHERV, MPI_IGATHERV,
• MPI_SCATTER, MPI_ISCATTER, MPI_SCATTERV, MPI_ISCATTERV,

Unofficial Draft for Comment Only

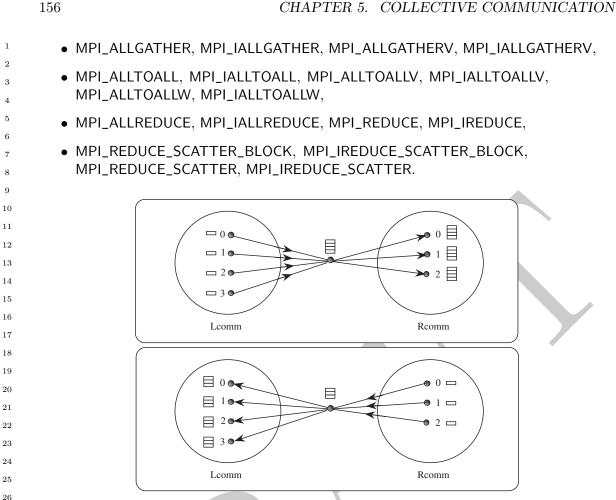


Figure 5.2: Intercommunicator allgather. The focus of data to one process is represented, not mandated by the semantics. The two phases do allgathers in both directions.

29 30 31

32

27

28

5.2.3 Specifics for Intercommunicator Collective Operations

All processes in both groups identified by the intercommunicator must call the collective 33 routine.

34 Note that the "in place" option for intracommunicators does not apply to intercom-35 municators since in the intercommunicator case there is no communication from a process 36 to itself. 37

For intercommunicator 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

Unofficial Draft for Comment Only

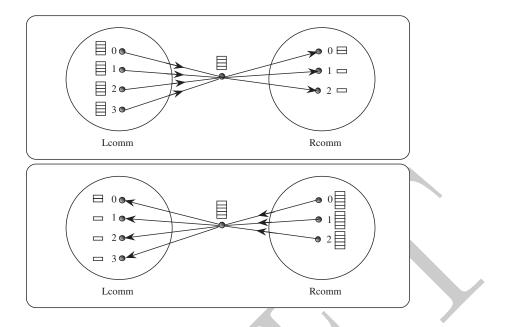


Figure 5.3: Intercommunicator reduce-scatter. The focus of data to one process is represented, not mandated by the semantics. The two phases do reduce-scatters in both directions.

category will often occur as part of an exchange, where it makes sense to communicate in both directions at once. (*End of rationale.*)

5.3 Barrier Synchronization

MPI_BARRIER(comm)						
IN	comm	communicator (handle)				
C binding						
int MPI_Ba	<pre>int MPI_Barrier(MPI_Comm comm)</pre>					
Fortran 20	008 binding					
MPI_Barrier(comm, ierror)						
TYPE(MPI_Comm), INTENT(IN) :: comm						
INTEGE	INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
Fortran binding						

MPI_BARRIER(COMM, IERROR) INTEGER COMM, IERROR

If comm is an intracommunicator, MPI_BARRIER blocks the caller until all group members have called it. The call returns at any process only after all group members have entered the call.

⁴⁶ If comm is an intercommunicator, MPI_BARRIER involves two groups. The call returns at processes in one group (group A) of the intercommunicator only after all members of the ⁴⁸ ⁴⁷ ⁴⁸

Unofficial Draft for Comment Only

```
1
     other group (group B) have entered the call (and vice versa). A process may return from
\mathbf{2}
      the call before all processes in its own group have entered the call.
3
4
            Broadcast
      5.4
5
6
7
8
      MPI_BCAST(buffer, count, datatype, root, comm)
9
                                               starting address of buffer (choice)
       INOUT
                 buffer
10
                                              number of entries in buffer (non-negative integer)
       IN
                 count
11
12
       IN
                 datatype
                                               data type of buffer (handle)
13
       IN
                  root
                                              rank of broadcast root (integer)
14
       IN
                 comm
                                               communicator (handle)
15
16
17
      C binding
      int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root,
18
19
                     MPI_Comm comm)
20
      Fortran 2008 binding
21
     MPI_Bcast(buffer, count, datatype, root, comm, ierror)
22
          TYPE(*), DIMENSION(..) :: buffer
23
          INTEGER, INTENT(IN) :: count, root
24
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
25
          TYPE(MPI_Comm), INTENT(IN) :: comm
26
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     Fortran binding
28
     MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT,
29
                                                      COMM. IERROR)
30
          <type> BUFFER(*)
^{31}
          INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
32
          If comm is an intracommunicator, MPI_BCAST broadcasts a message from the process
33
      with rank root to all processes of the group, itself included. It is called by all members of
34
      the group using the same arguments for comm and root. On return, the content of root's
35
      buffer is copied to all other processes.
36
          General, derived datatypes are allowed for datatype. The type signature of count,
37
      datatype on any process must be equal to the type signature of count, datatype at the root.
38
      This implies that the amount of data sent must be equal to the amount received, pairwise
39
      between each process and the root. MPI_BCAST and all other data-movement collective
40
     routines make this restriction. Distinct type maps between sender and receiver are still
41
     allowed.
42
          The "in place" option is not meaningful here.
43
          If comm is an intercommunicator, then the call involves all processes in the intercom-
44
      municator, but with one group (group A) defining the root process. All processes in the
45
      other group (group B) pass the same value in argument root, which is the rank of the root
46
      in group A. The root passes the value MPI_ROOT in root. All other processes in group A
47
      pass the value MPI_PROC_NULL in root. Data is broadcast from the root to all processes
48
```

in group B. The buffer arguments of the processes in group B must be consistent with the buffer argument of the root.

5.4.1 Example using MPI_BCAST

The examples in this section use intracommunicators.

Example 5.1 Broadcast 100 ints from process 0 to every process in the group.

```
MPI_Comm comm;
int array[100];
int root=0;
...
MPI_Bcast(array, 100, MPI_INT, root, comm);
```

As in many of our example code fragments, we assume that some of the variables (such as comm in the above) have been assigned appropriate values.

5.5 Gather

MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)				
IN	sendbuf	starting address of send buffer (choice)	24	
IN	sendcount	number of elements in send buffer (non-negative	25	
		integer)	26	
IN	sendtype	data type of send buffer elements (handle)	27	
OUT	recvbuf	address of receive buffer (choice, significant only at	28	
001	recobur	root)	29	
			30 31	
IN	recvcount	number of elements for any single receive	32	
		(non-negative integer, significant only at root)	33	
IN	recvtype	data type of recv buffer elements (handle, significant	34	
		only at root)	35	
IN	root	rank of receiving process (integer)	36	
IN	comm	communicator (handle)	37	
			38	
C bindi	ng		39	
	0	f, int sendcount, MPI_Datatype sendtype,	40	
-		recvcount, MPI_Datatype recvtype, int root,	41	
	MPI_Comm comm)		42	
D (43	
	2008 binding		44	
MP1_Gath		ndtype, recvbuf, recvcount, recvtype,	45	
root, comm, ierror)				
TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf INTEGER, INTENT(IN) :: sendcount, recvcount, root			47	
1111	GER, INTENT(IN) Sellaco		48	

Unofficial Draft for Comment Only

1 2

 $\frac{3}{4}$

5

6 7

8 9

10

11

12

13

14 15

16

17 18

 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(*), DIMENSION() :: recvbuf TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror 	
Fortran binding MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR</type>	
¹¹ If comm is an intracommunicator, each process (root process included) sends the c ¹² tents of its send buffer to the root process. The root process receives the messages and sto ¹³ them in rank order. The outcome is <i>as if</i> each of the n processes in the group (includ ¹⁴ the root process) had executed a call to	\mathbf{res}
 MPI_Send(sendbuf, sendcount, sendtype, root ,), and the root had executed n calls to 	
$MPI_{20} MPI_{Recv}(recvbuf+i \cdot recvcount \cdot extent(recvtype), recvcount, recvtype, i,),$	
 where extent(recvtype) is the type extent obtained from a call to MPI_Type_get_extent. An alternative description is that the n messages sent by the processes in the gra are concatenated in rank order, and the resulting message is received by the root as if b call to MPI_RECV(recvbuf, recvcount·n, recvtype,). The receive buffer is ignored for all non-root processes. General, derived datatypes are allowed for both sendtype and recvtype. The type sig ture of sendcount, sendtype on each process must be equal to the type signature of recvcour recvtype at the root. This implies that the amount of data sent must be equal to the amo of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed. All arguments to the function are significant on process root, while on other process only arguments sendbuf, sendcount, sendtype, root, and comm are significant. The argument root and comm must have identical values on all processes. The specification of counts and types should not cause any location on the root to written more than once. Such a call is erroneous. Note that the recvcount argument at the root indicates the number of items it receive from <i>each</i> process, not the total number of items it receives. The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE the value of sendbuf at the root. In such a case, sendcount and sendtype are ignored, a the contribution of the root to the gathered vector is assumed to be already in the corr place in the receive buffer. If comm is an intercommunicator, then the call involves all processes in the interced municator, but with one group (group A) defining the root process. All processes in other group (group B) pass the same value in argument root, which is the rank of the r in group A. The root passes the value MPI_ROOT in root. All other processes in group pass the value MPI_PROC_NULL in root. Data is gathered from all processes in gro	oup by a gna- unt, unt een ses, ents o be ives Ξ as and rect om- the root p A

the root. The send buffer arguments of the processes in group B must be consistent with the receive buffer argument of the root.

MPI_GATHERV	(sendbuf, sendcour	nt, sendtype, recvbu	f, recvcounts, displs	, recvtype, root,
-------------	--------------------	----------------------	-----------------------	-------------------

-	(0	
	comm)		6	
IN	sendbuf	starting address of send buffer (choice)	7	
IN	sendcount	number of elements in send buffer (non-negative	8	
	Schaebane	integer)	9	
IN	sendtype	data type of send buffer elements (handle)	10	
	•••		11 12	
OUT	recvbuf	address of receive buffer (choice, significant only at root)	12	
		,	14	
IN	recvcounts	non-negative integer array (of length group size)	15	
		containing the number of elements that are received	16	
		from each process (significant only at root)	17	
IN	displs	integer array (of length group size). Entry i specifies	18	
		the displacement relative to $recvbuf$ at which to place	19	
		the incoming data from process i (significant only at	20	
		root)	21	
IN	recvtype	data type of recv buffer elements (handle, significant	22	
		only at root)	23	
IN	root	rank of receiving process (integer)	24 25	
IN	comm	communicator (handle)	25 26	
			20	
C bindi	ng		28	
int MPI	_Gatherv(const void	d *sendbuf, int sendcount, MPI_Datatype sendtype,	29	
	void *recvbu	<pre>f, const int recvcounts[], const int displs[],</pre>	30	
	MPI_Datatype	recvtype, int root, MPI_Comm comm)	31	
Fortron	2008 binding		32	
		count, sendtype, recvbuf, recvcounts, displs,	33	
III I_Gat.		ot, comm, ierror)	34	
ТҮР), INTENT(IN) :: sendbuf	35	
		: sendcount, recvcounts(*), displs(*), root	36	
	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype			
TYPE(*), DIMENSION() :: recybuf				
	E(MPI_Comm), INTEN		39	
			40	

Fortran binding

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

 41

MPI_GATHERV extends the functionality of MPI_GATHER by allowing a varying count of data from each process, since recvcounts is now an array. It also allows more flexibility as to where the data is placed on the root, by providing the new argument, displs. If comm is an intracommunicator, the outcome is *as if* each process, including the root process, sends a message to the root,

MPI_Send(sendbuf, sendcount, sendtype, root, ...),

and the root executes n receives,

MPI_Recv(recvbuf+displs[j]· extent(recvtype), recvcounts[j], recvtype, i, ...).

The data received from process j is placed into recvbuf of the root process beginning at offset displs[j] elements (in terms of the recvtype).

The receive buffer is ignored for all non-root processes.

The type signature implied by sendcount, sendtype on process i must be equal to the type signature implied by recvcounts[i], recvtype at the root. This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed, as illustrated in Example 5.6.

All arguments to the function are significant on process root, while on other processes, only arguments sendbuf, sendcount, sendtype, root, and comm are significant. The arguments root and comm must have identical values on all processes.

The specification of counts, types, and displacements should not cause any location on the root to be written more than once. Such a call is erroneous.

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as the value of sendbuf at the root. In such a case, sendcount and sendtype are ignored, and the contribution of the root to the gathered vector is assumed to be already in the correct place in the receive buffer.

²⁹ If comm is an intercommunicator, then the call involves all processes in the intercom-³⁰ municator, but with one group (group A) defining the root process. All processes in the ³¹ other group (group B) pass the same value in argument root, which is the rank of the root ³² in group A. The root passes the value MPI_ROOT in root. All other processes in group A ³⁴ pass the value MPI_PROC_NULL in root. Data is gathered from all processes in group B to ³⁵ the root. The send buffer arguments of the processes in group B must be consistent with ³⁶ the receive buffer argument of the root.

5.5.1 Examples using MPI_GATHER, MPI_GATHERV

³⁹ The examples in this section use intracommunicators.

41 **Example 5.2** Gather 100 ints from every process in group to root. See Figure 5.4.

```
43 MPI_Comm comm;
44 int gsize,sendarray[100];
45 int root, *rbuf;
46 ...
47 MPI_Comm_size(comm, &gsize);
48 rbuf = (int *)malloc(gsize*100*sizeof(int));
```

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6

7 8

9 10

11 12

13

14

15

16

17

18

19

20

21

22

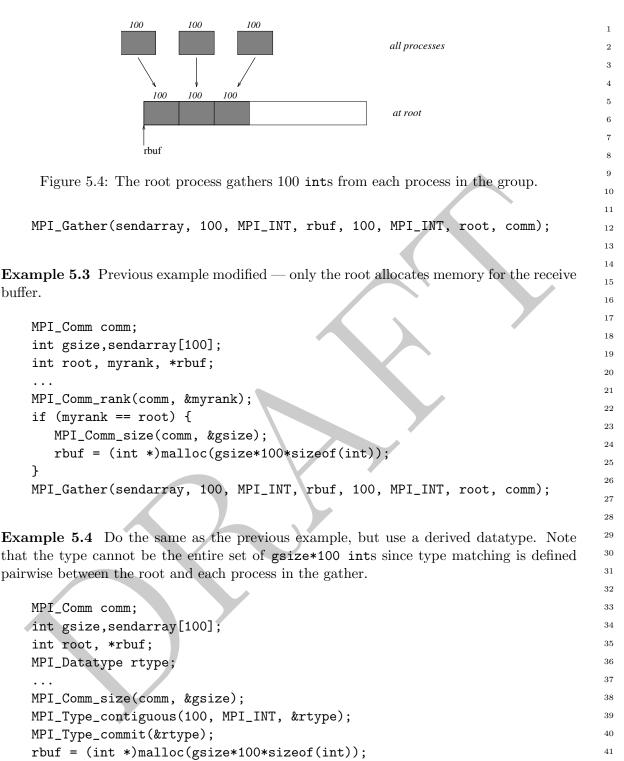
23

 24

37

38

40



MPI_Gather(sendarray, 100, MPI_INT, rbuf, 1, rtype, root, comm);

Example 5.5 Now have each process send 100 ints to root, but place each set (of 100) stride ints apart at receiving end. Use MPI_GATHERV and the displs argument to achieve this effect. Assume $stride \geq 100$. See Figure 5.5.

```
100
                                  100
                                           100
1
2
                                                                 all processes
3
4
                             100
                                    100
                                           100
5
                                                                 at root
6
7
                                    stride
                           rbuf
8
9
     Figure 5.5: The root process gathers 100 ints from each process in the group, each set is
10
     placed stride ints apart.
11
12
          MPI_Comm comm;
13
          int gsize,sendarray[100];
14
          int root, *rbuf, stride;
15
          int *displs,i,*rcounts;
16
17
          . . .
18
19
          MPI_Comm_size(comm, &gsize);
20
          rbuf = (int *)malloc(gsize*stride*sizeof(int));
21
          displs = (int *)malloc(gsize*sizeof(int));
22
          rcounts = (int *)malloc(gsize*sizeof(int));
23
          for (i=0; i<gsize; ++i) {</pre>
24
               displs[i] = i*stride;
25
               rcounts[i] = 100;
26
          }
27
          MPI_Gatherv(sendarray, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT,
28
                        root, comm);
29
30
          Note that the program is erroneous if stride < 100.
^{31}
32
     Example 5.6 Same as Example 5.5 on the receiving side, but send the 100 ints from the
33
     0th column of a 100 \times 150 int array, in C. See Figure 5.6.
34
35
          MPI_Comm comm;
36
          int gsize, sendarray[100][150];
37
          int root, *rbuf, stride;
38
          MPI_Datatype stype;
39
          int *displs,i,*rcounts;
40
41
          . . .
42
          MPI_Comm_size(comm, &gsize);
43
44
          rbuf = (int *)malloc(gsize*stride*sizeof(int));
45
          displs = (int *)malloc(gsize*sizeof(int));
46
          rcounts = (int *)malloc(gsize*sizeof(int));
47
          for (i=0; i<gsize; ++i) {</pre>
48
               displs[i] = i*stride;
```

Unofficial Draft for Comment Only

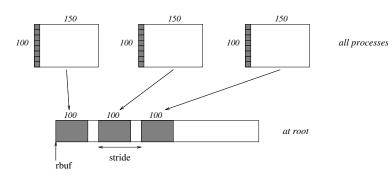


Figure 5.6: The root process gathers column 0 of a 100×150 C array, and each set is placed stride ints apart.

```
rcounts[i] = 100;
}
/* Create datatype for 1 column of array
*/
MPI_Type_vector(100, 1, 150, MPI_INT, &stype);
MPI_Type_commit(&stype);
MPI_Gatherv(sendarray, 1, stype, rbuf, rcounts, displs, MPI_INT,
root, comm);
```

Example 5.7 Process i sends (100-i) ints from the i-th column of a 100×150 int array, in C. It is received into a buffer with stride, as in the previous two examples. See Figure 5.7.

```
MPI_Comm comm;
                                                                                 27
int gsize,sendarray[100][150],*sptr;
                                                                                 28
int root, *rbuf, stride, myrank;
                                                                                 29
                                                                                 30
MPI_Datatype stype;
                                                                                 31
int *displs,i,*rcounts;
                                                                                 32
                                                                                 33
. .
                                                                                 34
MPI_Comm_size(comm, &gsize);
                                                                                 35
                                                                                 36
MPI_Comm_rank(comm, &myrank);
                                                                                 37
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
                                                                                 38
                                                                                 39
rcounts = (int *)malloc(gsize*sizeof(int));
                                                                                 40
for (i=0; i<gsize; ++i) {</pre>
                                                                                 41
    displs[i] = i*stride;
                                                                                 42
    rcounts[i] = 100-i;
                              /* note change from previous example */
}
                                                                                 43
                                                                                 44
/* Create datatype for the column we are sending
                                                                                 45
 */
                                                                                 46
MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &stype);
                                                                                 47
MPI_Type_commit(&stype);
                                                                                 48
/* sptr is the address of start of "myrank" column
```

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

9 10

11

12 13

14

15

16

17

18

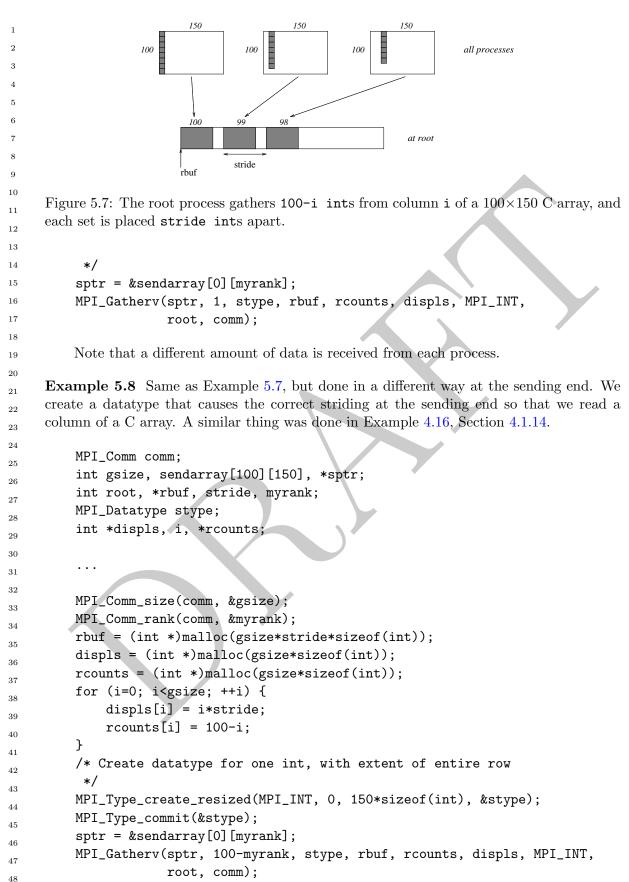
19

20

21 22 23

 24

25



Example 5.9 Same as Example 5.7 at sending side, but at receiving side we make the stride between received blocks vary from block to block. See Figure 5.8.

```
MPI_Comm comm;
int gsize, sendarray[100][150], *sptr;
int root, *rbuf, *stride, myrank, bufsize;
MPI_Datatype stype;
int *displs,i,*rcounts,offset;
. . .
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);
stride = (int *)malloc(gsize*sizeof(int));
 . . .
/* stride[i] for i = 0 to gsize-1 is set somehow
 */
/* set up displs and rcounts vectors first
 */
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
offset = 0;
for (i=0; i<gsize; ++i) {</pre>
    displs[i] = offset;
    offset += stride[i];
    rcounts[i] = 100-i;
}
/* the required buffer size for rbuf is now easily obtained
 */
bufsize = displs[gsize-1]+rcounts[gsize-1];
rbuf = (int *)malloc(bufsize*sizeof(int));
/* Create datatype for the column we are sending
 */
MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &stype);
MPI_Type_commit(&stype);
sptr = &sendarray[0][myrank];
MPI_Gatherv(sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
             root, comm);
```

Example 5.10 Process i sends 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.

1

 $\mathbf{2}$

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

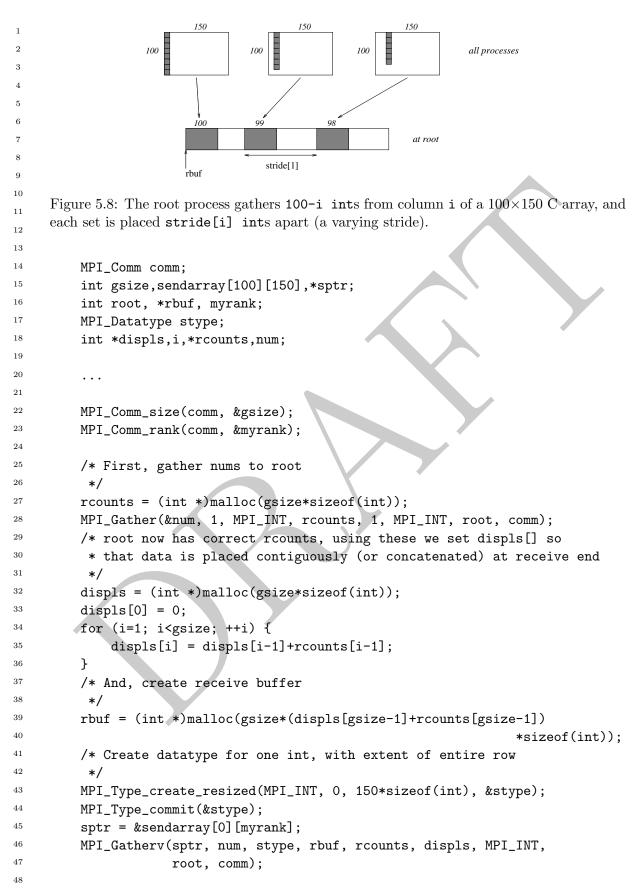
43

44

45

46 47 48

```
CHAPTER 5. COLLECTIVE COMMUNICATION
```



5.6 Scatter

MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	sendbuf	address of send buffer (choice, significant only at	6
		root)	7
IN	sendcount	number of elements sent to each process	8
		(non-negative integer, significant only at root)	9
IN	sendtype	data type of send buffer elements (handle, significant	10 11
		only at root)	11
OUT	recvbuf	address of receive buffer (choice)	13
IN	recvcount	number of elements in receive buffer (non-negative	14
		integer)	15
IN	racutura	data type of receive buffer elements (handle)	16
IIN	recvtype	data type of receive buller elements (nandle)	17
IN	root	rank of sending process (integer)	18
IN	comm	communicator (handle)	19
			20

C binding

int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

	37
ROOT, COMM, IERROR)	
<type> SENDBUF(*), RECVBUF(*)</type>	38
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR	39
INTEGER DEMDODONI, DEMDITTE, REGVOLONI, REGVITTE, ROOI, ODINI, IERROR	40
MPI_SCATTER is the inverse operation to MPI_GATHER.	41
If comm is an intracommunicator, the outcome is as if the root executed n send oper-	42
ations	43

ations,

MPI_Send(sendbuf+i· sendcount· extent(sendtype), sendcount, sendtype, i,...),

and each process executed a receive,

 31

 $45 \\ 46$

 $\frac{3}{4}$

MPI_Recv(recvbuf, recvcount, recvtype, i,...).

An alternative description is that the root sends a message with MPI_Send(sendbuf, sendcount n, sendtype, ...). This message is split into n equal segments, the *i*-th segment is sent to the *i*-th process in the group, and each process receives this message as above.

5 6

17

18

19

20 21

The send buffer is ignored for all non-root processes.

The type signature associated with sendcount, sendtype at the root must be equal to the type signature associated with recvcount, recvtype at all processes (however, the type maps may be different). This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

All arguments to the function are significant on process root, while on other processes,
 only arguments recvbuf, recvcount, recvtype, root, and comm are significant. The arguments
 root and comm must have identical values on all processes.

The specification of counts and types should not cause any location on the root to be read more than once.

Rationale. Though not needed, the last restriction is imposed so as to achieve symmetry with MPI_GATHER, where the corresponding restriction (a multiple-write restriction) is necessary. (*End of rationale.*)

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as the value of recvbuf at the root. In such a case, recvcount and recvtype are ignored, and root "sends" no data to itself. The scattered vector is still assumed to contain n segments, where n is the group size; the *root*-th segment, which root should "send to itself," is not moved.

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

MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root,					
	comm)		2		
IN	sendbuf	address of send buffer (choice, significant only at	3 4		
		root)	4 5		
IN	sendcounts	non-negative integer array (of length group size)	6		
		specifying the number of elements to send to each	7		
		rank (significant only at root)	8		
IN	displs	integer array (of length group size). Entry i specifies	9		
	·	the displacement (relative to sendbuf) from which to	10		
		take the outgoing data to process i (significant only	11		
		at root)	12		
IN	sendtype	data type of send buffer elements (handle, significant	13		
		only at root)	14		
OUT	recvbuf	address of receive buffer (choice)	15 16		
IN		number of elements in receive buffer (non-negative	17		
IIN	recvcount	integer)	18		
			19		
IN	recvtype	data type of receive buffer elements (handle)	20		
IN	root	rank of sending process (integer)	21		
IN	comm	communicator (handle)	22		
			23		
C bindi	ng		24		
int MPI	_Scatterv(const voi	d *sendbuf, const int sendcounts[],	25		
	const int dis	spls[], MPI_Datatype sendtype, void *recvbuf,	26		
	int recvcount	;, MPI_Datatype recvtype, int root, MPI_Comm comm)	27 28		
Fortran	2008 binding		28 29		
	U	counts, displs, sendtype, recvbuf, recvcount,	30		
-		ot, comm, ierror)	31		
TYPI		, INTENT(IN) :: sendbuf	32		
INTE	EGER, INTENT(IN) ::	<pre>sendcounts(*), displs(*), recvcount, root</pre>	33		
TYPI	E(MPI_Datatype), IN	TENT(IN) :: sendtype, recvtype	34		
	E(*), DIMENSION()		35		
TYPE(MPI_Comm), INTENT(IN) :: comm					
INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
Fortran binding					
MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,					
RECVTYPE, ROOT, COMM, IERROR)					
<type> SENDBUF(*), RECVBUF(*) 41</type>					
INTEGER SENDCOUNTS(*) DISPLS(*) SENDTYPE RECVCOUNT RECYTYPE ROOT					

INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR

MPI_SCATTERV is the inverse operation to MPI_GATHERV.

MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying 46 count of data to be sent to each process, since sendcounts is now an array. It also allows 47

43

44

45

1 more flexibility as to where the data is taken from on the root, by providing an additional $\mathbf{2}$ argument, displs. 3 If comm is an intracommunicator, the outcome is as if the root executed n send oper-4 ations, 5MPI_Send(sendbuf+displs[i] extent(sendtype), sendcounts[i], sendtype, i,...), 6 7 and each process executed a receive, 8 9 MPI_Recv(recvbuf, recvcount, recvtype, i,...). 10 11 The send buffer is ignored for all non-root processes. 12The type signature implied by sendcount[i], sendtype at the root must be equal to the 13 type signature implied by recvcount, recvtype at process i (however, the type maps may be 14different). This implies that the amount of data sent must be equal to the amount of data 15received, pairwise between each process and the root. Distinct type maps between sender 16and receiver are still allowed. 17All arguments to the function are significant on process root, while on other processes, 18 only arguments recvbuf, recvcount, recvtype, root, and comm are significant. The arguments 19root and comm must have identical values on all processes. 20The specification of counts, types, and displacements should not cause any location on 21the root to be read more than once. 22 The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as 23the value of recvbuf at the root. In such a case, recvcount and recvtype are ignored, and 24 root "sends" no data to itself. The scattered vector is still assumed to contain n segments, 25where n is the group size; the *root*-th segment, which root should "send to itself," is not 26moved. 27If comm is an intercommunicator, then the call involves all processes in the intercom-28municator, but with one group (group A) defining the root process. All processes in the 29 other group (group B) pass the same value in argument root, which is the rank of the root 30 in group A. The root passes the value MPI_ROOT in root. All other processes in group A 31 pass the value MPI_PROC_NULL in root. Data is scattered from the root to all processes in 32 group B. The receive buffer arguments of the processes in group B must be consistent with 33 the send buffer argument of the root. 34 35 Examples using MPI_SCATTER, MPI_SCATTERV 5.6.1 36 37 The examples in this section use intracommunicators. 38 39 **Example 5.11** The reverse of Example 5.2. Scatter sets of 100 ints from the root to each process in the group. See Figure 5.9. 4041 MPI_Comm comm; 42int gsize,*sendbuf; 43 int root, rbuf[100]; 44 . . . 45MPI_Comm_size(comm, &gsize); 46 sendbuf = (int *)malloc(gsize*100*sizeof(int)); 47

48

. . .

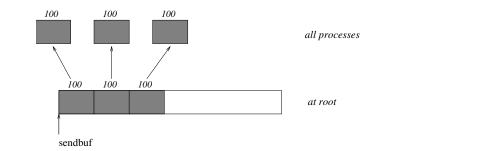


Figure 5.9: The root process scatters sets of 100 ints to each process in the group.

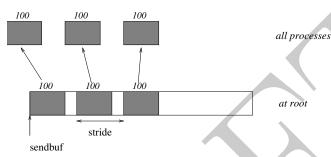


Figure 5.10: The root process scatters sets of 100 ints, moving by stride ints from send to send in the scatter.

MPI_Scatter(sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);

Example 5.12 The reverse of Example 5.5. The root process scatters sets of 100 ints to the other processes, but the sets of 100 are *stride ints* apart in the sending buffer. Requires use of MPI_SCATTERV. Assume *stride* \geq 100. See Figure 5.10.

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

 $\frac{24}{25}$

 31

Example 5.13 The reverse of Example 5.9. We have a varying stride between blocks at
 sending (root) side, at the receiving side we receive into the i-th column of a 100×150 C
 array. See Figure 5.11.

```
MPI_Comm comm;
5
         int gsize,recvarray[100][150],*rptr;
6
         int root, *sendbuf, myrank, *stride;
7
         MPI_Datatype rtype;
8
9
         int i, *displs, *scounts, offset;
10
          . . .
         MPI_Comm_size(comm, &gsize);
11
         MPI_Comm_rank(comm, &myrank);
12
13
         stride = (int *)malloc(gsize*sizeof(int));
14
15
          . . .
16
         /* stride[i] for i = 0 to gsize-1 is set somehow
           * sendbuf comes from elsewhere
17
           */
18
19
          . . .
         displs = (int *)malloc(gsize*sizeof(int));
20
         scounts = (int *)malloc(gsize*sizeof(int));
21
         offset = 0;
22
         for (i=0; i<gsize; ++i) {</pre>
23
^{24}
              displs[i] = offset;
              offset += stride[i];
25
26
              scounts[i] = 100 - i;
         }
27
          /* Create datatype for the column we are receiving
28
           */
29
         MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &rtype);
30
         MPI_Type_commit(&rtype);
31
         rptr = &recvarray[0][myrank];
32
         MPI_Scatterv(sendbuf, scounts, displs, MPI_INT, rptr, 1, rtype,
33
34
                        root, comm);
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

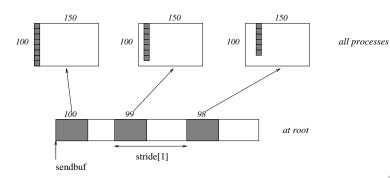


Figure 5.11: The root scatters blocks of 100-i ints into column i of a 100×150 C array. At the sending side, the blocks are stride[i] ints apart.

5.7 Gather-to-all

MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)				
IN	sendbuf	starting address of send buffer (choice)	18 19	
IN	sendcount		20	
IIN	senacount	number of elements in send buffer (non-negative integer)	21	
			22	
IN	sendtype	data type of send buffer elements (handle)	23	
OUT	recvbuf	address of receive buffer (choice)	24	
IN	recvcount	number of elements received from any process	25	
		(non-negative integer)	26	
IN	recvtype	data type of receive buffer elements (handle)	27	
			28	
IN	comm	communicator (handle)	29	
			30	
C binding			31 32	
<pre>int MPI_Allgather(const void *sendbuf, int sendcount,</pre>			32	
			34	
MPI_Datatype recvtype, MPI_Comm comm)			35	
Fortran 2	2008 binding		36	
MPI_Allga		sendtype, recvbuf, recvcount, recvtype,	37	
	comm, ierror)		38	
	(*), DIMENSION(), INTEN		39	
	ER, INTENT(IN) :: sendcou		40	
	(MPI_Datatype), INTENT(IN)		41	
	(*), DIMENSION() :: recu		42	
	TYPE(MPI_Comm), INTENT(IN) :: comm			
INTEG	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
Fortran binding			45	
MPI_ALLGA	MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,			
	COMM, IERROR)		47	
<type> SENDBUF(*), RECVBUF(*) 44</type>				

 $\mathbf{2}$

1	INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
2	
3	MPI_ALLGATHER can be thought of as MPI_GATHER, but where all processes receive
4	the result, instead of just the root. The block of data sent from the j-th process is received
5	by every process and placed in the j-th block of the buffer recvbuf.
6	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.
7	If comm is an intracommunicator, the outcome of a call to MPI_ALLGATHER() is as
8	if all processes executed n calls to
9	
10 11	MPI_Gather(sendbuf,sendcount,sendtype,recvbuf,recvcount,
12	recvtype,root,comm)
13	for root = 0,, n-1. The rules for correct usage of MPI_ALLGATHER are easily found
14	from the corresponding rules for MPI_GATHER.
15	The "in place" option for intracommunicators is specified by passing the value
16	MPI_IN_PLACE to the argument sendbuf at all processes. sendcount and sendtype are ignored.
17	Then the input data of each process is assumed to be in the area where that process would
18	receive its own contribution to the receive buffer.
19	If comm is an intercommunicator, then each process of one group (group A) contributes
20	sendcount data items; these data are concatenated and the result is stored at each process
21	in the other group (group B). Conversely the concatenation of the contributions of the processes in group A. The cond buffer arguments in
22 23	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	group A must be consistent with the receive buner arguments in group D, and vice versa.
25	Advice to users. The communication pattern of MPI_ALLGATHER executed on an
26	intercommunication domain need not be symmetric. The number of items sent by
27	processes in group A (as specified by the arguments sendcount, sendtype in group A
28	and the arguments recvcount, recvtype in group B), need not equal the number of
29	items sent by processes in group B (as specified by the arguments sendcount, sendtype
30	in group B and the arguments recvcount, recvtype in group A). In particular, one can
31	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	in the reverse direction. (End of dubice to users.)
34	
35	
36	
37	
38	
39	
40	
41 42	
42 43	
44	
45	
46	
47	
48	

MPI_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, ¹					
	comm)		2		
IN	sendbuf	starting address of send buffer (choice)	3 4		
IN	sendcount	number of elements in send buffer (non-negative integer)	5 6		
IN	sendtype	data type of send buffer elements (handle)	7		
OUT	recvbuf	address of receive buffer (choice)	8		
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	data type of receive buffer elements (handle)	16 17		
IN	comm	communicator (handle)	18		
			19		
C bind	ing		20		
int MP	•	id *sendbuf, int sendcount,	21		
		sendtype, void *recvbuf, const int recvcounts[],	22		
	const int disp	ols[], MPI_Datatype recvtype, MPI_Comm comm)	23 24		
Fortra	n 2008 binding		24 25		
MPI_A1	-	dcount, sendtype, recvbuf, recvcounts, displs,	26		
	recvtype, comm		27		
		INTENT(IN) :: sendbuf	28		
		<pre>sendcount, recvcounts(*), displs(*) ENT(IN) :: conductor = record;</pre>	29		
	PE(*), DIMENSION()	ENT(IN) :: sendtype, recvtype	30		
			31		
TYPE(MPI_Comm), INTENT(IN) :: comm 32 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 32					
Fortran binding					
MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,					
RECVTYPE, COMM, IERROR) 36					
•	<type> SENDBUF(*), RECVBUF(*) 37 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, 38</type>				
±N.	IERROR		39		
	·		40		
		thought of as MPI_GATHERV, but where all processes re-	41		
ceive the result, instead of just the root. The block of data sent from the j-th process is					
received by every process and placed in the j-th block of the buffer recvbuf. These blocks need not all be the same size.					
need not all be the same size.					

The type signature associated with sendcount, sendtype, at process j must be equal to the type signature associated with recvcounts[j], recvtype at any other process.

If comm is an intracommunicator, the outcome is as if all processes executed calls to

45

46

47 48

Unofficial Draft for Comment Only

1 2	MPI_G	atherv(sendbuf,sendcount	<pre>,sendtype,recvbuf,recvcounts,displs,</pre>		
3 4 5 6 7 8 9 10 11 12 13 14	for root = 0,, n-1. The rules for correct usage of MPI_ALLGATHERV are easily found from the corresponding rules for MPI_GATHERV. The "in place" option for intracommunicators is specified by passing the value MPI_IN_PLACE to the argument sendbuf at all processes. In such a case, sendcount and sendtype are ignored, and the input data of each process is assumed to be in the area where that process would receive its own contribution to the receive buffer. If comm is an intercommunicator, then each process of one group (group A) contributes sendcount data items; these data are concatenated and the result is stored at each process in the other group (group B). Conversely the concatenation of the contributions of the processes in group B is stored at each process in group A. The send buffer arguments in group A must be consistent with the receive buffer arguments in group B, and vice versa.				
15 16	5.7.1 Example using MPI_ALLGATHER				
17 18	The example in this section uses intracommunicators.				
18 19 20 21	Example 5.14 The all-gather version of Example 5.2. Using MPI_ALLGATHER, we will gather 100 ints from every process in the group to every process.				
22 23 24	int g int *	omm comm; size,sendarray[100]; rbuf;			
25 26 27 28	 MPI_Comm_size(comm, &gsize); rbuf = (int *)malloc(gsize*100*sizeof(int)); MPI_Allgather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, comm);				
29 30	After	the call, every process has the	e group-wide concatenation of the sets of data.		
31 32 33 34	5.8 All-	to-All Scatter/Gather			
35	MPI_ALLT	OALL(sendbuf, sendcount, sen	dtype, recvbuf, recvcount, recvtype, comm)		
36 37	IN	sendbuf	starting address of send buffer (choice)		
38 39	IN	sendcount	number of elements sent to each process (non-negative integer)		
40	IN	sendtype	data type of send buffer elements (handle)		
41	OUT	recvbuf	address of receive buffer (choice)		
42 43 44	IN	recvcount	number of elements received from any process (non-negative integer)		
45	IN	recvtype	data type of receive buffer elements (handle)		
46 47	IN	comm	communicator (handle)		
47	C binding	y 5			

int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype, 2 void *recvbuf, int recvcount, MPI_Datatype recvtype, 3 MPI_Comm comm) Fortran 2008 binding MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 6 comm. ierror) TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf INTEGER, INTENT(IN) :: sendcount, recvcount 9 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 10 TYPE(*), DIMENSION(...) :: recvbuf 11 TYPE(MPI_Comm), INTENT(IN) :: comm 12INTEGER, OPTIONAL, INTENT(OUT) :: ierror 13 14 Fortran binding MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 1516COMM, IERROR) 17<type> SENDBUF(*), RECVBUF(*) 18 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR 19 MPI_ALLTOALL is an extension of MPI_ALLGATHER to the case where each process 20sends distinct data to each of the receivers. The j-th block sent from process i is received 21by process j and is placed in the i-th block of recvbuf. 22 The type signature associated with sendcount, sendtype, at a process must be equal to 23 the type signature associated with recvcount, recvtype at any other process. This implies 24that the amount of data sent must be equal to the amount of data received, pairwise between 25every pair of processes. As usual, however, the type maps may be different. 26If comm is an intracommunicator, the outcome is as if each process executed a send to 27each process (itself included) with a call to, 2829 MPI_Send(sendbuf+i· sendcount· extent(sendtype),sendcount,sendtype,i, ...), 30 31and a receive from every other process with a call to, 32 33 MPI_Recv(recvbuf+i· recvcount· extent(recvtype),recvcount,recvtype,i,...). 34 All arguments on all processes are significant. The argument comm must have identical 35 values on all processes. 36 The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE to 37 the argument sendbuf at *all* processes. In such a case, sendcount and sendtype are ignored. 38 The data to be sent is taken from the recvbuf and replaced by the received data. Data sent 39 and received must have the same type map as specified by recvcount and recvtype. 40 41 Rationale. For large MPI_ALLTOALL instances, allocating both send and receive 42buffers may consume too much memory. The "in place" option effectively halves the 43 application memory consumption and is useful in situations where the data to be sent 44will not be used by the sending process after the MPI_ALLTOALL exchange (e.g., in 45parallel Fast Fourier Transforms). (End of rationale.) 46 47Advice to implementors. Users may opt to use the "in place" option in order to

Unofficial Draft for Comment Only

	180	CH	HAPTER 5.	COLLECTIVE COMMUNICATION		
1 2 3		erve memory. Quality MPI impring. (End of advice to implementation)		s should thus strive to minimize system		
4 5 6	sends a me i in group	If comm is an intercommunicator, then the outcome is as if each process in group A adds a message to each process in group B, and vice versa. The j-th send buffer of process a group A should be consistent with the i-th receive buffer of process j in group B, and				
7 8	vice versa.					
9	Advi	ce to users. When a complete	te exchange i	is executed on an intercommunication		
10 11		ain, then the number of data	items sent fr	com processes in group A to processes ems sent in the reverse direction. In		
12 13	-	cular, one can have unidirection everse direction. (<i>End of advi</i>		nication by specifying $sendcount = 0$ in		
14						
15 16 17	MPI_ALLT	· ·	displs, sendty	pe, recvbuf, recvcounts, rdispls,		
18		recvtype, comm)				
19	IN	sendbuf	starting add	lress of send buffer (choice)		
20	IN	sendcounts	-	e integer array (of length group size)		
21 22			specifying th rank	he number of elements to send to each		
23 24 25 26	IN	sdispls	the displace	y (of length group size). Entry j specifies ment (relative to sendbuf) from which to tgoing data destined for process j		
20	IN	sendtype	data type of	f send buffer elements (handle)		
28	OUT	recvbuf	address of r	eceive buffer (choice)		
29 30 31	IN	recvcounts	specifying the	e integer array (of length group size) he number of elements that can be m each rank		
32 33 34	IN	rdispls	integer array the displace	y (of length group size). Entry i specifies ment (relative to recvbuf) at which to coming data from process i		
35 36	INI		-	ů ·		
37	IN	recvtype		f receive buffer elements (handle)		
38	IN	comm	communicat	cor (handle)		
39 40	C binding					
41	int MPI_A	lltoallv(const void *send	-	-		
42		const int sdispis[], const int recvcounts	•	<pre>ype sendtype, void *recvbuf, int rdispls[]</pre>		
43		MPI_Datatype recvtyp		-		
44 45	Fortran 9	2008 binding				
46		0	, sdispls.	sendtype, recybuf, recycounts.		
47	<pre>MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, ierror)</pre>					
48	TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf					

```
INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
            rdispls(*)
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..) :: recvbuf
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

MPI_ALLTOALLV adds flexibility to MPI_ALLTOALL in that the location of data for the send is specified by sdispls and the location of the placement of the data on the receive side is specified by rdispls.

If comm is an intracommunicator, then the j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf. These blocks need not all have the same size.

The type signature associated with sendcounts[j], sendtype at process i must be equal to the type signature associated with recvcounts[i], recvtype at process j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. Distinct type maps between sender and receiver are still allowed.

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

 $MPI_Send(sendbuf+sdispls[i] \cdot extent(sendtype), sendcounts[i], sendtype, i, ...),$

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

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

All arguments on all processes are significant. The argument **comm** must have identical values on all processes.

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE to the argument sendbuf at *all* processes. In such a case, sendcounts, sdispls and sendtype are ignored. The data to be sent is taken from the recvbuf and replaced by the received data. Data sent and received must have the same type map as specified by the recvcounts array and the recvtype, and is taken from the locations of the receive buffer specified by rdispls.

Advice to users. Specifying the "in place" option (which must be given on all processes) implies that the same amount and type of data is sent and received between any two processes in the group of the communicator. Different pairs of processes can exchange different amounts of data. Users must ensure that recvcounts[j] and recvtype on process i match recvcounts[i] and recvtype on process j. This symmetric exchange can be useful in applications where the data to be sent will not be used by the sending process after the MPI_ALLTOALLV exchange. (*End of advice to users.*)

If comm is an intercommunicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The j-th send buffer of process

Unofficial Draft for Comment Only

 $46 \\ 47$

1 2	i in grou vice vers	-	tent with the i-th receive buffer of process j in group B, and
3 4 5 6 7	fle: cat	xibility as one would a tions, with two except	ions of MPI_ALLTOALL and MPI_ALLTOALLV give as much achieve by specifying n independent, point-to-point communi- tions: all messages use the same datatype, and messages are ared to) sequential storage. (<i>End of rationale.</i>)
8 9 10 11 12 13	ter fro Me	m sender to receiver essages can be forward	Although the discussion of collective communication in operation implies that each message is transferred directly , implementations may use a tree communication pattern. led by intermediate nodes where they are split (for scatter) or c), if this is more efficient. (<i>End of advice to implementors.</i>)
14 15			
16 17	MPI_AL	LTOALLW(sendbuf, se recvtypes, com	ndcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, m)
18 19	IN	sendbuf	starting address of send buffer (choice)
20 21 22	IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank
23 24 25 26	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)
27 28 29 30	IN	sendtypes	array of datatypes (of length group size). Entry j specifies the type of data to send to process j (array of handles)
31	OUT	recvbuf	address of receive buffer (choice)
32 33 34	IN	recvcounts	non-negative integer array (of length group size) specifying the number of elements that can be received from each rank
35 36 37 38 39	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)
40 41 42	IN	recvtypes	array of datatypes (of length group size). Entry i specifies the type of data received from process i (array of handles)
43 44	IN	comm	communicator (handle)
45 46 47	C bindi int MPI	_Alltoallw(const v	oid *sendbuf, const int sendcounts[],
48		const int so	<pre>displs[], const MPI_Datatype sendtypes[],</pre>

Unofficial Draft for Comment Only

1 void *recvbuf, const int recvcounts[], const int rdispls[], 2 const MPI_Datatype recvtypes[], MPI_Comm comm) 3 Fortran 2008 binding 4 MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, 5 rdispls, recvtypes, comm, ierror) 6 TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*), rdispls(*) 9 TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*) 10 TYPE(*), DIMENSION(...) :: recvbuf 11 TYPE(MPI_Comm), INTENT(IN) :: comm 12INTEGER, OPTIONAL, INTENT(OUT) :: ierror 13 14Fortran binding MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS, 1516RDISPLS, RECVTYPES, COMM, IERROR) 17<type> SENDBUF(*), RECVBUF(*) 18 INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*), 19 RDISPLS(*), RECVTYPES(*), COMM, IERROR 20MPI_ALLTOALLW is the most general form of complete exchange. Like 21MPI_TYPE_CREATE_STRUCT, the most general type constructor, MPI_ALLTOALLW al-22 lows separate specification of count, displacement and datatype. In addition, to allow max-23 imum flexibility, the displacement of blocks within the send and receive buffers is specified 24in bytes. 25If comm is an intracommunicator, then the j-th block sent from process i is received by 26process j and is placed in the i-th block of recvbuf. These blocks need not all have the same 27size. 28 The type signature associated with sendcounts[j], sendtypes[j] at process i must be equal 29 to the type signature associated with recvcounts[i], recvtypes[i] at process j. This implies that 30 the amount of data sent must be equal to the amount of data received, pairwise between 31 every pair of processes. Distinct type maps between sender and receiver are still allowed. 32 The outcome is as if each process sent a message to every other process with 33 34 MPI_Send(sendbuf+sdispls[i],sendcounts[i],sendtypes[i],i,...), 3536 and received a message from every other process with a call to 37 38 MPI_Recv(recvbuf+rdispls[i],recvcounts[i],recvtypes[i],i,...). 39

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

Like for MPI_ALLTOALLV, the "in place" option for intracommunicators is specified by 42 passing MPI_IN_PLACE to the argument sendbuf at *all* processes. In such a case, sendcounts, 43 sdispls and sendtypes are ignored. The data to be sent is taken from the recvbuf and replaced 44 by the received data. Data sent and received must have the same type map as specified 45 by the recvounts and recvtypes arrays, and is taken from the locations of the receive buffer 46 specified by rdispls. 47

Unofficial Draft for Comment Only

40 41

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

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

5.9 Global Reduction Operations

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

5.9.1 Reduce

MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)

24	_	X / / /		
25	IN	sendbuf	address of send buffer (choice)	
26 27	OUT	recvbuf	address of receive buffer (choice, significant only at	
28			root)	
29	IN	count	number of elements in send buffer (non-negative	
30			integer)	
31	IN	datatype	data type of elements of send buffer (handle)	
32 33	IN	ор	reduce operation (handle)	
34	IN	root	rank of root process (integer)	
35	IN	comm	communicator (handle)	
36				
37	C binding			
38	<pre>int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,</pre>			
39	MP1_Datatype datatype, MP1_Op op, int root, MP1_Comm comm)			
40	Fortron 2008 hinding			
41	Fortran 2008 binding			
42	MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)			
43	TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf			
44		(*), DIMENSION() :: rec		
45		GER, INTENT(IN) :: count,		
46		(MPI_Datatype), INTENT(IN)		
47		(MPI_Op), INTENT(IN) :: op		
48	TYPE(MPI_Comm), INTENT(IN) :: comm			

 $\mathbf{2}$

 $\overline{7}$

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
 <type> SENDBUF(*), RECVBUF(*)
 INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR

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

Section 5.9.2, lists the set of predefined operations provided by MPI. That section also enumerates the datatypes to which each operation can be applied.

In addition, users may define their own operations that can be overloaded to operate on several datatypes, either basic or derived. This is further explained in Section 5.9.5.

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

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

Advice to users. Some applications may not be able to ignore the non-associative nature of floating-point operations or may use user-defined operations (see Section 5.9.5) that require a special reduction order and cannot be treated as associative. Such applications should enforce the order of evaluation explicitly. For example, in the case of operations that require a strict left-to-right (or right-to-left) evaluation order, this could be done by gathering all operands at a single process (e.g., with MPI_GATHER), applying the reduction operation in the desired order (e.g., with MPI_REDUCE_LOCAL), and if needed, broadcast or scatter the result to the other processes (e.g., with MPI_BCAST). (End of advice to users.)

The datatype argument of MPI_REDUCE must be compatible with op. Predefined operators work only with the MPI types listed in Section 5.9.2 and Section 5.9.4. Furthermore, the datatype and op given for predefined operators must be the same on all processes.

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

42

43

44

Note that it is possible for users to supply different user-defined operations to
 MPI_REDUCE in each process. MPI does not define which operations are used on which
 operands in this case. User-defined operators may operate on general, derived datatypes.
 In this case, each argument that the reduce operation is applied to is one element described
 by such a datatype, which may contain several basic values. This is further explained in
 Section 5.9.5.

Advice to users. Users should make no assumptions about how MPI_REDUCE is implemented. It is safest to ensure that the same function is passed to MPI_REDUCE by each process. (*End of advice to users.*)

Overlapping datatypes are permitted in "send" buffers. Overlapping datatypes in "receive" buffers are erroneous and may give unpredictable results.

The "in place" option for intracommunicators is specified by passing the value MPI_IN_PLACE to the argument sendbuf at the root. In such a case, the input data is taken at the root from the receive buffer, where it will be replaced by the output data.

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

22 23

 24

7

8

9

10 11

12

13

14

15

16

17

18

19

20

21

5.9.2 Predefined Reduction Operations

The following predefined operations are supplied for MPI_REDUCE and related functions
 MPI_ALLREDUCE, MPI_REDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER,
 MPI_SCAN, MPI_EXSCAN, all nonblocking variants of those (see Section 5.12), and
 MPI_REDUCE_LOCAL. These operations are invoked by placing the following in op.

30		
31	Name	Meaning
32		
33	MPI_MAX	maximum
34	MPI_MIN	minimum
35	MPI_SUM	sum
36	MPI_PROD	product
37	MPI_LAND	logical and
38	MPI_BAND	bit-wise and
39	MPI_LOR	logical or
40	MPI_BOR	bit-wise or
41	MPI_LXOR	logical exclusive or (xor)
42	MPI_BXOR	bit-wise exclusive or (xor)
42	MPI_MAXLOC	max value and location
	MPI_MINLOC	min value and location
44		

The two operations MPI_MINLOC and MPI_MAXLOC are discussed separately in Section 5.9.4. For the other predefined operations, we enumerate below the allowed combinations of op and datatype arguments. First, define groups of MPI basic datatypes in the following way.

		1
C integer:	MPI_INT, MPI_LONG, MPI_SHORT,	2
-	MPI_UNSIGNED_SHORT, MPI_UNSIGNED,	3
	MPI_UNSIGNED_LONG,	4
	MPI_LONG_LONG_INT,	5
	MPI_LONG_LONG (as synonym),	6
	MPI_UNSIGNED_LONG_LONG,	7
	MPI_SIGNED_CHAR,	8
	MPI_UNSIGNED_CHAR,	9
	MPI_INT8_T, MPI_INT16_T,	10
	MPI_INT32_T, MPI_INT64_T,	11
	MPI_UINT8_T, MPI_UINT16_T,	12
	MPI_UINT32_T, and MPI_UINT64_T	13
Fortran integer:	MPI_INTEGER	14
	and handles returned from	15
	MPI_TYPE_CREATE_F90_INTEGER	16
	and, if available, MPI_INTEGER1,	17
	MPI_INTEGER2, MPI_INTEGER4,	18
	MPI_INTEGER8, and MPI_INTEGER16	19
Floating point:	MPI_FLOAT, MPI_DOUBLE, MPI_REAL,	20
	MPI_DOUBLE_PRECISION,	21
	MPI_LONG_DOUBLE,	22
	and handles returned from	23
	MPI_TYPE_CREATE_F90_REAL and, if available, MPI_REAL2,	24
	MPI_REAL4, MPI_REAL8, and MPI_REAL16	25
Logical	MPI_LOGICAL, MPI_C_BOOL,	26
Logical:	and MPI_CXX_BOOL	27
Complex:	MPI_COMPLEX, MPI_C_COMPLEX,	28
complex.	MPI_C_FLOAT_COMPLEX (as synonym),	29
	MPI_C_DOUBLE_COMPLEX,	30
	MPI_C_LONG_DOUBLE_COMPLEX,	31
	MPI_CXX_FLOAT_COMPLEX,	32
	MPI_CXX_DOUBLE_COMPLEX,	33
	MPI_CXX_LONG_DOUBLE_COMPLEX,	34
	and handles returned from	35
	MPI_TYPE_CREATE_F90_COMPLEX	36
	and, if available, MPI_DOUBLE_COMPLEX,	37
	MPI_COMPLEX4, MPI_COMPLEX8,	38
	MPI_COMPLEX16, and MPI_COMPLEX32	39
Byte:	MPI_BYTE	40
Multi-language types:	MPI_AINT, MPI_OFFSET, and MPI_COUNT	41
Now, the valid datatypes for each op	poration are specified below	42
now, the value datatypes for each op	cranon are specified below.	43
		44
Ор	Allowed Types	45
~r		46
MPI_MAX, MPI_MIN	C integer, Fortran integer, Floating point,	47
_ , _	Multi-language types	48

Unofficial Draft for Comment Only

```
1
       MPI_SUM, MPI_PROD
                                              C integer, Fortran integer, Floating point, Complex,
\mathbf{2}
                                              Multi-language types
3
       MPI_LAND, MPI_LOR, MPI_LXOR
                                              C integer, Logical
4
       MPI_BAND, MPI_BOR, MPI_BXOR
                                              C integer, Fortran integer, Byte, Multi-language types
5
          These operations together with all listed datatypes are valid in all supported program-
6
     ming languages, see also Reduce Operations on page 761 in Section 18.2.6.
7
          The following examples use intracommunicators.
8
9
     Example 5.15 A routine that computes the dot product of two vectors that are distributed
10
     across a group of processes and returns the answer at node zero.
11
12
     SUBROUTINE PAR_BLAS1(m, a, b, c, comm)
13
     REAL a(m), b(m)
                               ! local slice of array
14
     REAL c
                               ! result (at node zero)
15
     REAL sum
16
     INTEGER m, comm, i, ierr
17
18
     ! local sum
19
     sum = 0.0
20
     DO i = 1, m
21
         sum = sum + a(i)*b(i)
22
     END DO
23
^{24}
     ! global sum
25
     CALL MPI_REDUCE(sum, c, 1, MPI_REAL, MPI_SUM, 0, comm, ierr)
26
     RETURN
27
     END
28
29
     Example 5.16 A routine that computes the product of a vector and an array that are
30
     distributed across a group of processes and returns the answer at node zero.
^{31}
32
     SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
33
     REAL a(m), b(m,n)
                              ! local slice of array
34
     REAL c(n)
                              ! result
35
     REAL sum(n)
36
     INTEGER n, comm, i, j, ierr
37
38
     ! local sum
39
     DO j=1,n
40
         sum(j) = 0.0
41
         DO i=1,m
42
            sum(j) = sum(j) + a(i)*b(i,j)
43
         END DO
44
     END DO
45
46
     ! global sum
47
     CALL MPI_REDUCE(sum, c, n, MPI_REAL, MPI_SUM, 0, comm, ierr)
48
```

CHAPTER 5. COLLECTIVE COMMUNICATION

! return result at node zero (and garbage at the other nodes) RETURN END

5.9.3 Signed Characters and Reductions

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

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

5.9.4 MINLOC and MAXLOC

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

The operation that defines MPI_MAXLOC is:

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

where

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

and

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

MPI_MINLOC is defined similarly:

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$
⁴¹
⁴²
⁴³

where

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

Unofficial Draft for Comment Only

 24

and

1

ſ	i	if $u < v$
$k = \langle$	$\min(i, j)$	if u = v
	j	if u > v

6 Both operations are associative and commutative. Note that if MPI_MAXLOC is applied 7to reduce a sequence of pairs $(u_0, 0), (u_1, 1), \ldots, (u_{n-1}, n-1)$, then the value returned is 8 (u, r), where $u = \max_i u_i$ and r is the index of the first global maximum in the sequence. 9 Thus, if each process supplies a value and its rank within the group, then a reduce operation 10 with $op = MPI_MAXLOC$ will return the maximum value and the rank of the first process with 11that value. Similarly, MPI_MINLOC can be used to return a minimum and its index. More 12generally, MPI_MINLOC computes a *lexicographic minimum*, where elements are ordered 13according to the first component of each pair, and ties are resolved according to the second 14component.

¹⁵ The reduce operation is defined to operate on arguments that consist of a pair: value ¹⁶ and index. For both Fortran and C, types are provided to describe the pair. The potentially ¹⁷ mixed-type nature of such arguments is a problem in Fortran. The problem is circumvented, ¹⁸ for Fortran, by having the MPI-provided type consist of a pair of the same type as value, ¹⁹ and coercing the index to this type also. In C, the MPI-provided pair type has distinct ²⁰ types and the index is an int.

In order to use MPI_MINLOC and MPI_MAXLOC in a reduce operation, one must provide
 a datatype argument that represents a pair (value and index). MPI provides nine such
 predefined datatypes. The operations MPI_MAXLOC and MPI_MINLOC can be used with
 each of the following datatypes.

26	Fortran:		
27	Name	Description	
28	MPI_2REAL	pair of REALs	
29	MPI_2DOUBLE_PRECISION	pair of DOUBLE PRECISION variables	
30	MPI_2INTEGER	pair of INTEGERS	
31			
32			
33	C:		
34	Name	Description	
35	MPI_FLOAT_INT	float and int	
36	MPI_DOUBLE_INT	double and int	
37	MPI_LONG_INT	long and int	
38	MPI_2INT	pair of int	
39	MPI_SHORT_INT	short and int	
40	MPI_LONG_DOUBLE_INT	long double and int	
41	The datatype MPI_2REAL is as if defined by the following (see Section 4.1).		
42	The datatype MFT_2NLAL is us if defined by the following (see Section 4.1).		
43	MPI_Type_contiguous(2, MPI_R	EAL. MPI 2REAL):	
44		,,	
45	Similar statements apply for N	MPI_2INTEGER, MPI_2DOUBLE_PRECISION, and MPI_2INT.	
46	The datatype MPI_SHORT_IN	Γ is as if defined by the following sequence of instructions.	
47	v *		
48			

```
1
struct mystruct {
                                                                                           \mathbf{2}
    short val;
                                                                                           3
    int rank;
};
                                                                                           4
type[0] = MPI_SHORT;
                                                                                           5
                                                                                           6
type[1] = MPI_INT;
disp[0] = 0;
disp[1] = offsetof(struct mystruct, rank);
block[0] = 1;
                                                                                          10
block[1] = 1;
                                                                                          11
MPI_Type_create_struct(2, block, disp, type, MPI_SHORT_INT);
                                                                                          12
Similar statements apply for MPI_FLOAT_INT, MPI_LONG_INT and MPI_DOUBLE_INT.
                                                                                          13
    The following examples use intracommunicators.
                                                                                          14
                                                                                          15
Example 5.17 Each process has an array of 30 doubles, in C. For each of the 30 locations,
                                                                                          16
compute the value and rank of the process containing the largest value.
                                                                                          17
                                                                                          18
     . . .
                                                                                          19
    /* each process has an array of 30 double: ain[30]
                                                                                          20
      */
                                                                                          21
    double ain[30], aout[30];
                                                                                          22
    int ind[30];
                                                                                          23
    struct {
                                                                                          ^{24}
         double val;
                                                                                          25
         int
                rank;
                                                                                          26
    } in[30], out[30];
                                                                                          27
    int i, myrank, root;
                                                                                          28
                                                                                          29
    MPI_Comm_rank(comm, &myrank);
                                                                                          30
    for (i=0; i<30; ++i) {</pre>
                                                                                          31
         in[i].val = ain[i];
                                                                                          32
         in[i].rank = myrank;
                                                                                          33
                                                                                          34
    MPI_Reduce(in, out, 30, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm);
                                                                                          35
    /* At this point, the answer resides on process root
                                                                                          36
      */
                                                                                          37
    if (myrank == root) {
                                                                                          38
         /* read ranks out
                                                                                          39
          */
                                                                                          40
         for (i=0; i<30; ++i) {
                                                                                          41
             aout[i] = out[i].val;
                                                                                          42
             ind[i] = out[i].rank;
                                                                                          43
         }
                                                                                          44
    }
                                                                                          45
                                                                                          46
```

Example 5.18 Same example, in Fortran.

47

```
1
      . . .
\mathbf{2}
     ! each process has an array of 30 double: ain(30)
3
4
     DOUBLE PRECISION ain(30), aout(30)
\mathbf{5}
     INTEGER ind(30)
6
     DOUBLE PRECISION in(2,30), out(2,30)
7
     INTEGER i, myrank, root, ierr
8
9
     CALL MPI_COMM_RANK(comm, myrank, ierr)
10
     DO i=1,30
^{11}
         in(1,i) = ain(i)
12
         in(2,i) = myrank
                               ! myrank is coerced to a double
13
     END DO
14
15
     CALL MPI_REDUCE(in, out, 30, MPI_2DOUBLE_PRECISION, MPI_MAXLOC, root, &
16
                       comm, ierr)
17
     ! At this point, the answer resides on process root
18
19
     IF (myrank .EQ. root) THEN
20
         ! read ranks out
21
        DO i=1,30
22
            aout(i) = out(1,i)
23
            ind(i) = out(2,i) ! rank is coerced back to an integer
^{24}
        END DO
25
     END IF
26
27
     Example 5.19 Each process has a non-empty array of values. Find the minimum global
28
     value, the rank of the process that holds it and its index on this process.
29
30
     #define LEN
                      1000
31
32
                               /* local array of values */
     float val[LEN];
33
                               /* local number of values */
     int count;
34
     int myrank, minrank, minindex;
35
     float minval;
36
37
     struct {
38
          float value;
39
          int
                 index;
40
     } in, out;
41
42
          /* local minloc */
43
     in.value = val[0];
44
     in.index = 0;
45
     for (i=1; i < count; i++)</pre>
46
          if (in.value > val[i]) {
47
              in.value = val[i];
48
```

```
1
         in.index = i;
                                                                                          2
    }
    /* global minloc */
MPI_Comm_rank(comm, &myrank);
in.index = myrank*LEN + in.index;
MPI_Reduce(&in, &out, 1, MPI_FLOAT_INT, MPI_MINLOC, root, comm);
    /* At this point, the answer resides on process root
     */
                                                                                          10
if (myrank == root) {
                                                                                          11
    /* read answer out
     */
                                                                                          12
                                                                                          13
    minval = out.value;
                                                                                          14
    minrank = out.index / LEN;
                                                                                          15
    minindex = out.index % LEN;
                                                                                          16
}
                                                                                          17
     Rationale.
                  The definition of MPI_MINLOC and MPI_MAXLOC given here has the
                                                                                          18
     advantage that it does not require any special-case handling of these two operations:
                                                                                          19
     they are handled like any other reduce operation. A programmer can provide his or
                                                                                          20
     her own definition of MPI_MAXLOC and MPI_MINLOC, if so desired. The disadvantage
                                                                                          21
     is that values and indices have to be first interleaved, and that indices and values have
                                                                                          22
     to be coerced to the same type, in Fortran. (End of rationale.)
                                                                                          23
                                                                                          ^{24}
5.9.5 User-Defined Reduction Operations
                                                                                          25
                                                                                          26
                                                                                          27
                                                                                          28
MPI_OP_CREATE(user_fn, commute, op)
                                                                                          29
  IN
                                        user defined function (function)
           user_fn
                                                                                          30
  IN
           commute
                                       true if commutative; false otherwise.
                                                                                          31
                                                                                          32
  OUT
           op
                                       operation (handle)
                                                                                          33
                                                                                          34
C binding
                                                                                          35
int MPI_Op_create(MPI_User_function *user_fn, int commute, MPI_Op *op)
                                                                                          36
Fortran 2008 binding
                                                                                          37
                                                                                          38
MPI_Op_create(user_fn, commute, op, ierror)
                                                                                          39
    PROCEDURE(MPI_User_function) :: user_fn
    LOGICAL, INTENT(IN) :: commute
                                                                                          40
                                                                                          41
    TYPE(MPI_Op), INTENT(OUT) :: op
                                                                                          42
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                          43
Fortran binding
                                                                                          44
MPI_OP_CREATE(USER_FN, COMMUTE, OP, IERROR)
                                                                                          45
    EXTERNAL USER_FN
                                                                                          46
    LOGICAL COMMUTE
                                                                                          47
    INTEGER OP, IERROR
                                                                                          48
```

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 5.12), and
5	MPI_REDUCE_LOCAL. The user-defined operation is assumed to be associative. If commute
6	= true, then the operation should be both commutative and associative. If commute $=$ false,
7	then the order of operands is fixed and is defined to be in ascending, process rank order,
8	beginning with process zero. The order of evaluation can be changed, talking advantage of
9	
	the associativity of the operation. If $commute = true$ then the order of evaluation can be
10	changed, taking advantage of commutativity and associativity.
11	The argument $user_fn$ is the user-defined function, which must have the following four
12	arguments: invec, inoutvec, len, and datatype.
13	The ISO C prototype for the function is the following.
14	typedef void MPI_User_function(void *invec, void *inoutvec, int *len,
15	
	<pre>MPI_Datatype *datatype);</pre>
16	The Fortran declarations of the user-defined function user_fn appear below.
17	ABSTRACT INTERFACE
18	
19	SUBROUTINE MPI_User_function(invec, inoutvec, len, datatype)
20	USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
21	TYPE(C_PTR), VALUE :: invec, inoutvec
22	INTEGER :: len
	TYPE(MPI_Datatype) :: datatype
23	
24	SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, DATATYPE)
25	<type> INVEC(LEN), INOUTVEC(LEN)</type>
26	INTEGER LEN, DATATYPE
27	
28	The datatype argument is a handle to the data type that was passed into the call to
29	MPI_REDUCE. The user reduce function should be written such that the following holds:
30	Let u[0],, u[len-1] be the len elements in the communication buffer described by the
	arguments invec, len and datatype when the function is invoked; let $v[0], \ldots, v[len-1]$ be len
31	elements in the communication buffer described by the arguments inoutvec, len and datatype
32	
33	when the function is invoked; let $w[0], \ldots, w[len-1]$ be len elements in the communication
34	buffer described by the arguments inoutvec, len and datatype when the function returns;
35	then $w[i] = u[i] \circ v[i]$, for $i=0$,, len-1, where \circ is the reduce operation that the function
36	computes.
37	Informally, we can think of invec and inoutvec as arrays of len elements that user_fn
38	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
39	on len elements: i.e., the function returns in inoutvec[i] the value invec[i] \circ inoutvec[i], for
40	
41	i=0,, count-1, where \circ is the combining operation computed by the function.
42	
43	<i>Rationale.</i> The len argument allows MPI_REDUCE to avoid calling the function for
44	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
45	with Fortran.
46	
47	By internally comparing the value of the datatype argument to known, global handles,
48	it is possible to overload the use of a single user-defined function for several, different

Unofficial Draft for Comment Only

5.9. GLODAL REDUCTION OF ERATIONS 195	
data types. (End of rationale.)	1
	2
General datatypes may be passed to the user function. However, use of datatypes that	3
are not contiguous is likely to lead to inefficiencies.	4
No MPI communication function may be called inside the user function. MPI_ABORT	5
may be called inside the function in case of an error.	6 7
Advice to users. Suppose one defines a library of user-defined reduce functions that	8
are overloaded: the datatype argument is used to select the right execution path at each	9
invocation, according to the types of the operands. The user-defined reduce function	10
cannot "decode" the datatype argument that it is passed, and cannot identify, by itself,	11
the correspondence between the datatype handles and the datatype they represent.	12
This correspondence was established when the datatypes were created. Before the	13
library is used, a library initialization preamble must be executed. This preamble	14
code will define the datatypes that are used by the library, and store handles to these	15
datatypes in global, static variables that are shared by the user code and the library	16
code.	17
The Fortran version of MPI_REDUCE will invoke a user-defined reduce function using	18
the Fortran calling conventions and will pass a Fortran-type datatype argument; the	19
C version will use C calling convention and the C representation of a datatype handle.	20
Users who plan to mix languages should define their reduction functions accordingly.	21
(End of advice to users.)	22
	23
Advice to implementors. We outline below a naive and inefficient implementation of	24
MPI_REDUCE not supporting the "in place" option.	25 26
	20 27
<pre>MPI_Comm_size(comm, &groupsize);</pre>	28
<pre>MPI_Comm_rank(comm, &rank);</pre>	29
if $(rank > 0)$ {	30
MPI_Recv(tempbuf, count, datatype, rank-1,);	31
<pre>User_reduce(tempbuf, sendbuf, count, datatype); }</pre>	32
if (rank < groupsize-1) {	33
MPI_Send(sendbuf, count, datatype, rank+1,);	34
}	35
/* answer now resides in process groupsize-1 now send to root	36
*/	37
<pre>if (rank == root) {</pre>	38
<pre>MPI_Irecv(recvbuf, count, datatype, groupsize-1,, &req);</pre>	39
}	40
if (rank == groupsize-1) {	41
MDT Q = 1/2 = 1 + f = + - 1 + - + + -)	42

MPI_Send(sendbuf, count, datatype, root, ...);

}

}

if (rank == root) {

MPI_Wait(&req, &status);

43

44

45

46

```
1
           The reduction computation proceeds, sequentially, from process 0 to process
2
           groupsize-1. This order is chosen so as to respect the order of a possibly non-
3
           commutative operator defined by the function User_reduce(). A more efficient im-
4
           plementation is achieved by taking advantage of associativity and using a logarithmic
5
           tree reduction. Commutativity can be used to advantage, for those cases in which
6
           the commute argument to MPI_OP_CREATE is true. Also, the amount of temporary
7
           buffer required can be reduced, and communication can be pipelined with computa-
8
           tion, by transferring and reducing the elements in chunks of size len <count.
9
           The predefined reduce operations can be implemented as a library of user-defined
10
           operations. However, better performance might be achieved if MPI_REDUCE handles
11
           these functions as a special case. (End of advice to implementors.)
12
13
14
     MPI_OP_FREE(op)
15
16
       INOUT
                                             operation (handle)
                 op
17
18
     C binding
19
     int MPI_Op_free(MPI_Op *op)
20
21
     Fortran 2008 binding
     MPI_Op_free(op, ierror)
22
          TYPE(MPI_Op), INTENT(INOUT) :: op
23
24
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
25
     Fortran binding
26
     MPI_OP_FREE(OP, IERROR)
27
          INTEGER OP, IERROR
28
          Marks a user-defined reduction operation for deallocation and sets op to MPI_OP_NULL.
29
30
^{31}
     Example of User-Defined Reduce
32
     It is time for an example of user-defined reduction. The example in this section uses an
33
     intracommunicator.
34
35
     Example 5.20 Compute the product of an array of complex numbers, in C.
36
37
     typedef struct {
38
          double real, imag;
39
     } Complex;
40
41
     /* the user-defined function
42
      */
43
     void myProd(void *inP, void *inoutP, int *len, MPI_Datatype *dptr)
44
     ſ
45
          int i;
46
          Complex c;
47
          Complex *in = (Complex *)inP, *inout = (Complex *)inoutP;
48
```

CHAPTER 5. COLLECTIVE COMMUNICATION

196

Unofficial Draft for Comment Only

```
1
    for (i=0; i< *len; ++i) {</pre>
                                                                                       \mathbf{2}
        c.real = inout->real*in->real -
                                                                                       3
                    inout->imag*in->imag;
        c.imag = inout->real*in->imag +
                                                                                       4
                    inout->imag*in->real;
                                                                                       5
                                                                                       6
        *inout = c;
        in++; inout++;
                                                                                       7
    }
}
                                                                                       9
                                                                                       10
                                                                                       11
/* and, to call it...
 */
                                                                                       12
                                                                                       13
. . .
                                                                                       14
                                                                                       15
    /* each process has an array of 100 Complexes
                                                                                       16
     */
                                                                                       17
    Complex a[100], answer[100];
                                                                                       18
    MPI_Op myOp;
                                                                                       19
    MPI_Datatype ctype;
                                                                                       20
                                                                                      21
    /* explain to MPI how type Complex is defined
                                                                                       22
     */
    MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
                                                                                      23
                                                                                       ^{24}
    MPI_Type_commit(&ctype);
                                                                                       25
    /* create the complex-product user-op
                                                                                       26
     */
    MPI_Op_create(myProd, 1, &myOp);
                                                                                       27
                                                                                       28
                                                                                       29
    MPI_Reduce(a, answer, 100, ctype, myOp, root, comm);
                                                                                       30
                                                                                       31
    /* At this point, the answer, which consists of 100 Complexes,
                                                                                       32
     * resides on process root
                                                                                       33
     */
                                                                                      34
                                                                                      35
Example 5.21 How to use the mpi_f08 interface of the Fortran MPI_User_function.
                                                                                      36
                                                                                      37
subroutine my_user_function(invec, inoutvec, len, type)
                                                               bind(c)
                                                                                       38
   use, intrinsic :: iso_c_binding, only : c_ptr, c_f_pointer
                                                                                       39
   use mpi_f08 (
                                                                                       40
   type(c_ptr), value :: invec, inoutvec
                                                                                       41
   integer :: len
   type(MPI_Datatype) :: type
                                                                                       42
   real, pointer :: invec_r(:), inoutvec_r(:)
                                                                                       43
                                                                                       44
   if (type%MPI_VAL == MPI_REAL%MPI_VAL) then
      call c_f_pointer(invec, invec_r, (/ len /))
                                                                                       45
                                                                                       46
      call c_f_pointer(inoutvec, inoutvec_r, (/ len /))
                                                                                       47
      inoutvec_r = invec_r + inoutvec_r
                                                                                       48
```

Unofficial Draft for Comment Only

end if

1	end subroutine				
3	5.9.6 All-Reduce				
4 5 6 7 8	MPI inclue in a group	MPI includes a variant of the reduce operations where the result is returned to all processes in a group. MPI requires that all processes from the same group participating in these operations receive identical results.			
9 10	MPI_ALL	REDUCE(sendbuf, recvbuf, cou	nt, datatype, op, comm)		
11	IN	sendbuf	starting address of send buffer (choice)		
12	OUT	recvbuf	starting address of receive buffer (choice)		
13 14 15	IN	count	number of elements in send buffer (non-negative integer)		
16	IN	datatype	data type of elements of send buffer (handle)		
17	IN	ор	operation (handle)		
18 19	IN	comm	communicator (handle)		
21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37	<pre>int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,</pre>				
38 39 40 41 42 43 44 45 46 47 48	MPI_RED Adv duce perfe The MPI_IN_PI	UCE except that the result applice to implementors. The are, followed by a broadcast. He ormance. (End of advice to im "in place" option for intracom LACE to the argument sendbu	MPI_ALLREDUCE behaves the same as pears in the receive buffer of all the group members. All-reduce operations can be implemented as a re- powever, a direct implementation can lead to better <i>aplementors.</i>) amunicators is specified by passing the value f at all processes. In this case, the input data is puffer, where it will be replaced by the output data.		

If comm is an intercommunicator, then the result of the reduction of the data provided by processes in group A is stored at each process in group B, and vice versa. Both groups should provide count and datatype arguments that specify the same type signature.

The following example uses an intracommunicator.

Example 5.22 A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 5.16).

```
SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
REAL a(m), b(m,n)
                     ! local slice of array
REAL c(n)
                      ! result
REAL sum(n)
INTEGER n, comm, i, j, ierr
! local sum
DO j=1,n
   sum(j) = 0.0
   DO i=1,m
      sum(j) = sum(j) + a(i)*b(i,j)
   END DO
END DO
! global sum
CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, comm, ierr)
! return result at all nodes
RETURN
END
```

5.9.7 Process-Local Reduction

The functions in this section are of importance to library implementors who may want to implement special reduction patterns that are otherwise not easily covered by the standard MPI operations.

The following function applies a reduction operator to local arguments.

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

1 MPI_REDUCE_LOCAL(inbuf, inoutbuf, count, datatype, op) 2 IN inbuf input buffer (choice) 3 INOUT inoutbuf combined input and output buffer (choice) 4 5IN number of elements in inbuf and inoutbuf buffers count 6 (non-negative integer) 7 IN datatype data type of elements of inbuf and inoutbuf buffers 8 (handle) 9 operation (handle) IN ор 10 11 C binding 12int MPI_Reduce_local(const void *inbuf, void *inoutbuf, int count, 13 14MPI_Datatype datatype, MPI_Op op) 15Fortran 2008 binding 16MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror) 17TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf 18 TYPE(*), DIMENSION(..) :: inoutbuf 19 INTEGER, INTENT(IN) :: count 20TYPE(MPI_Datatype), INTENT(IN) :: datatype 21TYPE(MPI_Op), INTENT(IN) :: op 22 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 23 24 Fortran binding 25MPI_REDUCE_LOCAL(INBUF, INOUTBUF, COUNT, DATATYPE, OP, IERROR) 26<type> INBUF(*), INOUTBUF(*) INTEGER COUNT, DATATYPE, OP, IERROR 2728The function applies the operation given by op element-wise to the elements of inbuf 29 and inoutbuf with the result stored element-wise in inoutbuf, as explained for user-defined 30 operations in Section 5.9.5. Both inbuf and inoutbuf (input as well as result) have the 31 same number of elements given by count and the same datatype given by datatype. The 32 MPI_IN_PLACE option is not allowed. 33 Reduction operations can be queried for their commutativity. 34 35 36 MPI_OP_COMMUTATIVE(op, commute) 37 IN operation (handle) ор 38 OUT commute true if op is commutative, false otherwise (logical) 39 40 41 C binding 42int MPI_Op_commutative(MPI_Op op, int *commute) 43 Fortran 2008 binding 44 MPI_Op_commutative(op, commute, ierror) 45 TYPE(MPI_Op), INTENT(IN) :: op 46 LOGICAL, INTENT(OUT) :: commute 47 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 48

Fortran binding MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR) INTEGER OP, IERROR LOGICAL COMMUTE

5.10 Reduce-Scatter

MPI includes variants of the reduce operations where the result is scattered to all processes in a group on return. One variant scatters equal-sized blocks to all processes, while another variant scatters blocks that may vary in size for each process.

5.10.1 MPI_REDUCE_SCATTER_BLOCK

1516MPI_REDUCE_SCATTER_BLOCK(sendbuf, recvbuf, recvcount, datatype, op, comm) 17 sendbuf starting address of send buffer (choice) IN 18 OUT recvbuf starting address of receive buffer (choice) 19 20element count per block (non-negative integer) IN recvcount 21data type of elements of send and receive buffers IN datatype 22 (handle) 23IN operation (handle) 24 op 25IN comm communicator (handle) 2627C binding 28 int MPI_Reduce_scatter_block(const void *sendbuf, void *recvbuf, 29 int recvcount, MPI_Datatype datatype, MPI_Op op, 30 MPI_Comm comm) 31 Fortran 2008 binding 32 33 MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm, 34 ierror) TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf 35TYPE(*), DIMENSION(..) :: recvbuf 36 INTEGER, INTENT(IN) :: recvcount 37 TYPE(MPI_Datatype), INTENT(IN) :: datatype 38 TYPE(MPI_Op), INTENT(IN) :: op 39 TYPE(MPI_Comm), INTENT(IN) :: comm 40 41 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 42Fortran binding 43 MPI_REDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM, 44 IERROR) 45<type> SENDBUF(*), RECVBUF(*) 46INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR 47

1

2

5 6

9

10

11 12

13 14

1 If comm is an intracommunicator, MPI_REDUCE_SCATTER_BLOCK first performs a $\mathbf{2}$ global, element-wise reduction on vectors of $count = n^{*}recvcount$ elements in the send buffers 3 defined by sendbuf, count and datatype, using the operation op, where n is the number of 4 processes in the group of comm. The routine is called by all group members using the 5same arguments for recvcount, datatype, op and comm. The resulting vector is treated 6 as n consecutive blocks of recvcount elements that are scattered to the processes of the $\overline{7}$ group. The i-th block is sent to process i and stored in the receive buffer defined by recvbuf, 8 recvcount, and datatype.

Advice to implementors. The MPI_REDUCE_SCATTER_BLOCK routine is functionally equivalent to: an MPI_REDUCE collective operation with count equal to recvcount*n, followed by an MPI_SCATTER with sendcount equal to recvcount. However, a direct implementation may run faster. (*End of advice to implementors.*)

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in the **sendbuf** argument on *all* processes. In this case, the input data is taken from the receive buffer.

If comm is an intercommunicator, then the result of the reduction of the data provided by processes in one group (group A) is scattered among processes in the other group (group B) and vice versa. Within each group, all processes provide the same value for the recvcount argument, and provide input vectors of count = $n^{recvcount}$ elements stored in the send buffers, where n is the size of the group. The number of elements count must be the same for the two groups. The resulting vector from the other group is scattered in blocks of recvcount elements among the processes in the group.

Rationale. The last restriction is needed so that the length of the send buffer of one group can be determined by the local recvcount argument of the other group. Otherwise, a communication is needed to figure out how many elements are reduced. (End of rationale.)

29 30 31

32

33

34

25

26

27

28

9

10

11

12

13 14

5.10.2 MPI_REDUCE_SCATTER

MPI_REDUCE_SCATTER extends the functionality of MPI_REDUCE_SCATTER_BLOCK such that the scattered blocks can vary in size. Block sizes are determined by the recvcounts array, such that the i-th block contains recvcounts[i] elements.

MPI_RED	UCE_SCATTER(sendbuf, recvl	ouf, recvcounts, datatype, op, comm)	1
IN	sendbuf	starting address of send buffer (choice)	2
ουτ	recybuf	starting address of receive buffer (choice)	3
IN	recycounts	č (,	4 5
IIN	recvcounts	non-negative integer array (of length group size) specifying the number of elements of the result	6
		distributed to each process.	7
IN	datatype	data type of elements of send and receive buffers	8
	51	(handle)	9 10
IN	ор	operation (handle)	10
IN	comm	communicator (handle)	12
			13

MPL REDUCE SCATTER(sendbuf recybuf recycounts datatype on comm)

C binding

int	MPI_Reduce_scatter(const void *sendbuf, void *recvbuf,	
	<pre>const int recvcounts[], MPI_Datatype datatype, MPI_Op op,</pre>	
	MPI_Comm comm)	

Fortran 2008 binding

MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
ierror)
TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION() :: recvbuf
<pre>INTEGER, INTENT(IN) :: recvcounts(*)</pre>
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Op), INTENT(IN) :: op
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

```
MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
             IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR
```

If comm is an intracommunicator, MPI_REDUCE_SCATTER first performs a global, element-wise reduction on vectors of $count = \sum_{i=0}^{n-1} recvcounts[i]$ elements in the send buffers defined by sendbuf, count and datatype, using the operation op, where n is the number of processes in the group of comm. The routine is called by all group members using the same arguments for recvcounts, datatype, op and comm. The resulting vector is treated as n consecutive blocks where the number of elements of the i-th block is recvcounts[i]. The blocks are scattered to the processes of the group. The i-th block is sent to process i and stored in the receive buffer defined by recvbuf, recvcounts[i] and datatype.

Advice to implementors. The MPI_REDUCE_SCATTER routine is functionally equivalent to: an MPI_REDUCE collective operation with count equal to the sum of recvcounts[i] followed by MPI_SCATTERV with sendcounts equal to recvcounts. However, a direct implementation may run faster. (End of advice to implementors.)

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in the sendbuf argument. In this case, the input data is taken from the receive buffer. It is not required to specify the "in place" option on all processes, since the processes for which recvcounts[i] == 0 may not have allocated a receive buffer.

⁵ If comm is an intercommunicator, then the result of the reduction of the data provided ⁶ by processes in one group (group A) is scattered among processes in the other group (group ⁷ B), and vice versa. Within each group, all processes provide the same recvcounts argument, ⁸ and provide input vectors of count = $\sum_{i=0}^{n-1} \text{recvcounts}[i]$ elements stored in the send buffers, ⁹ where n is the size of the group. The resulting vector from the other group is scattered in ¹⁰ blocks of recvcounts[i] elements among the processes in the group. The number of elements ¹¹ count must be the same for the two groups.

Rationale. The last restriction is needed so that the length of the send buffer can be determined by the sum of the local **recvcounts** entries. Otherwise, a communication is needed to figure out how many elements are reduced. (*End of rationale.*)

5.11 Scan

5.11.1 Inclusive Scan

MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm)

24	IN	sendbuf	starting address of send buffer (choice)				
25	OUT	recvbuf	starting address of receive buffer (choice)				
26 27 28	IN	count	number of elements in input buffer (non-negative integer)				
29	IN	datatype	data type of elements of input buffer (handle)				
30	IN	ор	operation (handle)				
31 32	IN	comm	communicator (handle)				
33 34	C binding						
35	int MPI_S		void *recvbuf, int count,				
36		MPI_Datatype datatyp	e, MPI_Op op, MPI_Comm comm)				
37	Fortran 2	2008 binding					
38	MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror)						
39	TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf						
40	TYPE(*), DIMENSION() :: recvbuf						
41	INTEGER, INTENT(IN) :: count						
42	TYPE(MPI_Datatype), INTENT(IN) :: datatype						
43	TYPE(MPI_Op), INTENT(IN) :: o	p				
44	TYPE(MPI_Comm), INTENT(IN) ::	comm				
45	INTEG	ER, OPTIONAL, INTENT(OUT)) :: ierror				
46	The stars 1	· 1·					
47	Fortran b	0					
48	MPI_SCAN(SENDBUF, RECVBUF, COUNT,	DATATYPE, OP, COMM, IERROR)				

1

 $\mathbf{2}$

3

4

12

13

14

15 16 17

18 19

20 21 22

<type> SENDBUF(*), RECVBUF(*)</type>				
INTEGER COUNT, DATATYPE, OP, COMM, IERROR				
If c	If comm is an intracommunicator, MPI_SCAN is used to perform a prefix reduction on			
		coup. The operation returns, in the receive buffer of the process	4 5	
with rai	nk i, the reduction of	f the values in the send buffers of processes with ranks $0,\ldots,i$	6	
		lled by all group members using the same arguments for count,	7	
		cept that for user-defined operations, the same rules apply as	8	
		of operations supported, their semantics, and the constraints	9	
		are as for MPI_REDUCE. or intracommunicators is specified by passing MPI_IN_PLACE in	10	
		his case, the input data is taken from the receive buffer, and	11	
	by the output data.		12	
-	° *	l for intercommunicators.	13 14	
	*		14	
5.11.2	Exclusive Scan		16	
			17	
			18	
MPI_EX	SCAN(sendbuf, recvb	uf, count, datatype, op, comm)	19	
IN	sendbuf	starting address of send buffer (choice)	20	
OUT	recvbuf	starting address of receive buffer (choice)	21 22	
IN	count	number of elements in input buffer (non-negative	23	
		integer)	24	
IN	datatype	data type of elements of input buffer (handle)	25	
IN	ор	operation (handle)	26	
		intracommunicator (handle)	27	
IN	comm	intracommunicator (nandie)	28 29	
C bind	ing		30	
	0	d *sendbuf, void *recvbuf, int count,	31	
		pe datatype, MPI_Op op, MPI_Comm comm)	32	
D 4			33	
	1 2008 binding	white count detetions on commission	34	
		buf, count, datatype, op, comm, ierror) .), INTENT(IN) :: sendbuf	35	
	PE(*), DIMENSION(.		36 27	
	TEGER, INTENT(IN)		37 38	
		INTENT(IN) :: datatype	39	
TYI	PE(MPI_Op), INTENT	(IN) :: op	40	
	PE(MPI_Comm), INTE		41	
INT	TEGER, OPTIONAL, I	NTENT(OUT) :: ierror	42	
Fortrai	n binding		43	
		BUF, COUNT, DATATYPE, OP, COMM, IERROR)	44	
•	<pre>vpe> SENDBUF(*), F</pre>		45	
INT	TEGER COUNT, DATAT	YPE, OP, COMM, IERROR	$\frac{46}{47}$	
			48	

1 If comm is an intracommunicator, MPI_EXSCAN is used to perform a prefix reduction $\mathbf{2}$ on data distributed across the group. The value in recvbuf on the process with rank 0 is 3 undefined, and recvbuf is not significant on process 0. The value in recvbuf on the process 4 with rank 1 is defined as the value in sendbuf on the process with rank 0. For processes $\mathbf{5}$ with rank i > 1, the operation returns, in the receive buffer of the process with rank i, the 6 reduction of the values in the send buffers of processes with ranks $0, \ldots, i-1$ (inclusive). The $\overline{7}$ routine is called by all group members using the same arguments for count, datatype, op and 8 comm, except that for user-defined operations, the same rules apply as for MPI_REDUCE. 9 The type of operations supported, their semantics, and the constraints on send and receive 10 buffers, are as for MPI_REDUCE.

¹¹ The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in ¹² the sendbuf argument. In this case, the input data is taken from the receive buffer, and ¹³ replaced by the output data. The receive buffer on rank 0 is not changed by this operation. ¹⁴ This operation is invalid for intercommunicators.

Rationale. The exclusive scan is more general than the inclusive scan. Any inclusive scan operation can be achieved by using the exclusive scan and then locally combining the local contribution. Note that for non-invertable operations such as MPI_MAX, the exclusive scan cannot be computed with the inclusive scan. (*End of rationale.*)

5.11.3 Example using MPI_SCAN

²³ The example in this section uses an intracommunicator.

Example 5.23 This example uses a user-defined operation to produce a *segmented scan*.A segmented scan takes, as input, a set of values and a set of logicals, and the logicals delineate the various segments of the scan. For example:

values	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
logicals	0	0	1	1)	1	0	0	1
result	v_1	$v_1 + v_2$	v_3	$v_3 + v_4$	$v_3 + v_4 + v_5$	v_6	$v_6 + v_7$	v_8

The operator that produces this effect is

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

where

Note that this is a non-commutative operator. C code that implements it is given below.

 $w = \begin{cases} u+v & \text{if } i=j\\ v & \text{if } i\neq j \end{cases}.$

43 44

- 45
- 46
- 47 48

15

16

17

18 19

20 21

22

 24

25

26

37

38 39 40

41

```
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
 */
MPI_Get_address(&a, disp);
                                                                                   38
                                                                                   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

 $\mathbf{2}$

3

45

6

9

10 11

12 13 14

15

16

17

18

19

20 21

22

23 24

25

26

1 2

7

5.12Nonblocking 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 [31, 35]. One way of doing this would be to perform a blocking collective operation in a separate thread. An alternative 8 mechanism that often leads to better performance (e.g., avoids context switching, scheduler 9 overheads, and thread management) is to use nonblocking collective communication [33]. 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 5.13 and 7.8.

Rationale. Freeing an active nonblocking collective request could cause similar problems as discussed for point-to-point requests (see Section 3.7.3). Cancelling a request is not supported because the semantics of this operation are not well-defined. (End of rationale.)

Multiple nonblocking collective operations can be outstanding on a single communicator. If the nonblocking call causes some system resource to be exhausted, then it will fail and generate an MPI exception. Quality implementations of MPI should ensure that this happens only in pathological cases. That is, an MPI implementation should be able to support a large number of pending nonblocking operations.

Unlike point-to-point operations, nonblocking collective operations do not match with blocking collective operations, and collective operations do not have a tag argument. All processes must call collective operations (blocking and nonblocking) in the same order per communicator. In particular, once a process calls a collective operation, all other processes in the communicator must eventually call the same collective operation, and no other collective operation with the same communicator in between. This is consistent with 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 intracommunicators and intercommunicators after completion. Likewise, upon completion, nonblocking collective reduction operations have the same effect as their blocking counterparts, and the same restrictions and recommendations on reduction orders apply.

The use of the "in place" option is allowed exactly as described for the corresponding blocking collective operations. When using the "in place" option, message buffers function as both send and receive buffers. Such buffers should not be modified or accessed until the operation completes.

Progression rules for nonblocking collective operations are similar to progression of nonblocking point-to-point operations, refer to Section 3.7.4.

46Advice to implementors. Nonblocking collective operations can be implemented with 47local execution schedules [34] 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

$\frac{1}{2}$	5.12.1 Nonblocking Barrier Synchronization							
3								
4 5	MPI_IBARRIER(comm, request)							
6	IN	comm	communicator (handle)					
7 8	OUT	request	communication request (handle)					
9 10 11	C binding int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)							
12 13 14 15 16	MPI_Ibarr TYPE(TYPE(<pre>Fortran 2008 binding MPI_Ibarrier(comm, request, ierror) TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>						
17 18 19 20	Fortran binding MPI_IBARRIER(COMM, REQUEST, IERROR) INTEGER COMM, REQUEST, IERROR							
20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37	 MPI_IBARRIER is a nonblocking version of MPI_BARRIER. By calling MPI_IBARRIER, a process notifies that it has reached the barrier. The call returns immediately, independent of whether other processes have called MPI_IBARRIER. The usual barrier semantics are enforced at the corresponding completion operation (test or wait), which in the intracommunicator case will complete only after all other processes in the communicator have called MPI_IBARRIER. In the intercommunicator case, it will complete when all processes in the remote group have called MPI_IBARRIER. Advice to users. A nonblocking barrier can be used to hide latency. Moving independent computations between the MPI_IBARRIER and the subsequent completion call can overlap the barrier latency and therefore shorten possible waiting times. The semantic properties are also useful when mixing collective operations and point-to-point messages. (End of advice to users.) 5.12.2 Nonblocking Broadcast 							
38	MPI_IBCA	ST(buffer, count, dat	atype, root, comm, request)					
39	INOUT	buffer	starting address of buffer (choice)					
40 41	IN	count	number of entries in buffer (non-negative integer)					
42	IN	datatype	data type of buffer (handle)					
43	IN	root	rank of broadcast root (integer)					
44	IN	comm	communicator (handle)					
45 46	OUT	request	communication request (handle)					
47		~- 1						
48	C binding							

```
1
int MPI_Ibcast(void *buffer, int count, MPI_Datatype datatype, int root,
                                                                                        2
              MPI_Comm comm, MPI_Request *request)
                                                                                        3
Fortran 2008 binding
MPI_Ibcast(buffer, count, datatype, root, comm, request, ierror)
                                                                                        5
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
                                                                                        6
    INTEGER, INTENT(IN) :: count, root
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                        9
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                        10
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                        11
Fortran binding
                                                                                        12
                                                                                        13
MPI_IBCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR)
                                                                                        14
    <type> BUFFER(*)
                                                                                        15
    INTEGER COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR
                                                                                        16
    This call starts a nonblocking variant of MPI_BCAST (see Section 5.4).
                                                                                        17
                                                                                        18
Example using MPI_IBCAST
                                                                                        19
                                                                                        20
The example in this section uses an intracommunicator.
                                                                                        21
Example 5.24 Start a broadcast of 100 ints from process 0 to every process in the
                                                                                        22
                                                                                        23
group, perform some computation on independent data, and then complete the outstanding
                                                                                        ^{24}
broadcast operation.
                                                                                        25
                                                                                        26
    MPI_Comm comm;
    int array1[100], array2[100];
                                                                                        27
    int root=0;
                                                                                        28
    MPI_Request req;
                                                                                        29
                                                                                        30
    . . .
    MPI_Ibcast(array1, 100, MPI_INT, root, comm, &req);
                                                                                        31
    compute(array2, 100);
                                                                                        32
    MPI_Wait(&req, MPI_STATUS_IGNORE);
                                                                                        33
                                                                                        34
                                                                                        35
                                                                                        36
                                                                                        37
                                                                                        38
                                                                                        39
                                                                                        40
                                                                                        41
                                                                                        42
                                                                                        43
                                                                                        44
                                                                                        45
                                                                                        46
                                                                                        47
                                                                                        48
```

1 2	5.12.3 N	Ionblocking Gather						
3 4 5	MPI_IGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, request)							
6 7	IN sendbuf starting address of send buffer (choice)							
8 9	IN	sendcount	number of elements in send buffer (non-negative integer)					
10	IN	sendtype	data type of send buffer elements (handle)					
11 12 13	OUT	recvbuf	address of receive buffer (choice, significant only at root)					
$\frac{14}{15}$	IN	recvcount	number of elements for any single receive (non-negative integer, significant only at root)					
16 17	IN	recvtype	data type of recv buffer elements (handle, significant only at root)					
18 19	IN	root	rank of receiving process (integer)					
20	IN	comm	communicator (handle)					
21 22	OUT	request	communication request (handle)					
24 25 26 27	int MPI_Igather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI Comm comm. MPI Request *request)							
28 29	Fortran 2008 binding							
29 30	<pre>MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, request, ierror)</pre>							
31	TYPE		NTENT(IN), ASYNCHRONOUS :: sendbuf					
32	INTEGER, INTENT(IN) :: sendcount, recvcount, root							
33 34	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype							
35	TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf 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_IGAT		T, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,					
41	ROOT, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*)</type>							
42	01	-	PE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,					
43 44		IERROR	_,,,,,,,,,					
45	This	call starts a nonblocking v	variant of MPI_GATHER (see Section 5.5).					
46	This can starts a nonsidering variant of twi 1_OATHER (see Section 5.5).							
47								
48								

MPI_IGATI	HERV(sendbuf, sendcount, send comm, request)	dtype, recvbuf, recvcounts, displs, recvtype, root,	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	data type 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	data type of recv buffer elements (handle, significant only at root)	19 20			
IN	root	rank of receiving process (integer)	21			
IN	comm	communicator (handle)	22			
OUT	request	communication request (handle)	23 24			
C binding int MPI_Igatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)						
	008 binding	sendtype, recvbuf, recvcounts, displs,	30 31 32			
MP1_1gath	recvtype, root, comm		33			
TYPE(C(IN), ASYNCHRONOUS :: sendbuf	34			
	ER, INTENT(IN) :: sendcou	-	35			
	MPI_Datatype), INTENT(IN)		36 37			
	*), DIMENSION(), ASYNCH		38			
INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*) TYPE(MPI_Comm), INTENT(IN) :: comm						
	MPI_Request), INTENT(OUT)		40			
	ER, OPTIONAL, INTENT(OUT)	-	41			
Fortran b	inding		42			
	0	SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,	43 44			
	RECVTYPE, ROOT, COMM		44 45			
<type< td=""><td>> SENDBUF(*), RECVBUF(*)</td><td></td><td>46</td></type<>	> SENDBUF(*), RECVBUF(*)		46			
INTEG	INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT, COMM, REQUEST, IERROR					

Unofficial Draft for Comment Only

```
CHAPTER 5. COLLECTIVE COMMUNICATION
1
         This call starts a nonblocking variant of MPI_GATHERV (see Section 5.5).
\mathbf{2}
3
     5.12.4 Nonblocking Scatter
4
5
6
     MPI_ISCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm,
7
                     request)
8
       IN
                 sendbuf
                                             address of send buffer (choice, significant only at
9
                                             root)
10
11
       IN
                 sendcount
                                             number of elements sent to each process
12
                                             (non-negative integer, significant only at root)
13
                                             data type of send buffer elements (handle, significant
       IN
                 sendtype
14
                                             only at root)
15
       OUT
                                             address of receive buffer (choice)
                 recvbuf
16
17
       IN
                                             number of elements in receive buffer (non-negative
                 recvcount
18
                                             integer) •
19
                                             data type of receive buffer elements (handle)
       IN
                 recvtype
20
                                             rank of sending process (integer)
       IN
                 root
21
22
       IN
                                             communicator (handle)
                 comm
23
       OUT
                 request
                                             communication request (handle)
24
25
     C binding
26
     int MPI_Iscatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
27
                    void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
28
                    MPI_Comm comm, MPI_Request *request)
29
30
     Fortran 2008 binding
^{31}
     MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
32
                    root, comm, request, ierror)
33
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
34
         INTEGER, INTENT(IN) :: sendcount, recvcount, root
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
     Fortran binding
41
     MPI_ISCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
42
                    ROOT, COMM, REQUEST, IERROR)
43
          <type> SENDBUF(*), RECVBUF(*)
44
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,
45
                     IERROR
46
47
         This call starts a nonblocking variant of MPI_SCATTER (see Section 5.6).
48
```

MPI_ISCAT	TERV(sendbuf, sendcounts, dis comm, request)	spls, 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)	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	data type 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	data type of receive buffer elements (handle)	19		
IN	root	rank of sending process (integer)	20 21		
IN	comm	communicator (handle)	22		
OUT	request	communication request (handle)	23		
	·		24 25		
C binding	C binding				
int MPI_Is		<pre>buf, const int sendcounts[],</pre>	27		
		PI_Datatype sendtype, void *recvbuf,	28		
	<pre>Int recvcount, MPI_Da MPI_Request *request)</pre>	tatype recvtype, int root, MPI_Comm comm,	29		
_			30		
	008 binding		31 32		
MP1_1scatt	recvtype, root, comm,	displs, sendtype, recvbuf, recvcount,	33		
TYPE(*		(IN), ASYNCHRONOUS :: sendbuf	34		
	ER, INTENT(IN), ASYNCHRON		35		
	ER, INTENT(IN) :: displs(36		
	<pre>MPI_Datatype), INTENT(IN)</pre>	VI VI	37 38		
	<pre>k), DIMENSION(), ASYNCH 4PI_Comm), INTENT(IN) ::</pre>		39		
	<pre>/PI_COMM/, INTENT(IN/ /PI_Request), INTENT(OUT)</pre>		40		
	ER, OPTIONAL, INTENT(OUT)	-	41		
Fortran bi	inding		42		
		DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,	43 44		
	RECVTYPE, ROOT, COMM,		44 45		
• 1	> SENDBUF(*), RECVBUF(*)		46		
INTEGE		*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,	47		
	COMM, REQUEST, IERRO	n.	48		

	216	Cl	HAPTER 5. COLLECTIVE COMMUNICATION
1 2	This c	all starts a nonblocking varia	nt of MPI_SCATTERV (see Section 5.6).
3 4 5	5.12.5 N	onblocking Gather-to-all	
6 7	MPI_IALLO	GATHER(sendbuf, sendcount, s request)	endtype, recvbuf, recvcount, recvtype, comm,
8 9	IN	sendbuf	starting address of send buffer (choice)
10 11	IN	sendcount	number of elements in send buffer (non-negative integer)
12 13	IN	sendtype	data type of send buffer elements (handle)
13	OUT	recvbuf	address of receive buffer (choice)
15 16	IN	recvcount	number of elements received from any process (non-negative integer)
17 18	IN	recvtype	data type of receive buffer elements (handle)
19	IN	comm	communicator (handle)
20 21	OUT	request	communication request (handle)
23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 41 42 43 44 45 46 47	Fortran 2 MPI_Iallg TYPE(INTEG TYPE(TYPE(TYPE(TYPE(INTEG Fortran b MPI_IALLG <type INTEG</type 	<pre>MPI_Datatype recvtyp 2008 binding gather(sendbuf, sendcount</pre>	<pre>e, void *recvbuf, int recvcount, e, MPI_Comm comm, MPI_Request *request) , sendtype, recvbuf, recvcount, recvtype, r) F(IN), ASYNCHRONOUS :: sendbuf unt, recvcount) :: sendtype, recvtype HRONOUS :: recvbuf comm) :: request) :: ierror , SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,</pre>

MPI_IAI	LGATHERV(sendbuf, sen comm, request)	dcount, sendtype, recvbuf, recvcounts, displs, recvtype,	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	data type of send buffer elements (handle)	7
OUT	recvbuf	address of receive buffer (choice)	8
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	data type of receive buffer elements (handle)	16 17
IN	comm	communicator (handle)	18
OUT	request	communication request (handle)	19
MPI_Ial	recvtype, comm	ndcount, sendtype, recvbuf, recvcounts, displs, , request, ierror) INTENT(IN), ASYNCHRONOUS :: sendbuf	26 27 28 29 30
	EGER, INTENT(IN) :: :		30 31
		NT(IN) :: sendtype, recvtype	32
		ASYNCHRONOUS :: recvbuf	33
		<pre>NCHRONOUS :: recvcounts(*), displs(*)</pre>	34
	PE(MPI_Comm), INTENT(] PE(MPI_Request), INTEN		35 36
	EGER, OPTIONAL, INTER	-	37
Fortrar	n binding		38
	LGATHERV (SENDBUF, SEN	IDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, , REQUEST, IERROR)	39 40
·	<pre>vpe> SENDBUF(*), RECVE</pre>		41 42
INT	EGER SENDCOUNT, SEND REQUEST, IERRO	<pre>YPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,</pre>	43
			44
Thi	s call starts a nonblockin	g variant of MPI_ALLGATHERV (see Section 5.7).	45
			46 47

218		CHAPTER 5. COLLECTIVE COMMUNICATION
¹ 5.12.	.6 Nonblocking All-to-All Sca	atter/Gather
	IALLTOALL(sendbuf, sendcoun	t, sendtype, recvbuf, recvcount, recvtype, comm, request)
6 7 IN	sendbuf	starting address of send buffer (choice)
8 IN 9	sendcount	number of elements sent to each process (non-negative integer)
10 IN	sendtype	data type of send buffer elements (handle)
11 12 OU	IT recvbuf	address of receive buffer (choice)
13 IN 14	recvcount	number of elements received from any process (non-negative integer)
15 16	recvtype	data type of receive buffer elements (handle)
10 17 IN	comm	communicator (handle)
18 OU	IT request	communication request (handle)
21 int 22 23 23 Fort 25 Fort 26 MPI 27 28 29 30 31 32 33 34 35 Fort 36 MPI 37 38 39 40	MPI_Datatype rec ran 2008 binding Ialltoall(sendbuf, sendcou comm, request, i TYPE(*), DIMENSION(), II INTEGER, INTENT(IN) :: sen TYPE(MPI_Datatype), INTENT TYPE(*), DIMENSION(), AS TYPE(MPI_Comm), INTENT(IN) TYPE(MPI_Request), INTENT INTEGER, OPTIONAL, INTENT INTEGER, OPTIONAL, INTENT ran binding IALLTOALL(SENDBUF, SENDCOU COMM, REQUEST, I <type> SENDBUF(*), RECVBUI INTEGER SENDCOUNT, SENDTYN</type>	<pre>dtype, void *recvbuf, int recvcount, vtype, MPI_Comm comm, MPI_Request *request) unt, sendtype, recvbuf, recvcount, recvtype, error) NTENT(IN), ASYNCHRONOUS :: sendbuf ndcount, recvcount T(IN) :: sendtype, recvtype SYNCHRONOUS :: recvbuf) :: comm (OUT) :: request (OUT) :: ierror JNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ERROR)</pre>

- 45 46
- 47 48

MPI_IAL	LTOALLV(sendbuf, sen recvtype, comm	dcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, , request)	1 2
IN	sendbuf	starting address of send buffer (choice)	3
IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank	4 5 6 7
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	8 9 10
IN	sendtype	data type of send buffer elements (handle)	11
OUT	recvbuf	address of receive buffer (choice)	12 13
IN	recvcounts	non-negative integer array (of length group size) specifying the number of elements that can be received from each rank	14 15 16
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	17 18 19 20
IN	recvtype	data type of receive buffer elements (handle)	21
IN	comm	communicator (handle)	22
OUT	request	communication request (handle)	23 24 25
C bindi int MPI	_Ialltoallv(const v const int sdi const int rec	<pre>oid *sendbuf, const int sendcounts[], ispls[], MPI_Datatype sendtype, void *recvbuf, cvcounts[], const int rdispls[], recvtype, MPI_Comm comm, MPI_Request *request)</pre>	26 27 28 29
MPI_Ial TYP	2008 binding ltoallv(sendbuf, se rdispls, recv E(*), DIMENSION() EGER, INTENT(IN), A	<pre>ndcounts, sdispls, sendtype, recvbuf, recvcounts, vtype, comm, request, ierror) , INTENT(IN), ASYNCHRONOUS :: sendbuf SYNCHRONOUS :: sendcounts(*), sdispls(*),), rdispls(*)</pre>	30 31 32 33 34 35 36
TYP: TYP: TYP:	E(*), DIMENSION() E(MPI_Comm), INTENT	ENT(OUT) :: request	37 38 39 40 41
Fortran	binding		42 43
MPI_IAL	LTOALLV(SENDBUF, SE RDISPLS, RECV pe> SENDBUF(*), REC	NDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, /TYPE, COMM, REQUEST, IERROR) /VBUF(*) SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),	43 44 45 46 47
		MM, REQUEST, IERROR	48

	220	CH	IAPTER 5. COLLECTIVE COMMUNICATION
1 2 3	This ca	all starts a nonblocking variar	at of MPI_ALLTOALLV (see Section 5.8).
3 4 5	MPI_IALLT	OALLW(sendbuf, sendcounts, s recvtypes, comm, request	sdispls, sendtypes, recvbuf, recvcounts, rdispls,)
6	IN	sendbuf	starting address of send buffer (choice)
7 8 9 10	IN	sendcounts	integer array (of length group size) specifying the number of elements to send to each rank (array of non-negative integers)
11 12 13 14	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)
15 16 17	IN	sendtypes	array of datatypes (of length group size). Entry j specifies the type of data to send to process j (array of handles)
18 19	OUT	recvbuf	address of receive buffer (choice)
20 21 22	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)
23 24 25 26	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)
27 28 29 30	IN	recvtypes	array of datatypes (of length group size). Entry i specifies the type of data received from process i (array of handles)
31	IN	comm	communicator (handle)
32 33	Ουτ	request	communication request (handle)
34	C binding	S	
35 36	int MPI_I		dbuf, const int sendcounts[],
37		-	<pre>const MPI_Datatype sendtypes[], int requestration additional</pre>
38			<pre>int recvcounts[], const int rdispls[], ecvtypes[], MPI_Comm comm,</pre>
39		MPI_Request *request)	
40	Fortron 9	008 binding	
41 42		0	, sdispls, sendtypes, recvbuf,
43			recvtypes, comm, request, ierror)
44		*), DIMENSION(), INTENT	(IN), ASYNCHRONOUS :: sendbuf
45	INTEG		OUS :: sendcounts(*), sdispls(*),
46	TVDE (1	recvcounts(*), rdisp	
47 48	IIFE()	recvtypes(*)	, ASYNCHRONOUS :: sendtypes(*),
		JF ()	

```
TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                         1
                                                                                         2
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
MPI_IALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
               RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
                                                                                         10
               RDISPLS(*), RECVTYPES(*), COMM, REQUEST, IERROR
                                                                                         11
    This call starts a nonblocking variant of MPI_ALLTOALLW (see Section 5.8).
                                                                                         12
                                                                                         13
                                                                                         14
5.12.7 Nonblocking Reduce
                                                                                         15
                                                                                         16
                                                                                         17
MPI_IREDUCE(sendbuf, recvbuf, count, datatype, op, root, comm, request)
                                                                                         18
  IN
           sendbuf
                                       address of send buffer (choice)
                                                                                         19
  OUT
                                                                                         20
           recvbuf
                                       address of receive buffer (choice, significant only at
                                                                                         21
                                       root)
                                                                                         22
  IN
           count
                                       number of elements in send buffer (non-negative
                                                                                         23
                                       integer)
                                                                                         ^{24}
  IN
                                       data type of elements of send buffer (handle)
           datatype
                                                                                         25
                                                                                         26
  IN
                                       reduce operation (handle)
           op
                                                                                         27
  IN
                                       rank of root process (integer)
           root
                                                                                         28
  IN
                                       communicator (handle)
           comm
                                                                                         29
                                                                                         30
  OUT
           request
                                       communication request (handle)
                                                                                         31
                                                                                         32
C binding
                                                                                         33
int MPI_Ireduce(const void *sendbuf, void *recvbuf, int count,
                                                                                         34
               MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
                                                                                         35
               MPI_Request *request)
                                                                                         36
Fortran 2008 binding
                                                                                         37
MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request,
                                                                                         38
               ierror)
                                                                                         39
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                         40
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                         41
    INTEGER, INTENT(IN) :: count, root
                                                                                         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
                                                                                         48
```

1	Fortran b	inding			
2	MPI_IREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST,				
3	IERROR)				
4	• 1	> SENDBUF(*), RECVBUF(*)			
5	INTEG	INTEGER COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, IERROR			
6 7	This c	all starts a nonblocking varia	ant of MPI_REDUCE (see Section $5.9.1$).		
8	4.7.				
9		-	nplementation is explicitly allowed to use different ocking reduction operations that might change the		
10	0	0	ons. However, as for MPI_REDUCE, it is strongly		
11		-	be implemented so that the same result be obtained		
12			n the same arguments, appearing in the same order.		
$\frac{13}{14}$		that this may prevent optimi occesses. (<i>End of advice to im</i>	zations that take advantage of the physical location		
15	or pro		prementors.)		
16	Advie	ce to users. For operations w	which are not truly associative, the result delivered		
17 18	*	1	ng reduction may not exactly equal the result deliv-		
19			en when specifying the same arguments in the same		
20	order	. (End of advice to users.)			
21	E 10.0 N	anhlading All Daduca			
22	5.12.8 No	onblocking All-Reduce			
23					
24 25	MPI_IALLF	REDUCE(sendbuf, recvbuf, cou	int, datatype, op, comm, request)		
26	IN	sendbuf	starting address of send buffer (choice)		
27 28	OUT	recvbuf	starting address of receive buffer (choice)		
29 30	IN	count	number of elements in send buffer (non-negative integer)		
31	IN	datatype	data type of elements of send buffer (handle)		
32 33	IN	ор	operation (handle)		
34	IN	comm	communicator (handle)		
35	OUT	request	communication request (handle)		
36					
37	C binding	-			
38 39	int MPI_I		ndbuf, void *recvbuf, int count,		
40			pe, MPI_Op op, MPI_Comm comm,		
41		MPI_Request *request	:)		
42	Fortran 2	008 binding			
43	MPI_Iallr		count, datatype, op, comm, request,		
44		ierror)	- /		
45			T(IN), ASYNCHRONOUS :: sendbuf		
46		*), DIMENSION(), ASYNC ER, INTENT(IN) :: count	HKUNUUS :: TECVDUI		
47 48		MPI_Datatype), INTENT(IN) :: datatype		
40	11110	(IN	, aasasyps		

	PE(MPI_Op), INTENT(-		
	PE(MPI_Comm), INTENT			
	PE(MPI_Request), IN	1		
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
	an binding			
MP1_1	IERROR)	ECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, 7		
<1	type> SENDBUF(*), REG	s CVBUF(*)		
	• •	PE, OP, COMM, REQUEST, IERROR		
T	his call starts a nonblock	king variant of MPI_ALLREDUCE (see Section 5.9.6). 12		
5.12.9	Nonblocking Reduce-	Scatter with Equal Blocks		
	0	14		
MPI_IF	REDUCE_SCATTER_BL(request)	<pre>OCK(sendbuf, recvbuf, recvcount, datatype, op, comm, 1'</pre>		
IN	sendbuf	starting address of send buffer (choice)		
OUT	recvbuf	starting address of receive buffer (choice)		
IN	recvcount	element count per block (non-negative integer)		
IN	datatype	data type of elements of send and receive buffers 22		
		(handle) 24		
IN	ор	operation (handle) 22		
IN	comm	communicator (handle)		
OUT	request	communication request (handle) 22		
		23		
C bin	ding	30		
int MH		lock(const void *sendbuf, void *recvbuf, 33		
		t, MPI_Datatype datatype, MPI_Op op, 33		
	MPI_Comm com	m, MPI_Request *request) 33		
Fortra	an 2008 binding	3		
MPI_I	reduce_scatter_block	(sendbuf, recvbuf, recvcount, datatype, op, comm, $\frac{3}{36}$		
	request, ier	ror) _{3'}		
), INTENT(IN), ASYNCHRONOUS :: sendbuf		
), ASYNCHRONOUS :: recvbuf		
	NTEGER, INTENT(IN) :	40		
	• -	NTENT(IN) :: datatype 4		
	<pre>/PE(MPI_Op), INTENT(] /PE(MPI_Comm), INTEN]</pre>	▲ 44		
	YPE(MPI_Comm), INTEN YPE(MPI_Request), INT	44		
	VTEGER, OPTIONAL, IN	TENT(OUT) :: jerror		
		41		
	an binding			
WFT_TF		(SENDBOF, RECODONI, DATATIFE, UF, COMM,		
	REQUEST, IER	1.011/		

```
224
                                         CHAPTER 5. COLLECTIVE COMMUNICATION
1
          <type> SENDBUF(*), RECVBUF(*)
\mathbf{2}
          INTEGER RECVCOUNT, DATATYPE, OP, COMM, REQUEST, IERROR
3
         This call starts a nonblocking variant of MPI_REDUCE_SCATTER_BLOCK (see Sec-
4
     tion 5.10.1).
5
6
     5.12.10 Nonblocking Reduce-Scatter
7
8
9
     MPI_IREDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm, request)
10
11
       IN
                 sendbuf
                                             starting address of send buffer (choice)
12
       OUT
                 recvbuf
                                             starting address of receive buffer (choice)
13
       IN
14
                 recvcounts
                                             non-negative integer array specifying the number of
15
                                             elements in result distributed to each process. This
16
                                             array must be identical on all calling processes.
17
       IN
                 datatype
                                             data type of elements of input buffer (handle)
18
       IN
                                             operation (handle)
                 ор
19
20
       IN
                                             communicator (handle)
                 comm
21
       OUT
                                             communication request (handle)
                 request
22
23
     C binding
24
     int MPI_Ireduce_scatter(const void *sendbuf, void *recvbuf,
25
                    const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
26
                    MPI_Comm comm, MPI_Request *request)
27
     Fortran 2008 binding
28
     MPI_Ireduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
29
30
                    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
          TYPE(MPI_Comm), INTENT(IN) :: comm
36
37
          TYPE(MPI_Request), INTENT(OUT) :: request
38
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
     Fortran binding
40
     MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
41
                    REQUEST, IERROR)
42
          <type> SENDBUF(*), RECVBUF(*)
43
          INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, REQUEST, IERROR
44
45
         This call starts a nonblocking variant of MPI_REDUCE_SCATTER (see Section 5.10.2).
46
47
48
```

5.12.11 Nonblocking Inclusive Scan

J.12.11	Nonbiocking inclusive Scall		2
			3
MPI_ISC	AN(sendbuf, recvbuf, count, data	ntype, op, comm, request)	4
IN	sendbuf	starting address of send buffer (choice)	5
OUT	recvbuf	÷ , ,	6 7
		starting address of receive buffer (choice)	8
IN	count	number of elements in input buffer (non-negative integer)	9
IN	datatype	data type of elements of input buffer (handle)	10 11
IN	ор	operation (handle)	12
IN	comm	communicator (handle)	13
OUT	request	communication request (handle)	14
001	request	communication request (manue)	15 16
C bindi	ng		17
int MPI		, void *recvbuf, int count,	18
		e, MPI_Op op, MPI_Comm comm,	19
	MPI_Request *request)	20
	2008 binding		21 22
		, datatype, op, comm, request, ierror)	22
		(IN), ASYNCHRONOUS :: sendbuf	24
	E(*), DIMENSION(), ASYNCH EGER, INTENT(IN) :: count	RUNUUS :: recvbuf	25
	EGER, INTENI(IN) Count E(MPI_Datatype), INTENT(IN)	· · · datatype	26
	E(MPI_Op), INTENT(IN) :: or		27
	E(MPI_Comm), INTENT(IN) ::		28
TYPI	E(MPI_Request), INTENT(OUT)	:: request	29
INT	EGER, OPTIONAL, INTENT(OUT)	:: ierror	30 31
Fortran	binding		32
	5	, DATATYPE, OP, COMM, REQUEST, IERROR)	33
	<pre>De> SENDBUF(*), RECVBUF(*)</pre>		34
INT	EGER COUNT, DATATYPE, OP, O	COMM, REQUEST, IERROR	35
This	call starts a nonblocking varia	nt of MPI_SCAN (see Section 5.11).	36
1 1110			37
			38
			39 40
			40 41
			42
			43
			44
			45
			46
			47

1

```
226
                                          CHAPTER 5. COLLECTIVE COMMUNICATION
1
     5.12.12 Nonblocking Exclusive Scan
\mathbf{2}
3
4
     MPI_IEXSCAN(sendbuf, recvbuf, count, datatype, op, comm, request)
5
       IN
                 sendbuf
                                             starting address of send buffer (choice)
6
       OUT
7
                 recvbuf
                                              starting address of receive buffer (choice)
8
       IN
                                              number of elements in input buffer (non-negative
                 count
9
                                             integer)
10
       IN
                 datatype
                                              data type of elements of input buffer (handle)
11
       IN
                                              operation (handle)
12
                 op
13
       IN
                                              intracommunicator (handle)
                 comm
14
       OUT
                                             communication request (handle)
                                                                                request
15
16
     C binding
17
     int MPI_Iexscan(const void *sendbuf, void *recvbuf, int count,
18
                     MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
19
                     MPI_Request *request)
20
21
     Fortran 2008 binding
22
     MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
23
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
^{24}
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
25
          INTEGER, INTENT(IN) :: count
26
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
          TYPE(MPI_Op), INTENT(IN) :: op
28
          TYPE(MPI_Comm), INTENT(IN) :: comm
29
          TYPE(MPI_Request), INTENT(OUT) :: request
30
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
     Fortran binding
32
     MPI_IEXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)
33
         <type> SENDBUF(*), RECVBUF(*)
34
          INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR
35
36
          This call starts a nonblocking variant of MPI_EXSCAN (see Section 5.11.2).
37
38
             Persistent Collective Operations
     5.13
39
40
     Many parallel computation algorithms involve repetitively executing a collective commu-
41
     nication operation with the same arguments each time. As with persistent point-to-point
42
     operations (see Section 3.9), persistent collective operations allow the MPI programmer to
43
     specify operations that will be reused frequently (with fixed arguments). MPI can be de-
44
     signed to select a more efficient way to perform the collective operation based on the param-
45
     eters specified when the operation is initialized. This "planned-transfer" approach [48, 38]
46
     can offer significant performance benefits for programs with repetitive communication pat-
47
```

 $_{48}$ terns.

In terms of data movement, each persistent collective operation has the same effect as its blocking and nonblocking counterparts for intracommunicators and intercommunicators after completion. Likewise, upon completion, persistent collective reduction operations perform the same operation as their blocking and nonblocking counterparts, and the same restrictions and recommendations on reduction orders apply (see also Section 5.9.1).

Initialization calls for MPI persistent collective operations are non-local and follow all the existing rules for collective operations, in particular ordering; programs that do not conform to these restrictions are erroneous. After initialization, all arrays associated with input arguments (such as arrays of counts, displacements, and datatypes in the vector versions of the collectives) must not be modified until the corresponding persistent request is freed with MPI_REQUEST_FREE.

The request argument is an output argument that can be used zero or more times with MPI_START or MPI_STARTALL in order to start the collective operation. The request is initially inactive after the initialization call. Once initialized, persistent collective operations can be started in any order and the order can differ among processes in the communicator.

Rationale. All ordering requirements that an implementation may need to match up collective operations across the communicator are achieved through the ordering requirements of the initialization functions. This enables out-of-order starts for the persistent operations, and particularly supports their use in MPI_STARTALL. (*End of rationale.*)

Advice to implementors. An MPI implementation should do no worse than duplicating the communicator during the initialization function, caching the input arguments, and calling the appropriate nonblocking collective function, using the cached arguments, during MPI_START. High-quality implementations should be able to amortize setup costs and further optimize by taking advantage of early-binding, such as efficient and effective pre-allocation of certain resources and algorithm selection. (*End* of advice to implementors.)

A request must be inactive when it is started. Starting the operation makes the request active. Once any process starts a persistent collective operation, it must complete that operation and all other processes in the communicator must eventually start (and complete) the same persistent collective operation. Persistent collective operations cannot be matched with blocking or nonblocking collective operations. Completion of a persistent collective operation makes the corresponding request inactive. After starting a persistent collective operation, all associated send buffers must not be modified and all associated receive buffers must not be accessed until the corresponding persistent request is completed.

Completing a persistent collective request, for example using MPI_TEST or MPI_WAIT, makes it inactive, but does not free the request. This is the same behavior as for persistent point-to-point requests. Inactive persistent collective requests can be freed using MPI_REQUEST_FREE. It is erroneous to free an active persistent collective request. Persistent collective operations cannot be canceled; it is erroneous to use MPI_CANCEL on a persistent collective request.

For every nonblocking collective communication operation in MPI, there is a corresponding persistent collective operation with the analogous API signature.

The collective persistent API signatures include an MPI_INFO object in order to support optimization hints and other information that may be non-standard. Persistent collective

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

```
1
     operations may be optimized during communicator creation or by the initialization opera-
\mathbf{2}
      tion of an individual persistent collective. Note that communicator-scoped hints should be
3
      provided using MPI_COMM_SET_INFO while, for operation-scoped hints, they are supplied
4
     to the persistent collective communication initialization functions using the info argument.
5
6
      5.13.1 Persistent Barrier Synchronization
7
8
9
      MPI_BARRIER_INIT(comm, info, request)
10
       IN
                 comm
                                               communicator (handle)
11
12
       IN
                  info
                                              info argument (handle)
13
        OUT
                  request
                                               communication request (handle)
14
15
      C binding
16
      int MPI_Barrier_init(MPI_Comm comm, MPI_Info info, MPI_Request *request)
17
18
      Fortran 2008 binding
19
      MPI_Barrier_init(comm, info, request, ierror)
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
      Fortran binding
25
     MPI_BARRIER_INIT(COMM, INFO, REQUEST, IERROR)
26
          INTEGER COMM, INFO, REQUEST, IERROR
27
28
          Creates a persistent collective communication request for the barrier operation.
29
30
     5.13.2 Persistent Broadcast
^{31}
32
33
      MPI_BCAST_INIT(buffer, count, datatype, root, comm, info, request)
34
       INOUT
                 buffer
                                               starting address of buffer (choice)
35
36
       IN
                  count
                                               number of entries in buffer (non-negative integer)
37
       IN
                 datatype
                                               data type of buffer (handle)
38
39
       IN
                  root
                                               rank of broadcast root (integer)
40
       IN
                  comm
                                               communicator (handle)
41
                 info
       IN
                                              info argument (handle)
42
       OUT
                                               communication request (handle)
43
                  request
44
45
      C binding
46
      int MPI_Bcast_init(void *buffer, int count, MPI_Datatype datatype,
47
                     int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)
```

```
Unofficial Draft for Comment Only
```

MPI_Bcast TYPE(INTEG TYPE(I TYPE(I	<pre>Fortran 2008 binding MPI_Bcast_init(buffer, count, datatype, root, comm, info, request, ierror) TYPE(*), DIMENSION(), ASYNCHRONOUS :: buffer INTEGER, INTENT(IN) :: count, root TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm</pre>			
TYPE(1	MPI_Info), INTENT(IN) :: MPI_Request), INTENT(OUT) ER, OPTIONAL, INTENT(OUT)	:: request	7 8 9	
<type INTEG</type 	_INIT(BUFFER, COUNT, DATA > BUFFER(*) ER COUNT, DATATYPE, ROOT,	TYPE, ROOT, COMM, INFO, REQUEST, IERROR) COMM, INFO, REQUEST, IERROR nunication request for the broadcast operation.	10 11 12 13 14 15	
	rsistent Gather	unication request for the broadcast operation.	16 17 18	
MPI_GATH	ER_INIT(sendbuf, sendcount, s info, request)	sendtype, recvbuf, recvcount, recvtype, root, comm,	19 20 21 22	
IN	sendbuf	starting address of send buffer (choice)	22	
IN	sendcount	number of elements in send buffer (non-negative integer)	24 25	
IN	sendtype	data type of send buffer elements (handle)	26 27	
OUT	recvbuf	address of receive buffer (choice, significant only at root)	27 28 29	
IN	recvcount	number of elements for any single receive (non-negative integer, significant only at root)	30 31	
IN	recvtype	data type of recv buffer elements (handle, significant only at root)	32 33 34	
IN	root	rank of receiving process (integer)	35	
IN	comm	communicator (handle)	36	
IN	info	info argument (handle)	37 38	
OUT	request	communication request (handle)	39	
C binding int MPI_Gather_init(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info, MPI_Request *request) 40 41 42 42 43 44 45 46				
Fortran 2	Fortran 2008 binding 47 48			

1 MPI_Gather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, $\mathbf{2}$ root, comm, info, request, ierror) 3 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 4 INTEGER, INTENT(IN) :: sendcount, recvcount, root 5TYPE(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 Fortran binding 12MPI_GATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 13 ROOT, COMM, INFO, REQUEST, IERROR) 14 <type> SENDBUF(*), RECVBUF(*) 15INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO, 16REQUEST, IERROR 17 18 Creates a persistent collective communication request for the gather operation. 19 20MPI_GATHERV_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, 21root, comm, info, request) 22 23IN sendbuf starting address of send buffer (choice) 24IN sendcount number of elements in send buffer (non-negative 25integer) 26IN data type of send buffer elements (handle) sendtype 2728OUT address of receive buffer (choice, significant only at recvbuf 29 root) 30 IN non-negative integer array (of length group size) recvcounts 31 containing the number of elements that are received 32 from each process (significant only at root) 33 IN displs integer array (of length group size). Entry i specifies 34 the displacement relative to recvbuf at which to place 35 the incoming data from process i (significant only at 36 root) 37 38 IN recvtype data type of recv buffer elements (handle, significant 39 only at root) 40 rank of receiving process (integer) IN root 41 IN communicator (handle) comm 4243 IN info info argument (handle) 44 OUT communication request (handle) request 45 46 C binding 4748

```
1
int MPI_Gatherv_init(const void *sendbuf, int sendcount,
                                                                                      \mathbf{2}
              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)
                                                                                      4
                                                                                      5
Fortran 2008 binding
                                                                                      6
MPI_Gatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                      7
              recvtype, root, comm, info, request, ierror)
                                                                                      8
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                      9
    INTEGER, INTENT(IN) :: sendcount, root
                                                                                      10
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                      11
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                      12
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
                                                                                      13
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                      14
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                      15
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                      16
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                      17
                                                                                      18
Fortran binding
                                                                                      19
MPI_GATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
              RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)
                                                                                      20
                                                                                      21
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
                                                                                      22
                                                                                      23
               COMM, INFO, REQUEST, IERROR
                                                                                      ^{24}
    Creates a persistent collective communication request for the gathery operation.
                                                                                      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 5. COLLECTIVE COMMUNICATION

1	5.13.4 P	ersistent Scatter	
2 3			
4 5	MPI_SCA	TTER_INIT(sendbuf, sendcou comm, info, request)	nt, sendtype, recvbuf, recvcount, recvtype, root,
6 7 8	IN	sendbuf	address of send buffer (choice, significant only at root)
9 10	IN	sendcount	number of elements sent to each process (non-negative integer, significant only at root)
11 12	IN	sendtype	data type of send buffer elements (handle, significant only at root)
$13 \\ 14$	OUT	recvbuf	address of receive buffer (choice)
15 16	IN	recvcount	number of elements in receive buffer (non-negative integer)
17 18	IN	recvtype	data type of receive buffer elements (handle)
19	IN	root	rank of sending process (integer)
20	IN	comm	communicator (handle)
21 22	IN	info	info argument (handle)
22	OUT	request	communication request (handle)
26 27 28 29 30 31 32 33 34 35 36 37 38	Fortran 2 MPI_Scatt TYPE INTEC TYPE TYPE	<pre>MPI_Datatype sendty MPI_Datatype recvty MPI_Request *reques 2008 binding ter_init(sendbuf, sendco recvtype, root, com (*), DIMENSION(), INTE SER, INTENT(IN) :: sendc</pre>	<pre>unt, sendtype, recvbuf, recvcount, mm, info, request, ierror) NT(IN), ASYNCHRONOUS :: sendbuf count, recvcount, root N) :: sendtype, recvtype CHRONOUS :: recvbuf : comm</pre>
39		(MPI_Request), INTENT(OU	
40 41	INTEG	GER, OPTIONAL, INTENT(OU	T) :: ierror
42 43 44 45 46 47	<type< td=""><td>IER_INIT(SENDBUF, SENDCO RECVTYPE, ROOT, COM e> SENDBUF(*), RECVBUF(*</td><td>UNT, SENDTYPE, RECVBUF, RECVCOUNT, MM, INFO, REQUEST, IERROR)) RECVCOUNT, RECVTYPE, ROOT, COMM, INFO,</td></type<>	IER_INIT(SENDBUF, SENDCO RECVTYPE, ROOT, COM e> SENDBUF(*), RECVBUF(*	UNT, SENDTYPE, RECVBUF, RECVCOUNT, MM, INFO, REQUEST, IERROR)) RECVCOUNT, RECVTYPE, ROOT, COMM, INFO,
48			

Creates a persistent collective communication request for the scatter operation.

MPI_SCATTERV_INIT(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, rec	vtype,
root, comm, info, request)	

	root, comm, mio, request	·)	Б
IN	sendbuf	address of send buffer (choice, significant only at root)	6 7
IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank (significant only at root)	8 9 10 11
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)	12 13 14 15
IN	sendtype	data type of send buffer elements (handle, significant only at root)	16 17
OUT	recvbuf	address of receive buffer (choice, significant only at root)	18 19 20
IN	recvcount	number of elements in receive buffer (non-negative integer)	20 21 22
IN	recvtype	data type of receive buffer elements (handle)	23
IN	root	rank of sending process (integer)	24 25
IN	comm	communicator (handle)	26
IN	info	info argument (handle)	27
OUT	request	communication request (handle)	28
			29
C binding	g		30 31
int MPI_S	catterv_init(const void *	<pre>*sendbuf, const int sendcounts[],</pre>	32
	<pre>const int displs[], MPI_Datatype sendtype, void *recvbuf,</pre>		
int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm,			
	MPI_Info info, MPI_R	equest *request)	35
Fortran 2	Fortran 2008 binding		
MPI_Scatterv_init(sendbuf, sendcounts, displs, sendtype, recvbuf,			
			38

38recvcount, recvtype, root, comm, info, request, ierror) 39TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 40INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*) 41 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 42TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf 43 INTEGER, INTENT(IN) :: recvcount, root 44TYPE(MPI_Comm), INTENT(IN) :: comm 45TYPE(MPI_Info), INTENT(IN) :: info 46TYPE(MPI_Request), INTENT(OUT) :: request 47INTEGER, OPTIONAL, INTENT(OUT) :: ierror 48

1 2 3 4 5 6 7	<pre>Fortran binding MPI_SCATTERV_INIT(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF,</pre>			
8	Create	es a persistent collective comm	nunication request for the scattery operation.	
9 10 11 12	5.13.5 Pe	ersistent Gather-to-all		
13 14	MPI_ALLG	GATHER_INIT(sendbuf, sendcouinfo, request)	unt, sendtype, recvbuf, recvcount, recvtype, comm,	
15 16	IN	sendbuf	starting address of send buffer (choice)	
10 17 18	IN	sendcount	number of elements in send buffer (non-negative integer)	
19	IN	sendtype	data type of send buffer elements (handle)	
20 21	OUT	recvbuf	address of receive buffer (choice)	
21 22 23	IN	recvcount	number of elements received from any process (non-negative integer)	
24	IN	recvtype	data type of receive buffer elements (handle)	
25 26	IN	comm	communicator (handle)	
27	IN	info	info argument (handle)	
28 29	OUT	request	communication request (handle)	
30	C binding	g		
31 32	int MPI_A	allgather_init(const void		
33			e, void *recvbuf, int recvcount, e MPI Comm comm MPI Info info	
34	<pre>MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, MPI_Request *request)</pre>			
35	Fortran 2008 binding			
$\frac{36}{37}$	MPI_Allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,			
38		recvtype, comm, info	-	
39	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf			
40 41	INTEGER, INTENT(IN) :: sendcount, recvcount TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype			
42	TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf			
43	TYPE(MPI_Comm), INTENT(IN) :: comm			
44 45	TYPE(MPI_Info), INTENT(IN) :: info TYPE(MPI_Request), INTENT(OUT) :: request			
45 46	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
47 48	Fortran b	binding		

	RECVTYPE, COMM, INFO		
01	> SENDBUF(*), RECVBUF(*) ER SENDCOUNT, SENDTYPE, R IERROR	ECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,	
Create	es a persistent collective comm	nunication request for the allgather operation.	
MPI_ALLG	ATHERV_INIT(sendbuf, sendco comm, info, request)	ount, sendtype, recvbuf, recvcounts, displs, recvtype,	
IN	sendbuf	starting address of send buffer (choice)	
IN	sendcount	number of elements in send buffer (non-negative 1 integer) 1	
IN	sendtype	data type of send buffer elements (handle)	
OUT	recvbuf	address of receive buffer (choice)	
IN	recvcounts	non-negative integer array (of length group size)1containing the number of elements that are received1from each process2	
IN	displs	integer array (of length group size). Entry i specifies2the displacement (relative to recvbuf) at which to2place the incoming data from process i2	
IN	recvtype	data type of receive buffer elements (handle)	
IN	comm	communicator (handle) ²	
IN	info	info argument (handle)	
OUT	request	communication request (handle)	
		3	
C binding		3	
int MPI_A	-	<pre>1 *sendbuf, int sendcount, 3 e, void *recvbuf, const int recvcounts[], 3</pre>	
		e, void *recvbuf, const int recvcounts[], 3 MPI_Datatype recvtype, MPI_Comm comm, 3	
	MPI_Info info, MPI_Re		
Fortran 2	008 binding	3	
	U	ount, sendtype, recvbuf, recvcounts,	
		mm, info, request, ierror)	
		'(IN), ASYNCHRONOUS :: sendbuf	
	ER, INTENT(IN) :: sendcou	4	
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf			
INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)			
TYPE(MPI_Comm), INTENT(IN) :: comm			
TYPE(MPI_Info), INTENT(IN) :: info			
TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
INTEG	ER, UPIIUNAL, INIENI(UUI)	:: lerror 4	

1	Fortran k	ainding			
2	Fortran binding MPI_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,				
3		DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)			
4	<type< th=""><th><pre>> SENDBUF(*), RECVBUF(*)</pre></th><th>,,,,</th></type<>	<pre>> SENDBUF(*), RECVBUF(*)</pre>	,,,,		
5	• 1		RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,		
6		INFO, REQUEST, IERR			
7	Crost	og a porgistant collective com	munication request for the allgathery operation.		
8 9	Cleat	es a persistent conective com	numeration request for the angathery operation.		
10	5.13.6 P	ersistent All-to-All Scatter/G	ather		
11					
12 13 14	MPI_ALLT	OALL_INIT(sendbuf, sendcour info, request)	nt, sendtype, recvbuf, recvcount, recvtype, comm,		
15	IN	sendbuf	starting address of send buffer (choice)		
16	IN	sendcount	number of elements sent to each process		
17 18		Sendcount	(non-negative integer)		
19	IN	sendtype	data type of send buffer elements (handle)		
20 21	OUT	recvbuf	address of receive buffer (choice)		
21 22 23	IN	recvcount	number of elements received from any process (non-negative integer)		
24	IN	recvtype	data type of receive buffer elements (handle)		
25 26	IN	comm	communicator (handle)		
27	IN	info	info argument (handle)		
28	OUT	request	communication request (handle)		
29 30					
31	C binding	g Alltoall_init(const void	toondbuf int condcount		
32	IIIC PHI_F		be, void *recvbuf, int recvcount,		
33			pe, MPI_Comm comm, MPI_Info info,		
34		MPI_Request *request			
35	The data of the				
36		2008 binding	unt, sendtype, recvbuf, recvcount,		
37	MPI_AILU	recvtype, comm, info	• -		
38	TYPE		-		
$\frac{39}{40}$	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf INTEGER, INTENT(IN) :: sendcount, recvcount				
40	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	INTEC	GER, OPTIONAL, INTENT(OUT) :: ierror		
47	Fortran h	oinding			
48					

MPI.	RECVTYPE, CO	, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, MM, INFO, REQUEST, IERROR)
	<type> SENDBUF(*), REG INTEGER SENDCOUNT, SEN IERROR</type>	NDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
	Creates a persistent collec	tive communication request for the alltoall operation.
MPI	_ALLTOALLV_INIT(sendbu recvtype, comm	
IN	sendbuf	starting address of send buffer (choice)
IN	sendcounts	non-negative integer array (of length group size)1specifying the number of elements to send to each1rank1
IN	sdispls	Integer array (of length group size). Entry j specifies the displacement (relative to sendbuf) from which to take the outgoing data destined for process j
IN	sendtype	data type of send buffer elements (handle)
0	UT recvbuf	address of receive buffer (choice) 2
IN	recvcounts	non-negative integer array (of length group size)2specifying the number of elements that can be2received from each rank2
IN	rdispls	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i
IN	recvtype	data type of receive buffer elements (handle) 2
IN	comm	communicator (handle) 33
IN	info	info argument (handle)
0	UT request	communication request (handle) 33
C b	inding	3
int		<pre>ist void *sendbuf, const int sendcounts[],</pre>
		<pre>ispls[], MPI_Datatype sendtype, void *recvbuf, 3 cvcounts[], const int rdispls[], 3</pre>
		recvtype, MPI_Comm comm, MPI_Info info,
	MPI_Request	
For	tran 2008 binding	4
		f, sendcounts, sdispls, sendtype, recvbuf, 4
		rdispls, recvtype, comm, info, request, ierror) 4
), INTENT(IN), ASYNCHRONOUS :: sendbuf ASYNCHRONOUS :: sendcounts(*), sdispls(*), 4
		<pre>ASYNCHRONOUS :: Senacounts(*), saispis(*), 4 *), rdispls(*) 4</pre>
		ITENT(IN) :: sendtype, recvtype 4

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16	TYPE TYPE TYPE INTE Fortran MPI_ALLT <typ INTE Creat</typ 	OALLV_INIT(SENDBUF, SEND RECVCOUNTS, RDISPLS e> SENDBUF(*), RECVBUF(* GER SENDCOUNTS(*), SDISP RECVTYPE, COMM, IN	: comm : info T) :: request T) :: ierror COUNTS, SDISPLS, SENDTYPE, RECVBUF, S, RECVTYPE, COMM, INFO, REQUEST, IERROR)
17		recvtypes, comm, info,	request)
18 19	IN	sendbuf	starting address of send buffer (choice)
20 21 22	IN	sendcounts	integer array (of length group size) specifying the number of elements to send to each rank (array of non-negative integers)
23 24 25 26	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)
27 28 29	IN	sendtypes	Array of datatypes (of length group size). Entry j specifies the type of data to send to process j (array of handles)
30 31	OUT	recvbuf	address of receive buffer (choice)
32 33 34	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)
35 36 37 38	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)
39 40 41 42	IN	recvtypes	array of datatypes (of length group size). Entry i specifies the type of data received from process i (array of handles)
43	IN	comm	communicator (handle)
44 45	IN	info	info argument (handle)
45 46	OUT	request	communication request (handle)
47 48	C bindin	g	

1 int MPI_Alltoallw_init(const void *sendbuf, const int sendcounts[], 2 const int sdispls[], const MPI_Datatype sendtypes[], 3 void *recvbuf, const int recvcounts[], const int rdispls[], const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info, $\mathbf{4}$ MPI_Request *request) 5 6 Fortran 2008 binding MPI_Alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm, info, request, ierror) 9 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 10 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*), 11 recvcounts(*), rdispls(*) 12TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*), 13 recvtypes(*) 14TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf 15TYPE(MPI_Comm), INTENT(IN) :: comm 16TYPE(MPI_Info), INTENT(IN) :: info 17 TYPE(MPI_Request), INTENT(OUT) :: request 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 19 Fortran binding 2021MPI_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, 22RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST, IERROR) 23<type> SENDBUF(*), RECVBUF(*) 24 INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*), RDISPLS(*), RECVTYPES(*), COMM, INFO, REQUEST, IERROR 2526Creates a persistent collective communication request for the alltoally operation. 27285.13.7 Persistent Reduce 29 30 31MPI_REDUCE_INIT(sendbuf, recvbuf, count, datatype, op, root, comm, info, request) 32 33 IN sendbuf address of send buffer (choice) 34 OUT recvbuf address of receive buffer (choice, significant only at 35 root) 36 IN count number of elements in send buffer (non-negative 37 integer) 38 39 IN datatype data type of elements of send buffer (handle) 40 IN reduce operation (handle) op 41 IN root rank of root process (integer) 4243 IN communicator (handle) comm 44 IN info info argument (handle) 45OUT request communication request (handle) 464748

C binding

```
1
     int MPI_Reduce_init(const void *sendbuf, void *recvbuf, int count,
\mathbf{2}
                    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
3
                    MPI_Info info, MPI_Request *request)
4
     Fortran 2008 binding
5
     MPI_Reduce_init(sendbuf, recvbuf, count, datatype, op, root, comm, info,
6
                    request, ierror)
7
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
8
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
9
          INTEGER, INTENT(IN) :: count, root
10
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
11
          TYPE(MPI_Op), INTENT(IN) :: op
12
          TYPE(MPI_Comm), INTENT(IN) :: comm
13
          TYPE(MPI_Info), INTENT(IN) :: info
14
          TYPE(MPI_Request), INTENT(OUT) :: request
15
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
     Fortran binding
18
     MPI_REDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, INFO,
19
                    REQUEST, IERROR)
20
          <type> SENDBUF(*), RECVBUF(*)
21
          INTEGER COUNT, DATATYPE, OP, ROOT, COMM, INFO, REQUEST, IERROR
22
          Creates a persistent collective communication request for the reduce operation.
23
24
     5.13.8 Persistent All-Reduce
25
26
27
     MPI_ALLREDUCE_INIT(sendbuf, recvbuf, count, datatype, op, comm, info, request)
28
29
       IN
                 sendbuf
                                             starting address of send buffer (choice)
30
       OUT
                 recvbuf
                                             starting address of receive buffer (choice)
^{31}
32
       IN
                                             number of elements in send buffer (non-negative
                 count
33
                                             integer)
34
       IN
                 datatype
                                             data type of elements of send buffer (handle)
35
       IN
                                             operation (handle)
                 ор
36
37
       IN
                 comm
                                             communicator (handle)
38
       IN
                 info
                                             info argument (handle)
39
       OUT
                 request
                                             communication request (handle)
40
41
     C binding
42
     int MPI_Allreduce_init(const void *sendbuf, void *recvbuf, int count,
43
                    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
44
                    MPI_Info info, MPI_Request *request)
45
46
     Fortran 2008 binding
47
48
```

```
1
MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info,
                                                                                          2
               request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                         10
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                         11
                                                                                         12
Fortran binding
                                                                                         13
MPI_ALLREDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO,
                                                                                         14
               REQUEST, IERROR)
                                                                                         15
    <type> SENDBUF(*), RECVBUF(*)
                                                                                         16
    INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
                                                                                         17
                                                                                         18
    Creates a persistent collective communication request for the all reduce operation.
                                                                                         19
                                                                                         20
5.13.9 Persistent Reduce-Scatter with Equal Blocks
                                                                                         21
                                                                                         22
                                                                                         23
MPI_REDUCE_SCATTER_BLOCK_INIT(sendbuf, recvbuf, recvcount, datatype, op, comm,
                                                                                         24
               info, request)
                                                                                         25
  IN
           sendbuf
                                       starting address of send buffer (choice)
                                                                                         26
                                                                                         27
  OUT
           recvbuf
                                       starting address of receive buffer (choice)
                                                                                         28
  IN
                                       element count per block (non-negative integer)
           recvcount
                                                                                         29
                                       data type of elements of send and receive buffers
  IN
           datatype
                                                                                         30
                                       (handle)
                                                                                         31
                                                                                         32
  IN
                                       operation (handle)
           op
                                                                                         33
                                       communicator (handle)
  IN
           comm
                                                                                         34
  IN
           info
                                       info argument (handle)
                                                                                         35
                                                                                         36
  OUT
           request
                                       communication request (handle)
                                                                                         37
                                                                                         38
C binding
                                                                                         39
int MPI_Reduce_scatter_block_init(const void *sendbuf, void *recvbuf,
                                                                                         40
               int recvcount, MPI_Datatype datatype, MPI_Op op,
                                                                                         41
               MPI_Comm comm, MPI_Info info, MPI_Request *request)
                                                                                         42
```

```
Fortran 2008 binding
```

```
1
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
2
          TYPE(MPI_Op), INTENT(IN) :: op
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
     Fortran binding
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
13
          Creates a persistent collective communication request for the reduce-scatter with equal
14
     blocks operation.
15
16
     5.13.10 Persistent Reduce-Scatter
17
18
19
     MPI_REDUCE_SCATTER_INIT(sendbuf, recvbuf, recvcounts, datatype, op, comm, info,
20
                     request)
21
       IN
                 sendbuf
                                             starting address of send buffer (choice)
22
23
                                             starting address of receive buffer (choice)
       OUT
                 recvbuf
24
       IN
                 recvcounts
                                             non-negative integer array specifying the number of
25
                                             elements in result distributed to each process. This
26
                                             array must be identical on all calling processes.
27
       IN
                 datatype
                                             data type of elements of input buffer (handle)
28
29
       IN
                                             operation (handle)
                 ор
30
       IN
                                             communicator (handle)
                 comm
^{31}
       IN
                 info
                                             info argument (handle)
32
33
       OUT
                 request
                                             communication request (handle)
34
35
     C binding
36
     int MPI_Reduce_scatter_init(const void *sendbuf, void *recvbuf,
37
                     const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
38
                    MPI_Comm comm, MPI_Info info, MPI_Request *request)
39
     Fortran 2008 binding
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, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
45
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
46
          TYPE(MPI_Op), INTENT(IN) :: op
47
          TYPE(MPI_Comm), INTENT(IN) :: comm
48
```

	TYPE(MPI_Info), INTENT(IN) :: info		
	TYPE(MPI_Request), INTENT(OUT) :: request		
TNT	INTEGER, OPTIONAL, INTENT(OUT) :: ierror		
	binding		5
MPI_RED		BUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,	6
	INFO, REQUEST,		7
•	pe> SENDBUF(*), RECVB		8
	INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, INFO, REQUEST, IERROR		
Crea	ates a persistent collective	e communication request for the reduce-scatter operation.	10 11
- 10 11			11
5.13.11	Persistent Inclusive Sca	n	13
			14
		annet detecture on anne infer varuant)	15
		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 20
		integer)	20 21
IN	datatype	data type of elements of input buffer (handle)	22
IN	ор	operation (handle)	23
IN	comm	communicator (handle)	24
			25
IN	info	info argument (handle)	26
OUT	request	communication request (handle)	27 28
~			20
C bindi	U		30
int MPI		<pre>*sendbuf, void *recvbuf, int count, tatype, MPI_Op op, MPI_Comm comm,</pre>	31
		MPI_Request *request)	32
			33
	2008 binding		34
MP1_Sca	n_init(sendbuf, recvb ierror)	uf, count, datatype, op, comm, info, request,	35
тур		INTENT(IN) ASYNCHRONOUS · · condbuf	36 37
	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf		
	INTEGER, INTENT(IN) :: count		
	TYPE(MPI_Datatype), INTENT(IN) :: datatype		
TYP	TYPE(MPI_Op), INTENT(IN) :: op		
	TYPE(MPI_Comm), INTENT(IN) :: comm		
TYPE(MPI_Info), INTENT(IN) :: info			43 44
	TYPE(MPI_Request), INTENT(OUT) :: request		
TNT	INTEGER, OPTIONAL, INTENT(OUT) :: ierror		
Fortran binding			46 47
			48

```
1
     MPI_SCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,
\mathbf{2}
                     IERROR)
3
          <type> SENDBUF(*), RECVBUF(*)
4
          INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
5
          Creates a persistent collective communication request for the inclusive scan operation.
6
7
     5.13.12 Persistent Exclusive Scan
8
9
10
     MPI_EXSCAN_INIT(sendbuf, recvbuf, count, datatype, op, comm, info, request)
11
12
       IN
                 sendbuf
                                             starting address of send buffer (choice)
13
       OUT
                 recvbuf
                                             starting address of receive buffer (choice)
14
                                             number of elements in input buffer (non-negative
15
       IN
                 count
16
                                             integer)
17
       IN
                 datatype
                                             data type of elements of input buffer (handle)
18
       IN
                                             operation (handle)
                 ор
19
       IN
                                             intracommunicator (handle)
20
                 comm
21
       IN
                 info
                                             info argument (handle)
22
       OUT
                 request
                                             communication request (handle)
23
24
     C binding
25
     int MPI_Exscan_init(const void *sendbuf, void *recvbuf, int count,
26
                    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
27
                    MPI_Info info, MPI_Request *request)
28
29
     Fortran 2008 binding
30
     MPI_Exscan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
^{31}
                    ierror)
32
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
33
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
34
          INTEGER, INTENT(IN) :: count
35
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
36
          TYPE(MPI_Op), INTENT(IN) :: op
37
          TYPE(MPI_Comm), INTENT(IN) :: comm
38
          TYPE(MPI_Info), INTENT(IN) :: info
39
          TYPE(MPI_Request), INTENT(OUT) :: request
40
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
     Fortran binding
42
     MPI_EXSCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,
43
                    IERROR)
44
          <type> SENDBUF(*), RECVBUF(*)
45
          INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
46
47
          Creates a persistent collective communication request for the exclusive scan operation.
48
```

5.14Correctness

A correct, portable program must invoke collective communications so that deadlock will not occur, whether collective communications are synchronizing or not. The following examples illustrate dangerous use of collective routines on intracommunicators.

Example 5.25 The following is erroneous.

```
switch(rank) {
   case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Bcast(buf2, count, type, 1, comm);
        break;
   case 1:
        MPI_Bcast(buf2, count, type, 1, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
```

}

We assume that the group of comm is $\{0,1\}$. Two processes execute two broadcast operations in reverse order. If the operation is synchronizing then a deadlock will occur.

Collective operations must be executed in the same order at all members of the communication group.

Example 5.26 The following is erroneous.

```
switch(rank) {
   case 0:
        MPI_Bcast(buf1, count, type, 0, comm0);
        MPI_Bcast(buf2, count, type, 2, comm2);
        break;
    case 1:
        MPI_Bcast(buf1, count, type, 1, comm1);
        MPI_Bcast(buf2, count, type, 0, comm0);
        break;
    case 2:
        MPI_Bcast(buf1, count, type, 2, comm2);
        MPI_Bcast(buf2, count, type, 1, comm1);
        break;
```

}

Assume that the group of comm0 is $\{0,1\}$, of comm1 is $\{1, 2\}$ and of comm2 is $\{2,0\}$. If the broadcast is a synchronizing operation, then there is a cyclic dependency: the broadcast in comm2 completes only after the broadcast in comm0; the broadcast in comm0 completes only after the broadcast in comm1; and the broadcast in comm1 completes only after the broadcast in comm2. Thus, the code will deadlock.

Collective operations must be executed in an order so that no cyclic dependencies occur. Nonblocking collective operations can alleviate this issue.

Example 5.27 The following is erroneous.

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

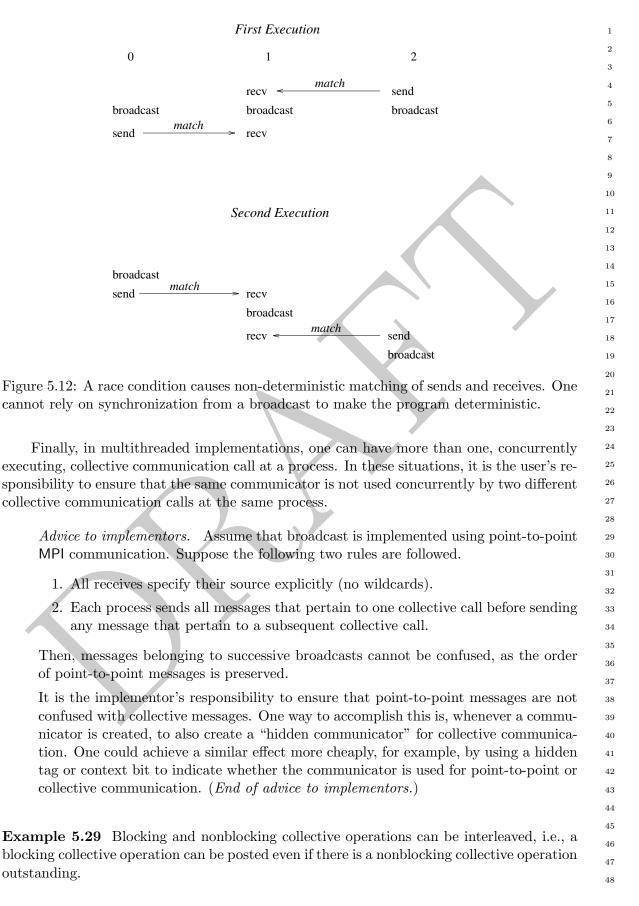
43

44

45

46

```
1
     switch(rank) {
2
          case 0:
3
               MPI_Bcast(buf1, count, type, 0, comm);
4
               MPI_Send(buf2, count, type, 1, tag, comm);
5
               break:
6
          case 1:
7
               MPI_Recv(buf2, count, type, 0, tag, comm, status);
8
               MPI_Bcast(buf1, count, type, 0, comm);
9
               break;
10
      }
11
          Process zero executes a broadcast, followed by a blocking send operation. Process one
12
      first executes a blocking receive that matches the send, followed by broadcast call that
13
      matches the broadcast of process zero. This program may deadlock. The broadcast call on
14
      process zero may block until process one executes the matching broadcast call, so that the
15
16
      send is not executed. Process one will definitely block on the receive and so, in this case,
      never executes the broadcast.
17
          The relative order of execution of collective operations and point-to-point operations
18
19
      should be such, so that even if the collective operations and the point-to-point operations
      are synchronizing, no deadlock will occur.
20
21
      Example 5.28 An unsafe, non-deterministic program.
22
23
      switch(rank) {
24
          case 0:
25
               MPI_Bcast(buf1, count, type, 0, comm);
26
               MPI_Send(buf2, count, type, 1, tag, comm);
27
               break;
28
          case 1:
29
               MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
30
               MPI_Bcast(buf1, count, type, 0, comm);
31
               MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
32
               break;
33
          case 2:
34
               MPI_Send(buf2, count, type, 1, tag, comm);
35
               MPI_Bcast(buf1, count, type, 0, comm);
36
               break;
37
      }
38
39
          All three processes participate in a broadcast. Process 0 sends a message to process
40
      1 after the broadcast, and process 2 sends a message to process 1 before the broadcast.
41
      Process 1 receives before and after the broadcast, with a wildcard source argument.
42
          Two possible executions of this program, with different matchings of sends and receives,
43
      are illustrated in Figure 5.12. Note that the second execution has the peculiar effect that
44
      a send executed after the broadcast is received at another node before the broadcast. This
45
      example illustrates the fact that one should not rely on collective communication functions
46
      to have particular synchronization effects. A program that works correctly only when the
47
      first execution occurs (only when broadcast is synchronizing) is erroneous.
48
```



```
1
     MPI_Request req;
\mathbf{2}
3
     MPI_Ibarrier(comm, &req);
4
     MPI_Bcast(buf1, count, type, 0, comm);
5
     MPI_Wait(&req, MPI_STATUS_IGNORE);
6
          Each process starts a nonblocking barrier operation, participates in a blocking broad-
7
     cast and then waits until every other process started the barrier operation. This ef-
8
     fectively turns the broadcast into a synchronizing broadcast with possible communica-
9
     tion/communication overlap (MPI_Bcast is allowed, but not required to synchronize).
10
11
     Example 5.30 The starting order of collective operations on a particular communicator
12
     defines their matching. The following example shows an erroneous matching of different
13
     collective operations on the same communicator.
14
15
     MPI_Request req;
16
     switch(rank) {
17
          case 0:
18
              /* erroneous matching */
19
              MPI_Ibarrier(comm, &req);
20
              MPI_Bcast(buf1, count, type, 0, comm);
21
              MPI_Wait(&req, MPI_STATUS_IGNORE);
22
              break:
23
          case 1:
24
              /* erroneous matching */
25
              MPI_Bcast(buf1, count, type, 0, comm);
26
              MPI_Ibarrier(comm, &req);
27
              MPI_Wait(&req, MPI_STATUS_IGNORE);
28
              break;
29
     }
30
^{31}
          This ordering would match MPI_Ibarrier on rank 0 with MPI_Bcast on rank 1 which is
32
     erroneous and the program behavior is undefined. However, if such an order is required, the
33
     user must create different duplicate communicators and perform the operations on them.
34
     If started with two processes, the following program would be correct:
35
36
     MPI_Request req;
37
     MPI_Comm dupcomm;
38
     MPI_Comm_dup(comm, &dupcomm);
39
     switch(rank) {
40
          case 0:
41
              MPI_Ibarrier(comm, &req);
42
              MPI_Bcast(buf1, count, type, 0, dupcomm);
43
              MPI_Wait(&reg, MPI_STATUS_IGNORE);
44
              break:
45
          case 1:
46
              MPI_Bcast(buf1, count, type, 0, dupcomm);
47
              MPI_Ibarrier(comm, &req);
48
              MPI_Wait(&req, MPI_STATUS_IGNORE);
```

break;

}

Advice to users. The use of different communicators offers some flexibility regarding the matching of nonblocking collective operations. In this sense, communicators could be used as an equivalent to tags. However, communicator construction might induce overheads so that this should be used carefully. (*End of advice to users.*)

Example 5.31 Nonblocking collective operations can rely on the same progression rules as nonblocking point-to-point messages. Thus, if started with two processes, the following program is a valid MPI program and is guaranteed to terminate:

```
MPI_Request req;
```

```
switch(rank) {
   case 0:
    MPI_Ibarrier(comm, &req);
    MPI_Wait(&req, MPI_STATUS_IGNORE);
    MPI_Send(buf, count, dtype, 1, tag, comm);
    break;
   case 1:
    MPI_Ibarrier(comm, &req);
    MPI_Recv(buf, count, dtype, 0, tag, comm, MPI_STATUS_IGNORE);
    MPI_Wait(&req, MPI_STATUS_IGNORE);
    break;
}
```

}

The MPI library must progress the barrier in the MPI_Recv call. Thus, the MPI_Wait call in rank 0 will eventually complete, which enables the matching MPI_Send so all calls eventually return.

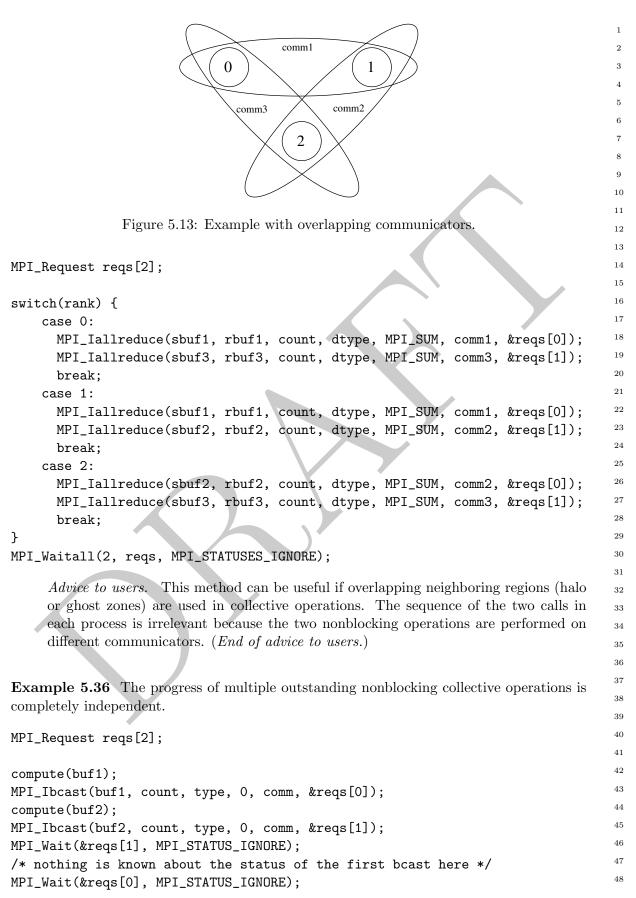
Example 5.32 Blocking and nonblocking collective operations do not match. The following example is erroneous.

```
MPI_Request req;
switch(rank) {
    case 0:
        /* erroneous false matching of Alltoall and Ialltoall */
        MPI_Ialltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm, &req);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        break;
    case 1:
        /* erroneous false matching of Alltoall and Ialltoall */
        MPI_Alltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm);
        break;
}
```

 $\mathbf{2}$

 24

```
1
     Example 5.33 Collective and point-to-point requests can be mixed in functions that
\mathbf{2}
     enable multiple completions. If started with two processes, the following program is valid.
3
     MPI_Request reqs[2];
4
5
     switch(rank) {
6
          case 0:
7
            MPI_Ibarrier(comm, &reqs[0]);
8
            MPI_Send(buf, count, dtype, 1, tag, comm);
9
            MPI_Wait(&reqs[0], MPI_STATUS_IGNORE);
10
            break:
11
          case 1:
12
            MPI_Irecv(buf, count, dtype, 0, tag, comm, &reqs[0]);
13
            MPI_Ibarrier(comm, &reqs[1]);
14
            MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);
15
            break;
16
     }
17
18
          The MPI_Waitall call returns only after the barrier and the receive completed.
19
     Example 5.34 Multiple nonblocking collective operations can be outstanding on a single
20
     communicator and match in order.
21
22
     MPI_Request reqs[3];
23
^{24}
     compute(buf1);
25
     MPI_Ibcast(buf1, count, type, 0, comm, &reqs[0]);
26
     compute(buf2);
27
     MPI_Ibcast(buf2, count, type, 0, comm, &reqs[1]);
28
     compute(buf3);
29
     MPI_Ibcast(buf3, count, type, 0, comm, &reqs[2]);
30
     MPI_Waitall(3, reqs, MPI_STATUSES_IGNORE);
^{31}
32
           Advice to users. Pipelining and double-buffering techniques can efficiently be used
33
           to overlap computation and communication. However, having too many outstanding
34
           requests might have a negative impact on performance. (End of advice to users.)
35
           Advice to implementors.
                                       The use of pipelining may generate many outstanding
36
           requests. A high-quality hardware-supported implementation with limited resources
37
           should be able to fall back to a software implementation if its resources are exhausted.
38
           In this way, the implementation could limit the number of outstanding requests only
39
           by the available memory. (End of advice to implementors.)
40
41
42
     Example 5.35 Nonblocking collective operations can also be used to enable simultane-
43
     ous collective operations on multiple overlapping communicators (see Figure 5.13). The
^{44}
     following example is started with three processes and three communicators. The first com-
45
     municator comm1 includes ranks 0 and 1, comm2 includes ranks 1 and 2, and comm3 spans
46
     ranks 0 and 2. It is not possible to perform a blocking collective operation on all commu-
47
     nicators because there exists no deadlock-free order to invoke them. However, nonblocking
```



1

 $\mathbf{2}$

3 4 5 Finishing the second MPI_IBCAST is completely independent of the first one. This means that it is not guaranteed that the first broadcast operation is finished or even started after the second one is completed via reqs[1].

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

Chapter 6

Groups, Contexts, Communicators, and Caching

6.1 Introduction

This chapter introduces MPI features that support the development of parallel libraries. Parallel libraries are needed to encapsulate the distracting complications inherent in parallel implementations of key algorithms. They help to ensure consistent correctness of such procedures, and provide a "higher level" of portability than MPI itself can provide. As such, libraries prevent each programmer from repeating the work of defining consistent data structures, data layouts, and methods that implement key algorithms (such as matrix operations). Since the best libraries come with several variations on parallel systems (different data layouts, different strategies depending on the size of the system or problem, or type of floating point), this too needs to be hidden from the user.

We refer the reader to [58] and [4] for further information on writing libraries in MPI, using the features described in this chapter.

6.1.1 Features Needed to Support Libraries

The key features needed to support the creation of robust parallel libraries are as follows:

- Safe communication space, that guarantees that libraries can communicate as they need to, without conflicting with communication extraneous to the library,
- Group scope for collective operations, that allow libraries to avoid unnecessarily synchronizing uninvolved processes (potentially running unrelated code),
- Abstract process naming to allow libraries to describe their communication in terms suitable to their own data structures and algorithms,
- The ability to "adorn" a set of communicating processes with additional user-defined attributes, such as extra collective operations. This mechanism should provide a means for the user or library writer effectively to extend a message-passing notation.

In addition, a unified mechanism or object is needed for conveniently denoting communication context, the group of communicating processes, to house abstract process naming, and to store adornments.

6.1.2 MPI's Support for Libraries

The corresponding concepts that MPI provides, specifically to support robust libraries, are as follows:

- **Contexts** of communication,
- Groups of processes,
- Virtual topologies,
- Attribute caching,
- Communicators.

Communicators (see [22, 56, 60]) 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 7 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 7 to provide this feature. Intra-communicators as described in this chapter do not have topologies.
- Attributes define the local information that the user or library has added to a communicator for later reference.

Advice to users. The practice in many communication libraries is that there is a unique, predefined communication universe that includes all processes available when the parallel program is initiated; the processes are assigned consecutive ranks. Participants in a point-to-point communication are identified by their rank; a collective communication (such as broadcast) always involves all processes. When using the World Model (Section 10.2), 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 an 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.

6.2 Basic Concepts

In this section, we turn to a more formal definition of the concepts introduced above.

6.2.1 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.)

6.2.2 Contexts

A context is a property of communicators (defined next) that allows partitioning of the communication space. A message sent in one context cannot be received in another context. Furthermore, where permitted, collective operations are independent of pending point-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.*)

6.2.3 Intra-Communicators

Intra-communicators bring together the concepts of group and context. To support implementation-specific optimizations, and application topologies (defined in the next chapter, Chapter 7), communicators may also "cache" additional information (see Section 6.7). MPI communication operations reference communicators to determine the scope and the "communication universe" in which a point-to-point or collective operation is to operate.

Each communicator contains a group of valid participants; this group always includes the local process. The source and destination of a message is identified by process rank within that group.

For collective communication, the intra-communicator specifies the set of processes that participate in the collective operation (and their order, when significant). Thus, the communicator restricts the "spatial" scope of communication, and provides machine-independent process addressing through ranks.

Intra-communicators are represented by opaque **intra-communicator objects**, and hence cannot be directly transferred from one process to another.

6.2.4 Predefined Intra-Communicators

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 10.3) 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

257

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

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
     6.3
            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
     6.3.1 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
     Fortran 2008 binding
48
```

TYPE(INTEC	o_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 ER GROUP, RANK, IERROR	3)	6 7 8 9	
	IP TRANSLATE RANKS(gro	up1, n, ranks1, group2, ranks2)	10 11	
	1		11	
IN	group1	group1 (handle)	13	
IN	n	number of ranks in ranks1 and ranks2 arrays (integer)	14	
IN	ranks1	array of zero or more valid ranks in group1	15	
IN	group2	group2 (handle)	16	
OUT	ranks2	array of corresponding ranks in group2,	17 18	
		MPI_UNDEFINED when no correspondence exists.	19	
			20	
C bindin			21	
int MPI_C	-	Group group1, int n, const int ranks1[],	22	
	MPI_Group group2, in	t ranksz[])	23	
Fortrail 2008 binding			24	
-	MPI_Group_translate_ranks(group1, n, ranks1, group2, ranks2, ierror) 2 ²⁵ TYPE(MPI_Group), INTENT(IN) :: group1, group2 2 ⁶			
	-		20	
	ER, INTENT(IN) :: n, ran ER, INTENT(OUT) :: ranks		28	
	ER, OPTIONAL, INTENT(OUT)		29	
			30	
Fortran h	3		31	
		N, RANKS1, GROUP2, RANKS2, IERROR) GROUP2, RANKS2(*), IERROR	32	
			33 34	
	-	nining the relative numbering of the same processes	35	
		ne knows the ranks of certain processes in the group to know their ranks in a subset of that group.	36	
		nput to MPI_GROUP_TRANSLATE_RANKS, which	37	
	PI_PROC_NULL as the translat		38	
			39	
			40	
	<pre>JP_COMPARE(group1, group2</pre>	,	41 42	
IN	group1	first group (handle)	43	
IN	group2	second group (handle)	44	
OUT	result	result (integer)	45	
			46	
	C binding 47			
int MPI_C	int MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result) 48			

1	Fortran 2	008 binding			
2	<pre>MPI_Group_compare(group1, group2, result, ierror)</pre>				
3	TYPE(MPI_Group), INTE	ENT(IN) :: group1, group2		
4	INTEG	ER, INTENT(OUT)	:: result		
5	INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
6	Fortron b	inding			
7	Fortran b				
8			, GROUP2, RESULT, IERROR)		
9	TNIEG	ER GRUUFI, GRUUF	P2, RESULT, IERROR		
10	MPI_IDENT	results if the group	o members and group order is exactly the same in both groups.		
11	This happe	ens for instance if g	<code>roup1</code> and <code>group2</code> are the same handle. MPI_SIMILAR results if		
12	the group r	members are the sam	me but the order is different. MPI_UNEQUAL results otherwise.		
13					
14	6.3.2 Gro	oup Constructors			
15		•			
16			to constructing groups. In the first approach, MPI procedures		
17	-		superset existing groups. These constructors construct new		
18			In the second approach, a group is created using a session		
19		-	set. This second approach is available when using the Sessions		
20			nes, these are local operations, and distinct groups may be		
21		-	a process may also define a group that does not include itself.		
22	Consistent definitions are required when groups are used as arguments in communicator-				
23	building functions. When using the World Model for initializing MPI, the base group, upon				
24	which all other groups are defined, is the group associated with the initial communicator MPI_COMM_WORLD (accessible through the function MPI_COMM_GROUP).				
25	MPI_COMM	I_WORLD (accessib	le through the function MPI_COMM_GROUP).		
26	Ratic	onale. In what f	ollows, there is no group duplication function analogous to		
27			ned later in this chapter. There is no need for a group dupli-		
28			reated, can have several references to it by making copies of		
29			ng constructors address the need for subsets and supersets of		
30		ing groups. (End o			
31	CABU	ing groups. (Ena o	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1		
32	Advid	ce to implementors	Each group constructor behaves as if it returned a new		
33		-	his new group is a copy of an existing group, then one can		
34 35	avoid	creating such new	v objects, using a reference-count mechanism. (End of advice		
	to im	nplementors.)			
36 37					
38					
39			`		
40	MPI_COM	M_GROUP(comm,	group)		
41	IN	comm	communicator (handle)		
42	OUT	group	group corresponding to comm (handle)		
43	001	8. • • P	Stoup conceptionang to comm (namate)		
44	C binding	r			
45	-	5	omm comm, MPI_Group *group)		
46		.			
47		008 binding			
48	MPI_Comm_	group(comm, grou	ıp, ierror)		

TYPE(MPI_Comm), INTENT(IN) :: comm			
TYPE(MPI_Group), INTENT(OUT) :: group			2
INTE	GER, OPTIONAL, INTENT(OUT	C) :: ierror	3
Fortran	binding		4
	_GROUP(COMM, GROUP, IERRO) (5
	GER COMM, GROUP, IERROR		6
			7
MPI_	COMM_GROUP returns in gr	oup a handle to the group of comm.	8 9
			9 10
MPI_GRC	UP_UNION(group1, group2, n	ewgroup)	10
IN	group1	first group (handle)	12
IN	group2	second group (handle)	13
		· · ·	14
OUT	newgroup	union group (handle)	15
			16
C bindir	•		17
int MPI_	Group_union(MPI_Group gro		18
	MPI_Group *newgroup)	19
Fortran	2008 binding		20 21
MPI_Grou	p_union(group1, group2, n	newgroup, ierror)	21
TYPE	(MPI_Group), INTENT(IN) :	: group1, group2	23
	(MPI_Group), INTENT(OUT)		24
INTEGER, OPTIONAL, INTENT(OUT) :: ierror			25
Fortran	binding		26
	P_UNION(GROUP1, GROUP2, N	IEWGROUP. IERROR)	27
	GER GROUP1, GROUP2, NEWGH		28
			29
			30
MPI_GRC	UP_INTERSECTION(group1,	group2, newgroup)	31
IN	group1	first group (handle)	32
			33 34
IN	group2	second group (handle)	34
OUT	newgroup	intersection group (handle)	36
			37
C bindir	ıg		38
int MPI_	-	coup group1, MPI_Group group2,	39
	MPI_Group *newgroup)	40
Fortran	2008 binding		41
	p_intersection(group1, gr	coup2, newgroup, ierror)	42
	(MPI_Group), INTENT(IN)		43
	(MPI_Group), INTENT(OUT)		44
	GER, OPTIONAL, INTENT(OUT		45
Fortran	binding		46
	P_INTERSECTION(GROUP1, GF		47
III 1_01100	(00011, 01		48

```
1
          INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
\mathbf{2}
3
4
      MPI_GROUP_DIFFERENCE(group1, group2, newgroup)
5
       IN
                 group1
                                               first group (handle)
6
       IN
                  group2
                                               second group (handle)
7
8
       OUT
                 newgroup
                                               difference group (handle)
9
10
      C binding
11
      int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
12
                     MPI_Group *newgroup)
13
      Fortran 2008 binding
14
     MPI_Group_difference(group1, group2, newgroup, ierror)
15
          TYPE(MPI_Group), INTENT(IN) :: group1, group2
16
          TYPE(MPI_Group), INTENT(OUT) :: newgroup
17
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
      Fortran binding
20
     MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)
21
          INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
22
23
      The set-like operations are defined as follows:
^{24}
      union All elements of the first group (group1), followed by all elements of second group
25
           (group2) not in the first group.
26
      intersect all elements of the first group that are also in the second group, ordered as in
27
28
           the first group.
29
      difference all elements of the first group that are not in the second group, ordered as in
30
           the first group.
^{31}
     Note that for these operations the order of processes in the output group is determined
32
      primarily by order in the first group (if possible) and then, if necessary, by order in the
33
34
      second group. Neither union nor intersection are commutative, but both are associative.
          The new group can be empty, that is, equal to MPI_GROUP_EMPTY.
35
36
37
      MPI_GROUP_INCL(group, n, ranks, newgroup)
38
39
       IN
                                               group (handle)
                  group
40
       IN
                                               number of elements in array ranks (and size of
                  n
41
                                               newgroup) (integer)
42
       IN
                                               ranks of processes in group to appear in newgroup
                  ranks
43
                                               (array of integers)
44
       OUT
                                               new group derived from above, in the order defined
                  newgroup
45
                                               by ranks (handle)
46
47
48
      C binding
```

Fortran 2008 binding

```
MPI_Group_incl(group, n, ranks, newgroup, ierror)
   TYPE(MPI_Group), INTENT(IN) :: group
   INTEGER, INTENT(IN) :: n, ranks(n)
   TYPE(MPI_Group), INTENT(OUT) :: newgroup
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
```

The function MPI_GROUP_INCL creates a group newgroup that consists of the n 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)

			22
IN	group	group (handle)	23
IN	n	number of elements in array ranks (integer)	24
IN	ranks	array of integer ranks of processes in group not to	25
		appear in newgroup	26
OUT	newgroup	new group derived from above, preserving the order	27
001	newgroup	defined by group (handle)	28
		dollind by group (handle)	29 30
C bindi	ng		30
		roup group, int n, const int ranks[],	32
Int mi.	MPI_Group *		33
	-	no.Prodb)	34
Fortran	2008 binding		35
		ranks, newgroup, ierror)	36
	E(MPI_Group), INT		37
	EGER, INTENT(IN)		38
		ENT(OUT) :: newgroup	39
TN.L.	EGER, UPTIONAL, I	NTENT(OUT) :: ierror	40
Fortran	binding		41
MPI_GROU	JP_EXCL(GROUP, N,	RANKS, NEWGROUP, IERROR)	42
INT	EGER GROUP, N, RAI	NKS(*), NEWGROUP, IERROR	43
The	function MPL CROU	P_EXCL creates a group of processes newgroup that is obtained	44
		F_LACE creates a group of processes newgroup that is obtained	45

The function MPI_GROUP_EXCL creates a group of processes newgroup that is obtained by deleting from group those processes with ranks ranks[0],..., ranks[n-1]. The ordering of processes in newgroup is identical to the ordering in group. Each of the n elements of ranks

264 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

```
1
      must be a valid rank in group and all elements must be distinct; otherwise, the program is
\mathbf{2}
      erroneous. If n = 0, then newgroup is identical to group.
3
4
      MPI_GROUP_RANGE_INCL(group, n, ranges, newgroup)
5
6
        IN
                                                  group (handle)
                   group
7
        IN
                                                  number of triplets in array ranges (integer)
                   n
8
        IN
                   ranges
                                                  a one-dimensional array of integer triplets, of the
9
                                                  form (first rank, last rank, stride) indicating ranks in
10
                                                  group of processes to be included in newgroup
11
                                                  new group derived from above, in the order defined
12
        OUT
                   newgroup
13
                                                  by ranges (handle)
14
15
      C binding
16
      int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],
17
                      MPI_Group *newgroup)
18
      Fortran 2008 binding
19
      MPI_Group_range_incl(group, n, ranges, newgroup, ierror)
20
           TYPE(MPI_Group), INTENT(IN) :: group
21
           INTEGER, INTENT(IN) :: n, ranges(3, n)
22
           TYPE(MPI_Group), INTENT(OUT) :: newgroup
23
           INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{24}
25
      Fortran binding
26
      MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)
27
           INTEGER GROUP, N, RANGES(3, *), NEWGROUP, IERROR
28
      If ranges consists of the triplets
29
30
            (first_1, last_1, stride_1), \ldots, (first_n, last_n, stride_n)
^{31}
32
      then newgroup consists of the sequence of processes in group with ranks
33
           first_1, first_1 + stride_1, \dots, first_1 + \left| \frac{last_1 - first_1}{stride_1} \right| stride_1, \dots,
34
35
           first_n, first_n + stride_n, \dots, first_n + \left\lfloor \frac{last_n - first_n}{stride_n} \right\rfloor stride_n.
36
37
38
39
           Each computed rank must be a valid rank in group and all computed ranks must be
40
      distinct, or else the program is erroneous. Note that we may have first_i > last_i, and stride_i
^{41}
      may be negative, but cannot be zero.
42
           The functionality of this routine is specified to be equivalent to expanding the array
43
      of ranges to an array of the included ranks and passing the resulting array of ranks and
^{44}
      other arguments to MPI_GROUP_INCL. A call to MPI_GROUP_INCL is equivalent to a call
45
      to MPI_GROUP_RANGE_INCL with each rank i in ranks replaced by the triplet (i,i,1) in the
46
      argument ranges.
47
```

MPI_GROUP_RANGE_EXCL(group, n, ranges, newgroup)			
IN	group	group (handle)	
IN	n	number of triplets in array ranges (integer)	
IN	ranges	a one-dimensional array of integer triplets, of the form (first rank, last rank, stride) indicating ranks in group of processes to be excluded from the output group newgroup (array of integers)	
OUT	newgroup	new group derived from above, preserving the order in group (handle)	

MPI_GROUP_RANGE_EXCL(group, n, ranges, newgroup)

C binding

Fortran 2008 binding

MPI_Group_range_excl(group, n, ranges, newgroup, ierror)
 TYPE(MPI_Group), INTENT(IN) :: group
 INTEGER, INTENT(IN) :: n, ranges(3, n)
 TYPE(MPI_Group), INTENT(OUT) :: newgroup
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

```
MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR)
INTEGER GROUP, N, RANGES(3, *), NEWGROUP, IERROR
```

Each computed rank must be a valid rank in **group** and all computed ranks must be distinct, or else the program is erroneous.

The functionality of this routine is specified to be equivalent to expanding the array of ranges to an array of the excluded ranks and passing the resulting array of ranks and other arguments to MPI_GROUP_EXCL. A call to MPI_GROUP_EXCL is equivalent to a call to MPI_GROUP_RANGE_EXCL with each rank i in ranks replaced by the triplet (i,i,1) in the argument ranges.

Advice to users. The range operations do not explicitly enumerate ranks, and therefore are more scalable if implemented efficiently. Hence, we recommend MPI programmers to use them whenenever possible, as high-quality implementations will take advantage of this fact. (*End of advice to users.*)

Advice to implementors. The range operations should be implemented, if possible, without enumerating the group members, in order to obtain better scalability (time and space). (End of advice to implementors.)

 24

```
266 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING
```

```
1
     MPI_GROUP_FROM_SESSION_PSET(session, pset_name, newgroup)
2
       IN
                session
                                            session (handle)
3
       IN
                                            name of process set to use to create the new group
                 pset_name
4
                                            (string)
5
6
       OUT
                                            new group derived from supplied session and process
                newgroup
7
                                            set (handle)
8
9
     C binding
10
     int MPI_Group_from_session_pset(MPI_Session session, const char *pset_name,
11
                    MPI_Group *newgroup)
12
     Fortran 2008 binding
13
     MPI_Group_from_session_pset(session, pset_name, newgroup, ierror)
14
         TYPE(MPI_Session), INTENT(IN) :: session
15
         CHARACTER(LEN=*), INTENT(IN) :: pset_name
16
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
     Fortran binding
20
     MPI_GROUP_FROM_SESSION_PSET(SESSION, PSET_NAME, NEWGROUP, IERROR)
21
          INTEGER SESSION, NEWGROUP, IERROR
22
         CHARACTER*(*) PSET_NAME
23
         The function MPI_GROUP_FROM_SESSION_PSET creates a group newgroup using the
^{24}
     provided session handle and process set. The process set name must be one returned from
25
     an invocation of MPI_SESSION_GET_NTH_PSET using the supplied session handle. If the
26
     pset_name does not exist, MPI_GROUP_NULL will be returned in the newgroup argument.
27
     As with other group constructors, MPI_GROUP_FROM_SESSION_PSET is a local function.
28
     See section 10.3 for more information on sessions and process sets.
29
30
^{31}
     6.3.3 Group Destructors
32
33
34
     MPI_GROUP_FREE(group)
35
       INOUT
                group
                                            group (handle)
36
37
     C binding
38
     int MPI_Group_free(MPI_Group *group)
39
40
     Fortran 2008 binding
41
     MPI_Group_free(group, ierror)
42
         TYPE(MPI_Group), INTENT(INOUT) :: group
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
     Fortran binding
45
     MPI_GROUP_FREE(GROUP, IERROR)
46
         INTEGER GROUP, IERROR
47
48
```

This operation marks a group object for deallocation. The handle group is set to MPI_GROUP_NULL by the call. Any on-going operation using this group will complete normally.

Advice to implementors. One can keep a reference count that is incremented for each call to MPI_COMM_GROUP, MPI_COMM_CREATE, MPI_COMM_DUP, and MPI_COMM_IDUP, and decremented for each call to MPI_GROUP_FREE or MPI_COMM_FREE; the group object is ultimately deallocated when the reference count drops to zero. (*End of advice to implementors.*)

6.4 Communicator Management

This section describes the manipulation of communicators in MPI. Operations that access communicators are local and their execution does not require interprocess communication. Operations that create communicators are collective and may require interprocess communication.

Advice to implementors. High-quality implementations should amortize the overheads associated with the creation of communicators (for the same group, or subsets thereof) over several calls, by allocating multiple contexts with one collective communication. (*End of advice to implementors.*)

6.4.1	Communicator Accessors		23
Tho fo	blowing are all local operation		24
The ic	mowing are an local operation	5.	25
			26
MPI_C	COMM_SIZE(comm, size)		27 28
IN	comm	communicator (handle)	28 29
Ουτ	size	number of processes in the group of comm (integer)	30
001	SIZC	number of processes in the group of comm (megor)	31
C bin	ding		32
	PI_Comm_size(MPI_Comm comm	m int trizo)	33
IIIC M	F1_COMM_SIZe(MF1_COMM COM	I, IIIt ≁SIZe)	34
Fortr	an 2008 binding		35
MPI_C	omm_size(comm, size, ierro	or)	36
Т	YPE(MPI_Comm), INTENT(IN)	:: comm	37
I	NTEGER, INTENT(OUT) :: si:	ze	38
I	NTEGER, OPTIONAL, INTENT()	DUT) :: ierror	39
Fonte	on hinding		40
	an binding		41
	OMM_SIZE(COMM, SIZE, IERR		42
T	NTEGER COMM, SIZE, IERROR		43
			44
-	Rationale. This function is e	quivalent to accessing the communicator's group with	45

Rationale. This function is equivalent to accessing the communicator's group with ⁴⁵ MPI_COMM_GROUP (see above), computing the size using MPI_GROUP_SIZE, and ⁴⁶ then freeing the temporary group via MPI_GROUP_FREE. However, this function is ⁴⁷ so commonly used that this shortcut was introduced. (*End of rationale.*) ⁴⁸

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\overline{7}$

```
1
           Advice to users.
                               This function indicates the number of processes involved in a
2
           communicator. For MPI_COMM_WORLD, it indicates the total number of processes
3
           available unless the number of processes has been changed by using the functions
4
           described in Chapter 10; note that the number of processes in MPI_COMM_WORLD
5
           does not change during the life of an MPI program.
6
           This call is often used with the next call to determine the amount of concurrency
7
           available for a specific library or program. The following call, MPI_COMM_RANK
8
           indicates the rank of the process that calls it in the range from 0, \ldots, size-1, where
9
           size is the return value of MPI_COMM_SIZE.(End of advice to users.)
10
11
12
     MPI_COMM_RANK(comm, rank)
13
14
       IN
                                              communicator (handle)
                 comm
15
                                              rank of the calling process in group of comm (integer)
       OUT
                 rank
16
17
     C binding
18
     int MPI_Comm_rank(MPI_Comm comm, int *rank)
19
20
     Fortran 2008 binding
21
     MPI_Comm_rank(comm, rank, ierror)
22
          TYPE(MPI_Comm), INTENT(IN) :: comm
23
          INTEGER, INTENT(OUT) :: rank
24
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26
     Fortran binding
     MPI_COMM_RANK(COMM, RANK, IERROR)
27
          INTEGER COMM, RANK, IERROR
28
29
30
           Rationale.
                       This function is equivalent to accessing the communicator's group with
31
           MPI_COMM_GROUP (see above), computing the rank using MPI_GROUP_RANK,
32
           and then freeing the temporary group via MPI_GROUP_FREE. However, this function
33
           is so commonly used that this shortcut was introduced. (End of rationale.)
34
35
           Advice to users. This function gives the rank of the process in the particular commu-
36
           nicator's group. It is useful, as noted above, in conjunction with MPI_COMM_SIZE.
37
           Many programs will be written with the supervisor/executor or manager/worker
38
           model, where one process (such as the rank-zero process) will play a supervisory
39
           role, and the other processes will serve as compute nodes. In this framework, the
40
           two preceding calls are useful for determining the roles of the various processes of a
41
           communicator. (End of advice to users.)
42
43
44
45
46
47
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.

6.4.2 Communicator Constructors

The following are collective functions that are invoked by all processes in the group or groups associated with comm, with the exception of MPI_COMM_CREATE_GROUP, 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 and MPI_COMM_SPLIT can be used to create both intracommunicators and intercommunicators; MPI_COMM_CREATE_GROUP, MPI_COMM_CREATE_FROM_GROUP, and MPI_INTERCOMM_MERGE (see Section 6.6.2) can be used to create intracommunicators;

```
1
     MPI_INTERCOMM_CREATE and MPI_INTERCOMM_CREATE_FROM_GROUPS (see Sec-
\mathbf{2}
     tion 6.6.2) can be used to create intercommunicators.
3
          An intracommunicator involves a single group while an intercommunicator involves
4
     two groups. Where the following discussions address intercommunicator semantics, the
\mathbf{5}
     two groups in an intercommunicator are called the left and right groups. A process in an
6
     intercommunicator is a member of either the left or the right group. From the point of view
\overline{7}
     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 intercommunicator 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 6.7.1).
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 7). 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.)
```

		(comm, info, newcomm)
IN	comm	communicator (handle)
IN	info	info object (handle)
OUT	newcomm	copy of comm (handle)
bindi:	ng	
nt MPI	_Comm_dup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm)
ortran	2008 binding	
	•	, info, newcomm, ierror)
TYPI	E(MPI_Comm), INTENT(IN) :: comm
	E(MPI_Info), INTENT(
	E(MPI_Comm), INTENT(
TN.1,F	EGER, OPTIONAL, INTE	NI(UUI) :: lerror
	binding	
		I, INFO, NEWCOMM, IERROR)
TN.LI	EGER COMM, INFO, NEW	CUMM, IERRUR
		NFO behaves exactly as MPI_COMM_DUP except that the
ints pro	vided by the argument in	nfo are associated with the output communicator $newcomm.$
Ra	tionale It is expected t	that some hints will only be valid at communicator creation
	-	reasons, most communicator creation calls do not provide
		y associate info hints with a duplicate of any communicator
at o	creation time through a	call to MPI_COMM_DUP_WITH_INFO. (<i>End of rationale.</i>)
IPI_COI	MM_IDUP(comm, newco	omm, request)
IN	comm	communicator (handle)
OUT	newcomm	copy of comm (handle)
OUT		
001	request	communication request (handle)
! bindi	no	
bindin nt MPI		comm. MPI Comm *newcomm. MPI Request *request)
nt MPI <u></u>	_Comm_idup(MPI_Comm	<pre>comm, MPI_Comm *newcomm, MPI_Request *request)</pre>
nt MPI <u></u> ortran	_Comm_idup(MPI_Comm 2008 binding	
nt MPI <u></u> Ortran PI_Comr	_Comm_idup(MPI_Comm 2008 binding n_idup(comm, newcomm	, request, ierror)
nt MPI <u></u> ortran PI_Comr TYPI	_Comm_idup(MPI_Comm 2008 binding n_idup(comm, newcomm E(MPI_Comm), INTENT(, request, ierror)
nt MPI <u></u> ortran PI_Comr TYPI TYPI	_Comm_idup(MPI_Comm 2008 binding n_idup(comm, newcomm E(MPI_Comm), INTENT(, request, ierror) IN) :: comm OUT), ASYNCHRONOUS :: newcomm
nt MPI_ ortran PI_Comr TYPI TYPI TYPI	_Comm_idup(MPI_Comm 2008 binding n_idup(comm, newcomm E(MPI_Comm), INTENT(E(MPI_Comm), INTENT(, request, ierror) IN) :: comm OUT), ASYNCHRONOUS :: newcomm NT(OUT) :: request
nt MPI ortran PI_Comr TYPI TYPI TYPI INTI	_Comm_idup(MPI_Comm 2008 binding n_idup(comm, newcomm E(MPI_Comm), INTENT(E(MPI_Comm), INTENT(E(MPI_Request), INTE EGER, OPTIONAL, INTE	, request, ierror) IN) :: comm OUT), ASYNCHRONOUS :: newcomm NT(OUT) :: request
nt MPI ortran PI_Comr TYPI TYPI TYPI INTI	_Comm_idup(MPI_Comm 2008 binding n_idup(comm, newcomm E(MPI_Comm), INTENT(E(MPI_Comm), INTENT(E(MPI_Request), INTE	, request, ierror) IN) :: comm OUT), ASYNCHRONOUS :: newcomm NT(OUT) :: request NT(OUT) :: ierror
nt MPI ortran PI_Comr TYPI TYPI TYPI INTI ortran PI_COM	_Comm_idup(MPI_Comm 2008 binding n_idup(comm, newcomm E(MPI_Comm), INTENT(E(MPI_Comm), INTENT(E(MPI_Request), INTE EGER, OPTIONAL, INTE binding	, request, ierror) IN) :: comm OUT), ASYNCHRONOUS :: newcomm NT(OUT) :: request NT(OUT) :: ierror , REQUEST, IERROR)

1			ing variant of MPI_COMM_DUP. With the exception	
2	of its nonblocking behavior, the semantics of MPI_COMM_IDUP are as if MPI_COMM_DUP			
3	was executed at the time that MPI_COMM_IDUP is called. For example, attributes changed			
4			copied to the new communicator. All restrictions and	
5	-	_	we operations (see Section 5.12) apply to	
6 7		IM_IDUP and the returned	*	
8		before the MPI_COMM_IDU	nicator newcomm as an input argument to other MPI	
9	Tunctions i		or operation completes.	
10				
11	MPI_COM	IM_IDUP_WITH_INFO(com	m, info, newcomm, request)	
12	IN	comm	communicator (handle)	
13 14	IN	info	info object (handle)	
15	OUT	newcomm	copy of comm (handle)	
16	OUT	request	communication request (handle)	
17				
18	C binding	g		
19	int MPI_C	-	_Comm comm, MPI_Info info,	
20 21		MPI_Comm *newcomm,	MPI_Request *request)	
22	Fortran 2	2008 binding		
23	MPI_Comm_	_idup_with_info(comm, i	nfo, newcomm, request, ierror)	
24	TYPE(MPI_Comm), INTENT(IN) :: comm			
25	TYPE(MPI_Info), INTENT(IN) :: info			
26	TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS :: newcomm TYPE(MPI_Request), INTENT(OUT) :: request			
27		-		
28	INTEG	GER, OPTIONAL, INTENT(O	UT) :: ierror	
29	Fortran k	oinding		
30	MPI_COMM_	_IDUP_WITH_INFO(COMM, I	NFO, NEWCOMM, REQUEST, IERROR)	
31 32	INTEG	GER COMM, INFO, NEWCOMM	, REQUEST, IERROR	
33	MPL (COMM IDUP WITH INFO	is a nonblocking variant of	
34			h the exception of its nonblocking behavior, the se-	
35			_INFO are as if MPI_COMM_DUP_WITH_INFO was	
36	executed a	t the time that MPI_COMM	_IDUP_WITH_INFO is called. For example, attributes	
37	or info hin	its changed after MPI_CON	IM_IDUP_WITH_INFO will not be copied to the new	
38			ssumptions for nonblocking collective operations (see	
39		, = = -	DUP_WITH_INFO and the returned request.	
40			nicator newcomm as an input argument to other MPI	
41	tunctions h	before the MPI_COMM_IDU	JP_WITH_INFO operation completes.	
42	Rati	onale. The MPI_COMM_	IDUP and MPI_COMM_IDUP_WITH_INFO functions	
43 44			t of purely nonblocking libraries (see [37]). (End of	
44		onale.)		
46		1		
47				
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 intracommunicator, this function returns a new communicator newcomm with communication group defined by the group argument. No cached information propagates from comm to newcomm. Each process must call MPI_COMM_CREATE with a group argument that is a subgroup of the group associated with comm; this could be MPI_GROUP_EMPTY. The processes may specify different values for the group argument. If a process calls with a non-empty group then all processes in that group must call the function with the same group as argument, that is the same processes in the same order. Otherwise, the call is erroneous. This implies that the set of groups specified across the processes must be disjoint. If the calling process is a member of the group given as group argument, then newcomm is a communicator with group as its associated group. In the case that a process calls with a group to which it does not belong, e.g., MPI_GROUP_EMPTY, then MPI_COMM_NULL is returned as newcomm. The function is collective and must be called by all processes in the group of comm.

Rationale. The interface supports the original mechanism from MPI-1.1, which required the same group in all processes of comm. It was extended in MPI-2.2 to allow the use of disjoint subgroups in order to allow implementations to eliminate unnecessary communication that MPI_COMM_SPLIT would incur when the user already knows the membership of the disjoint subgroups. (*End of rationale.*)

Rationale. The requirement that the entire group of comm participate in the call stems from the following considerations:

- It allows the implementation to layer MPI_COMM_CREATE on top of regular collective communications.
- It provides additional safety, in particular in the case where partially overlapping groups are used to create new communicators.
- It permits implementations to sometimes avoid communication related to context creation.

Unofficial Draft for Comment Only

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

274 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

```
1
           (End of rationale.)
2
3
           Advice to users. MPI_COMM_CREATE provides a means to subset a group of pro-
           cesses for the purpose of separate MIMD computation, with separate communication
4
           space. newcomm, which emerges from MPI_COMM_CREATE, can be used in subse-
5
           quent calls to MPI_COMM_CREATE (or other communicator constructors) to further
6
           subdivide a computation into parallel sub-computations. A more general service is
7
           provided by MPI_COMM_SPLIT, below. (End of advice to users.)
8
9
           Advice to implementors. When calling MPI_COMM_DUP, all processes call with the
10
           same group (the group associated with the communicator). When calling
11
           MPI_COMM_CREATE, the processes provide the same group or disjoint subgroups.
12
           For both calls, it is theoretically possible to agree on a group-wide unique context
13
           with no communication. However, local execution of these functions requires use
14
           of a larger context name space and reduces error checking. Implementations may
15
           strike various compromises between these conflicting goals, such as bulk allocation of
16
           multiple contexts in one collective operation.
17
18
           Important: If new communicators are created without synchronizing the processes
19
           involved then the communication system must be able to cope with messages arriving
20
           in a context that has not yet been allocated at the receiving process. (End of advice
21
           to implementors.)
22
     If comm is an intercommunicator, then the output communicator is also an intercommun-
23
     icator where the local group consists only of those processes contained in group (see Fig-
^{24}
     ure 6.1). The group argument should only contain those processes in the local group of
25
26
     the input intercommunicator that are to be a part of newcomm. All processes in the same
     local group of comm must specify the same value for group, i.e., the same members in the
27
     same order. If either group does not specify at least one process in the local group of the
28
     intercommunicator, or if the calling process is not included in the group, MPI_COMM_NULL
29
     is returned.
30
^{31}
           Rationale. In the case where either the left or right group is empty, a null communi-
32
           cator is returned instead of an intercommunicator with MPI_GROUP_EMPTY because
33
         the side with the empty group must return MPI_COMM_NULL. (End of rationale.)
34
35
36
     Example 6.1 The following example illustrates how the first node in the left side of an
37
     intercommunicator could be joined with all members on the right side of an intercommun-
38
     icator to form a new intercommunicator.
39
40
              MPI_Comm inter_comm, new_inter_comm;
41
              MPI_Group local_group, group;
42
               int
                          rank = 0; /* rank on left side to include in
43
                                         new inter-comm */
44
45
               /* Construct the original intercommunicator: "inter_comm" */
46
47
48
               /* Construct the group of processes to be in new
```

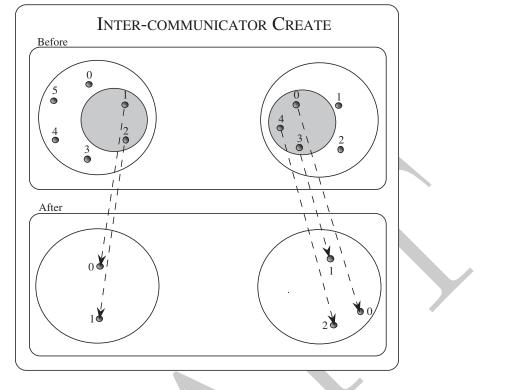


Figure 6.1: Intercommunicator creation using MPI_COMM_CREATE extended to intercommunicators. The input groups are those in the grey circle.

```
intercommunicator */
                                                                                         26
        if (/* I'm on the left side of the intercommunicator */) {
                                                                                         27
           MPI_Comm_group(inter_comm, &local_group);
                                                                                         28
           MPI_Group_incl(local_group, 1, &rank, &group);
                                                                                         29
           MPI_Group_free(&local_group);
                                                                                         30
        }
                                                                                         ^{31}
         else
                                                                                         32
           MPI_Comm_group(inter_comm, &group);
                                                                                         33
                                                                                         34
        MPI_Comm_create(inter_comm, group, &new_inter_comm);
                                                                                         35
        MPI_Group_free(&group);
                                                                                         36
                                                                                         37
                                                                                         38
MPI_COMM_CREATE_GROUP(comm, group, tag, newcomm)
                                                                                         39
                                                                                         40
 IN
           comm
                                       intracommunicator (handle)
                                                                                         41
 IN
                                       group, which is a subset of the group of comm
           group
                                                                                         42
                                       (handle)
                                                                                         43
                                                                                         44
 IN
                                       tag (integer)
           tag
                                                                                         45
 OUT
           newcomm
                                       new communicator (handle)
                                                                                         46
                                                                                         47
```

1

 $\mathbf{2}$

3 4

5 6

7

8

9 10 11

23

 $\frac{24}{25}$

1 2	int MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag, MPI_Comm *newcomm)
3 4 5 6 7 8 9 10	<pre>Fortran 2008 binding MPI_Comm_create_group(comm, group, tag, newcomm, ierror) TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Group), INTENT(IN) :: group INTEGER, INTENT(IN) :: tag TYPE(MPI_Comm), INTENT(OUT) :: newcomm INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>
11 12 13	Fortran binding MPI_COMM_CREATE_GROUP(COMM, GROUP, TAG, NEWCOMM, IERROR) INTEGER COMM, GROUP, TAG, NEWCOMM, IERROR
14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29	MPI_COMM_CREATE_GROUP is similar to MPI_COMM_CREATE; however, MPI_COMM_CREATE must be called by all processes in the group of comm, whereas MPI_COMM_CREATE_GROUP must be called by all processes in group, which is a subgroup of the group of comm. In addition, MPI_COMM_CREATE_GROUP requires that comm is an intracommunicator. MPI_COMM_CREATE_GROUP returns a new intracommunicator, newcomm, for which the group argument defines the communication group. No cached infor- mation propagates from comm to newcomm. Each process must provide a group argument that is a subgroup of the group associated with comm; this could be MPI_GROUP_EMPTY. If a non-empty group is specified, then all processes in that group must call the function, and each of these processes must provide the same arguments, including a group that contains the same members with the same ordering. Otherwise the call is erroneous. If the calling process is a member of the group given as the group argument, then newcomm is a commu- nicator with group as its associated group. If the calling process is not a member of group, e.g., group is MPI_GROUP_EMPTY, then the call is a local operation and MPI_COMM_NULL is returned as newcomm.
30 31 32 33 34 35 36 37	Rationale. Functionality similar to MPI_COMM_CREATE_GROUP can be imple- mented through repeated MPI_INTERCOMM_CREATE and MPI_INTERCOMM_MERGE calls that start with the MPI_COMM_SELF communicators at each process in group and build up an intracommunicator with group group [17]. Such an algorithm requires the creation of many intermediate communicators; MPI_COMM_CREATE_GROUP can provide a more efficient implementation that avoids this overhead. (<i>End of rationale.</i>)
38 39 40 41 42	Advice to users. An intercommunicator can be created collectively over processes in the union of the local and remote groups by creating the local communicator using MPI_COMM_CREATE_GROUP and using that communicator as the local communicator argument to MPI_INTERCOMM_CREATE. (<i>End of advice to users.</i>)
43 44 45 46 47 48	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

MPI_Comm_split(comm, color, key, newcomm, ierror)
 TYPE(MPI_Comm), INTENT(IN) :: comm
 INTEGER, INTENT(IN) :: color, key
 TYPE(MPI_Comm), INTENT(OUT) :: newcomm
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

MPI_	_COMM_S	PLIT(CON	1M, COL	OR, K	EY, N	EWCOM	M, IH	ERROR)
	INTEGE	R COMM,	COLOR,	KEY,	NEWC	OMM.	IERRO)r

This function partitions the group associated with comm into disjoint subgroups, one for each value of color. Each subgroup contains all processes of the same color. Within each subgroup, the processes are ranked in the order defined by the value of the argument key, with ties broken according to their rank in the old group. A new communicator is created for each subgroup and returned in newcomm. A process may supply the color value MPI_UNDEFINED, in which case newcomm returns MPI_COMM_NULL. This is a collective call, but each process is permitted to provide different values for color and key.

With an intracommunicator comm, a call to MPI_COMM_CREATE(comm, group, newcomm) is equivalent to a call to MPI_COMM_SPLIT(comm, color, key, newcomm), where processes that are members of their group argument provide color = number of the group (based on a unique numbering of all disjoint groups) and key = rank in group, and all processes that are not members of their group argument provide color = MPI_UNDEFINED.

The value of color must be non-negative or MPI_UNDEFINED.

This is an extremely powerful mechanism for dividing a single Advice to users. 41 communicating group of processes into k subgroups, with k chosen implicitly by the 42user (by the number of colors asserted over all the processes). Each resulting com-43 municator will be non-overlapping. Such a division could be useful for defining a 44 hierarchy of computations, such as for multigrid, or linear algebra. For intracommu-45nicators, MPI_COMM_SPLIT provides similar capability as MPI_COMM_CREATE to 46 split a communicating group into disjoint subgroups. MPI_COMM_SPLIT is useful 47when some processes do not have complete information of the other members in their 48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4 5 6

15

16

17

18

19

20

21

22 23

24 25 26

27

28

29

30

 31

32

33

34

35

36

37

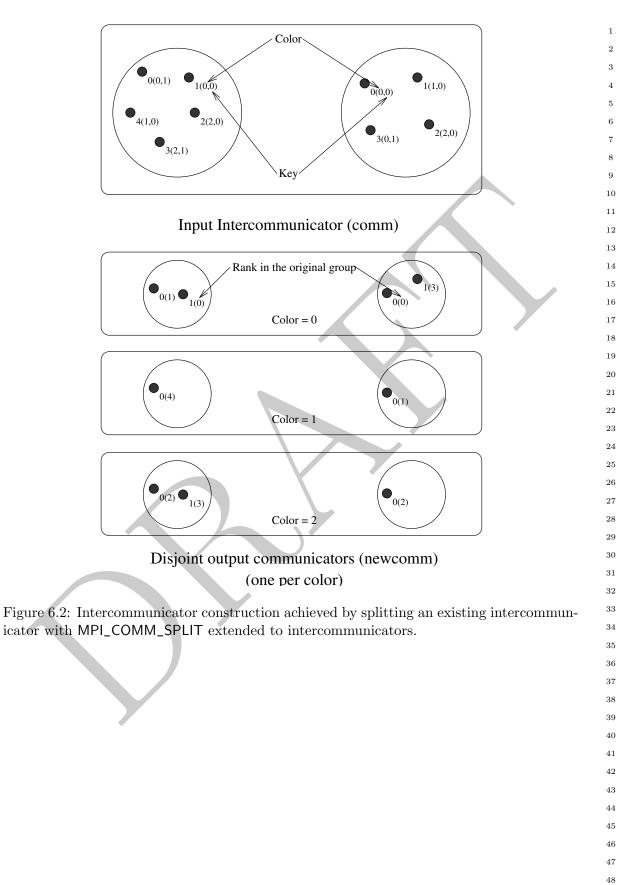
38

1 group, but all processes know (the color of) the group to which they belong. In this 2 case, the MPI implementation discovers the other group members via communication. 3 MPI_COMM_CREATE is useful when all processes have complete information of the 4 members of their group. In this case, MPI can avoid the extra communication required 5to discover group membership. MPI_COMM_CREATE_GROUP is useful when all pro-6 cesses in a given group have complete information of the members of their group and 7 synchronization with processes outside the group can be avoided. 8 Multiple calls to MPI_COMM_SPLIT can be used to overcome the requirement that 9 any call have no overlap of the resulting communicators (each process is of only one 10 color per call). In this way, multiple overlapping communication structures can be 11 created. Creative use of the color and key in such splitting operations is encouraged. 12Note that, for a fixed color, the keys need not be unique. It is MPI_COMM_SPLIT's 13 responsibility to sort processes in ascending order according to this key, and to break 14ties in a consistent way. If all the keys are specified in the same way, then all the 15processes in a given color will have the relative rank order as they did in their parent 16group. 17 18 Essentially, making the key value zero for all processes of a given color means that one 19 does not really care about the rank-order of the processes in the new communicator. (End of advice to users.) 2021*Rationale.* color is restricted to be non-negative, so as not to confict with the value 22 assigned to MPI_UNDEFINED. (End of rationale.) 23 24 The result of MPI_COMM_SPLIT on an intercommunicator is that those processes on the 25left with the same color as those processes on the right combine to create a new intercom-26municator. The key argument describes the relative rank of processes on each side of the 27intercommunicator (see Figure 6.2). For those colors that are specified only on one side of 28 the intercommunicator, MPI_COMM_NULL is returned. MPI_COMM_NULL is also returned 29 to those processes that specify MPI_UNDEFINED as the color. 30 31Advice to users. For intercommunicators, MPI_COMM_SPLIT is more general than 32 MPI_COMM_CREATE. A single call to MPI_COMM_SPLIT can create a set of disjoint 33 intercommunicators, while a call to MPI_COMM_CREATE creates only one. (End of 34 advice to users.) 3536 **Example 6.2** (Parallel client-server model). The following client code illustrates how clients 37 on the left side of an intercommunicator could be assigned to a single server from a pool of 38 servers on the right side of an intercommunicator. 39 40 /* Client code */ 41 MPI_Comm multiple_server_comm; 42MPI_Comm single_server_comm; 43 int color, rank, num_servers; 4445/* Create intercommunicator with clients and servers: 46multiple_server_comm */

47

48

. . .



```
1
              /* Find out the number of servers available */
2
              MPI_Comm_remote_size(multiple_server_comm, &num_servers);
3
4
              /* Determine my color */
5
              MPI_Comm_rank(multiple_server_comm, &rank);
6
              color = rank % num_servers;
7
8
              /* Split the intercommunicator */
9
              MPI_Comm_split(multiple_server_comm, color, rank,
10
                              &single_server_comm);
11
     The following is the corresponding server code:
12
13
              /* Server code */
14
              MPI_Comm multiple_client_comm;
15
              MPI_Comm single_server_comm;
16
              int
                         rank:
17
18
              /* Create intercommunicator with clients and servers:
19
                 multiple_client_comm */
20
              . . .
21
22
              /* Split the intercommunicator for a single server per group
23
                 of clients */
24
              MPI_Comm_rank(multiple_client_comm, &rank);
25
              MPI_Comm_split(multiple_client_comm, rank, 0,
26
                              &single_server_comm);
27
28
29
     MPI_COMM_SPLIT_TYPE(comm, split_type, key, info, newcomm)
30
^{31}
       IN
                                            communicator (handle)
                comm
32
       IN
                split_type
                                            type of processes to be grouped together (integer)
33
       IN
                                            control of rank assignment (integer)
                key
34
35
       INOUT
                info
                                           info argument (handle)
36
       OUT
                newcomm
                                            new communicator (handle)
37
38
     C binding
39
     int MPI_Comm_split_type(MPI_Comm comm, int split_type, int key,
40
                    MPI_Info info, MPI_Comm *newcomm)
41
42
     Fortran 2008 binding
43
     MPI_Comm_split_type(comm, split_type, key, info, newcomm, ierror)
44
         TYPE(MPI_Comm), INTENT(IN) :: comm
45
         INTEGER, INTENT(IN) :: split_type, key
46
         TYPE(MPI_Info), INTENT(IN) :: info
47
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
48
         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 process.

For split_type, the following values are defined by MPI:

MPI_COMM_TYPE_SHARED — all MPI processes in newcomm can create a shared memory segment (e.g., with a successful call to MPI_WIN_ALLOCATE_SHARED). This segment can subsequently be used for load/store accesses by all MPI processes in newcomm.

Advice to users. Since the location of some of the MPI processes may change during the application execution, the communicators created with the value MPI_COMM_TYPE_SHARED before this change may not reflect an actual ability to share memory between MPI processes after this change. (*End of advice to users.*)

MPI_COMM_TYPE_HW_GUIDED — this value specifies that the communicator comm is split according to a hardware resource type (for example a computing core or an L3 cache) specified by the mpi_hw_resource_type info key. Each output communicator newcomm corresponds to a single instance of the specified hardware resource type. The MPI processes in the group associated with the output communicator newcomm utilize that specific hardware resource type instance, and no other instance of the same hardware resource type.

If an MPI process does not meet the above criteria, then MPI_COMM_NULL is returned in newcomm for such process.

MPI_COMM_NULL is also returned in newcomm in the following cases:

- No info key is provided.
- The info handle does not include the key mpi_hw_resource_type.
- The MPI implementation neither recognizes nor supports the info key mpi_hw_resource_type.
- The MPI implementation does not recognize the value associated with the info key mpi_hw_resource_type.

The MPI implementation will return in the group of the output communicator **newcomm** the largest subset of MPI processes that match the splitting criterion.

The processes in the group associated with newcomm are ranked in the order defined ⁴⁶ by the value of the argument key with ties broken according to their rank in the group ⁴⁷ associated with comm.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

282 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

1	Advice to users. The set of hardware resources that an MPI process is able to					
2	utilize may change during the application execution (e.g., because of the reloca-					
3	tion of an MPI process), in which case the communicators created with the value MPI_COMM_TYPE_HW_GUIDED before this change may not reflect the utiliza-					
4						
5	tion of hardware resources of such process at any time after the communicator					
6	creation. (End of advice to users.)					
7						
8	The user explicitly constrains with the info argument the splitting of the input com-					
9 10	municator comm. To this end, the info key mpi_hw_resource_type is reserved and its associated value is an implementation-defined string designating the type of the re-					
10	quested hardware resource (e.g., "NUMANode", "Package" or "L3Cache").					
12						
13	The value mpi_shared_memory is reserved and its use is equivalent to using					
14	MPI_COMM_TYPE_SHARED for the split_type parameter.					
15	<i>Rationale.</i> The value mpi_shared_memory is defined in order to ensure consistency					
16	between the use of MPI_COMM_TYPE_SHARED and the use of					
17	MPI_COMM_TYPE_HW_GUIDED. (End of rationale.)					
18						
19	All MPI processes must provide the same value for the info key mpi_hw_resource_type.					
20						
21	Example 6.3 Splitting MPI_COMM_WORLD into NUMANode subcommunicators.					
22	NDI Info info.					
23	MPI_Info info; MPI_Comm hwcomm;					
24	int rank;					
25	int faik,					
26	<pre>MPI_Comm_rank(MPI_COMM_WORLD,&rank);</pre>					
27	MPI_Info_create(&info);					
28 29	<pre>MPI_Info_set(info, "mpi_hw_resource_type", "NUMANode");</pre>					
30	MPI_Comm_split_type(MPI_COMM_WORLD,					
31	MPI_COMM_TYPE_HW_GUIDED,					
32	<pre>rank,info,&hwcomm);</pre>					
33						
34						
35	MPI_COMM_TYPE_HW_UNGUIDED — the group of MPI processes associated with newcomm					
36	must be a <i>strict</i> subset of the group associated with comm and each					
37	newcomm corresponds to a single instance of a hardware resource type (for example					
38	a computing core or an L3 cache).					
39	All MPI processes in the group associated with comm which utilize that specific hard-					
40	ware resource type instance – and no other instance of the same hardware resource					
41	type – are included in the group of newcomm .					
42	If a given MPI process cannot be a member of a communicator that forms such a					
43	strict subset, or does not meet the above criteria, then MPI_COMM_NULL is returned					
44	in newcomm for this process.					
45						
46	Advice to implementors. In a high-quality MPI implementation, the number of					
47 48	different new valid communicators newcomm produced by this splitting operation					
-10	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 6.4. This sequence of returned newcomm communicators may differ from the sets of hardware resource types, as shown in the second splitting operation in Figure 6.3. (*End of rationale.*)

Advice to users. Each output communicator newcomm can represent a different hardware resource type (see Figure 6.3 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 info key mpi_hw_resource_type for each MPI process in the group associated with a returned newcomm communicator and the info key value is an implementation-defined string that indicates the hardware resource type represented by newcomm. The same hardware resource type must be set in all MPI processes in the group associated with newcomm.

Example 6.4 Recursive splitting of MPI_COMM_WORLD.

```
29
#define MAX_NUM_LEVELS 32
                                                                                30
                                                                                31
MPI_Comm hwcomm [MAX_NUM_LEVELS];
                                                                                32
int rank, level_num = 0;
                                                                                33
                                                                                34
hwcomm[level_num] = MPI_COMM_WORLD;
                                                                                35
                                                                                36
while((hwcomm[level_num] != MPI_COMM_NULL)
                                                                                37
      && (level_num < MAX_NUM_LEVELS-1)){
                                                                                38
  MPI_Comm_rank(hwcomm[level_num],&rank);
                                                                                39
  MPI_Comm_split_type(hwcomm[level_num],
                                                                                40
                        MPI_COMM_TYPE_HW_UNGUIDED,
                                                                                41
                        rank,
                                                                                42
                        MPI_INFO_NULL,
                                                                                43
                        &hwcomm[level_num+1]);
                                                                                44
 level_num++;
                                                                                45
}
                                                                                46
                                                                                47
```

 $\overline{7}$

 24

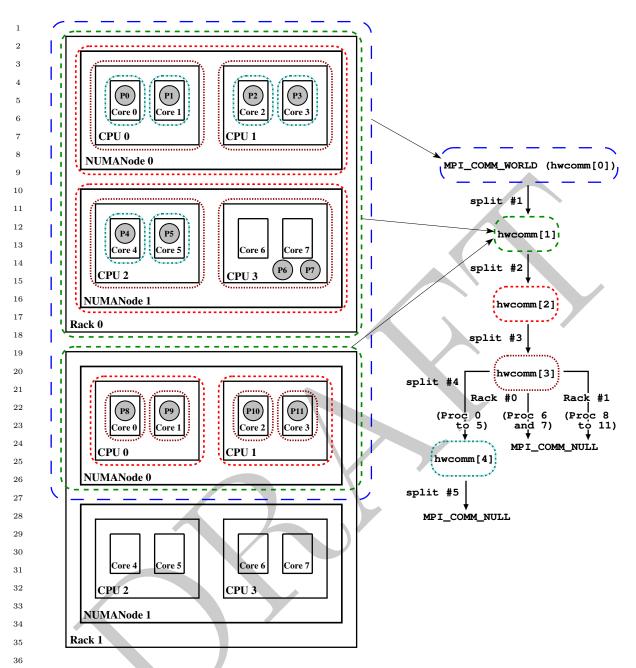


Figure 6.3: Recursive splitting of MPI_COMM_WORLD with MPI_COMM_SPLIT_TYPE and 37 MPI_COMM_TYPE_HW_UNGUIDED. Dashed lines represent communicators whilst solid lines 38 represent hardware resources. MPI processes (P0 to P11) utilize exclusively their respective 39 core, except for P6 and P7 which utilize 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

Unofficial Draft for Comment Only

6.4. COMMUNICATOR MANAGEMENT

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. (*End of advice to implementors.*)

MPI_COMM_CREATE_FROM_GROUP(group, stringtag, info, errhandler, newcomm) IN group group (handle) IN stringtag unique identifier for this operation (string) IN info info object (handle) IN errhandler error handler to be attached to new intra-communicator (handle) OUT new communicator (handle) newcomm C binding int MPI_Comm_create_from_group(MPI_Group group, const char *stringtag, MPI_Info info, MPI_Errhandler errhandler, MPI_Comm *newcomm) Fortran 2008 binding MPI_Comm_create_from_group(group, stringtag, info, errhandler, newcomm, ierror) TYPE(MPI_Group), INTENT(IN) :: group CHARACTER(LEN=*), INTENT(IN) :: stringtag TYPE(MPI_Info), INTENT(IN) :: info TYPE(MPI_Errhandler), INTENT(IN) :: errhandler TYPE(MPI_Comm), INTENT(OUT) :: newcomm INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_COMM_CREATE_FROM_GROUP(GROUP, STRINGTAG, INFO, ERRHANDLER, NEWCOMM, IERROR) INTEGER GROUP, INFO, ERRHANDLER, NEWCOMM, IERROR CHARACTER*(*) STRINGTAG MPI_COMM_CREATE_FROM_GROUP is similar to MPI_COMM_CREATE_GROUP, ex-

3536 cept that the set of MPI processes involved in the creation of the new intracommunicator 37 is specified by a group argument, rather than the group associated with a pre-existing com-38 municator. If a non-empty group is specified, then all MPI processes in that group must call 39 the function and each of these MPI processes must provide the same arguments, including 40 a group that contains the same members with the same ordering, and identical stringtag 41 value. In the event that MPI_GROUP_EMPTY is supplied as the group argument, then the 42call is a local operation and MPI_COMM_NULL is returned as newcomm. The stringtag argument is analogous to the tag used for MPI_COMM_CREATE_GROUP. If multiple threads at 4344a given MPI process perform concurrent MPI_COMM_CREATE_FROM_GROUP operations, 45the user must distinguish these operations by providing different stringtag arguments. The 46stringtag shall not exceed MPI_MAX_FROM_GROUP_TAG characters in length. For C, this 47includes space for a null terminating character. The errhandler argument specifies an error

1

 $\mathbf{2}$

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

285

48

handler to be attached to the new intracommunicator. This error handler will also be in voked if the MPI_COMM_CREATE_FROM_GROUP function encounters an error. The info
 argument provides hints and assertions, possibly MPI implementation dependent, which
 indicate desired characteristics and guide communicator creation.

Advice to users. The stringtag argument is used to distinguish concurrent communicator construction operations issued by different entities. As such, it is important to ensure that this argument is unique for each concurrent call to

MPI_COMM_CREATE_FROM_GROUP. Reverse domain name notation convention [1] is one approach to constructing unique stringtag arguments. See also example 10.8. (*End of advice to users.*)

12 13

5

6

7

8

9

10 11

6.4.3 Communicator Destructors

14 15 16

17

18

26

29 30

31

32

33

34

35 36

37

38

MPI_COMM_FREE(comm)

INOUT comm

communicator to be destroyed (handle)

```
<sup>19</sup>
20 C binding
```

int MPI_Comm_free(MPI_Comm *comm)

²² Fortran 2008 binding

```
23 MPI_Comm_free(comm, ierror)
24 TYPE(MPI_Comm), INTENT(INOUT) :: comm
```

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

```
<sup>27</sup> Fortran binding
```

```
<sup>28</sup> MPI_COMM_FREE(COMM, IERROR)
```

INTEGER COMM, IERROR

This collective operation marks the communication object for deallocation. The handle is set to MPI_COMM_NULL. Any pending operations that use this communicator will complete normally; the object is actually deallocated only if there are no other active references to it. This call applies to intra- and inter-communicators. The delete callback functions for all cached attributes (see Section 6.7) are called in arbitrary order.

Advice to implementors. Though collective, it is anticipated that this operation will normally be implemented to be local, though a debugging version of an MPI library might choose to synchronize. (*End of advice to implementors.*)

- 39 40 41
- 6.4.4 Communicator Info

⁴² Hints specified via info (see Chapter 9) allow a user to provide information to direct ⁴³ optimization. Providing hints may enable an implementation to deliver increased per-⁴⁴ formance or minimize use of system resources. An implementation is free to ignore all ⁴⁵ hints; however, applications must comply with any info hints they provide that are used ⁴⁶ by the MPI implementation (i.e., are returned by a call to MPI_COMM_GET_INFO) and ⁴⁷ that place a restriction on the behavior of the application. Hints are specified on a per ⁴⁸ communicator basis, in MPI_COMM_DUP_WITH_INFO, MPI_COMM_IDUP_WITH_INFO,

MPI_COMM_SET_INFO, MPI_COMM_SPLIT_TYPE, MPI_DIST_GRAPH_CREATE, and MPI_DIST_GRAPH_CREATE_ADJACENT, via the opaque info object. When an info object that specifies a subset of valid hints is passed to MPI_COMM_SET_INFO, there will be no effect on previously set or defaulted hints that the info does not specify.

Advice to implementors. It may happen that a program is coded with hints for one system, and later executes on another system that does not support these hints. In general, unsupported hints should simply be ignored. Needless to say, no hint can be mandatory. However, for each hint used by a specific implementation, a default value must be provided when the user does not specify a value for this hint. (*End of advice to implementors.*)

Info hints are not propagated by MPI from one communicator to another. The following info keys are valid for all communicators.

- mpi_assert_no_any_tag (boolean, default: false): If set to true, then the implementation
 may assume that the process will not use the MPI_ANY_TAG wildcard on the given
 communicator.
- mpi_assert_no_any_source (boolean, default: false): If set to true, then the implementation
 may assume that the process will not use the MPI_ANY_SOURCE wildcard on the given
 communicator.
- mpi_assert_exact_length (boolean, default: false): If set to true, then the implementation may assume that the lengths of messages received by the process are equal to the lengths of the corresponding receive buffers, for point-to-point communication operations on the given communicator.
- mpi_assert_allow_overtaking (boolean, default: false): If set to true, then the implementation may assume that point-to-point communications on the given communicator do not rely on the non-overtaking rule specified in Section 3.5. In other words, the application asserts that send operations are not required to be matched at the receiver in the order in which the send operations were posted by the sender, and receive operations are not required to be matched in the order in which they were posted by the receiver.

Advice to users. Use of the mpi_assert_allow_overtaking info key can result in nondeterminism in the message matching order. (*End of advice to users.*)

Advice to users. Some optimizations may only be possible when all processes in the group of the communicator provide a given info key with the same value. (End of advice to users.)

MPI_COMM_SET_INFO(comm, info)

INOUT	comm	communicator (handle)
IN	info	info object (handle)

C binding

Unofficial Draft for Comment Only

 24

1 int MPI_Comm_set_info(MPI_Comm comm, MPI_Info info) 2 Fortran 2008 binding 3 MPI_Comm_set_info(comm, info, ierror) 4 TYPE(MPI_Comm), INTENT(IN) :: comm 5TYPE(MPI_Info), INTENT(IN) :: info 6 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 7 8 Fortran binding 9 MPI_COMM_SET_INFO(COMM, INFO, IERROR) 10 INTEGER COMM, INFO, IERROR 11 MPI_COMM_SET_INFO updates the hints of the communicator associated with comm 12using the hints provided in info. This operation has no effect on previously set or defaulted 13 hints that are not specified by info. It also has no effect on previously set or defaulted 14 hints that are specified by info, but are ignored by the MPI implementation in this call to 15MPI_COMM_SET_INFO. MPI_COMM_SET_INFO is a collective routine. The info object 16 may be different on each process, but any info entries that an implementation requires to 17be the same on all processes must appear with the same value in each process's info object. 18 19 Advice to users. Some info items that an implementation can use when it creates 20a communicator cannot easily be changed once the communicator has been created. 21Thus, an implementation may ignore hints issued in this call that it would have 22 accepted in a creation call. An implementation may also be unable to update certain 23info hints in a call to MPI_COMM_SET_INFO. MPI_COMM_GET_INFO can be used to 24determine whether updates to existing info hints were ignored by the implementation. 25(End of advice to users.) 2627Setting info hints on the predefined communicators Advice to users. 28MPI_COMM_WORLD and MPI_COMM_SELF may have unintended effects, as changes to 29 these global objects may affect all components of the application, including libraries 30 and tools. Users must ensure that all components of the application that use a given 31communicator, including libraries and tools, can comply with any info hints associated 32 with that communicator. (End of advice to users.) 33 34 35 MPI_COMM_GET_INFO(comm, info_used) 36 37 IN communicator object (handle) comm 38 OUT info_used new info object (handle) 39 40 C binding 41 int MPI_Comm_get_info(MPI_Comm comm, MPI_Info *info_used) 4243 Fortran 2008 binding 44MPI_Comm_get_info(comm, info_used, ierror) 45TYPE(MPI_Comm), INTENT(IN) :: comm 46TYPE(MPI_Info), INTENT(OUT) :: info_used 47 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 48

Fortran binding

```
MPI_COMM_GET_INFO(COMM, INFO_USED, IERROR)
    INTEGER COMM, INFO_USED, IERROR
```

MPI_COMM_GET_INFO returns a new info object containing the hints of the communicator associated with comm. The current setting of all hints related to this communicator is returned in info_used. An MPI implementation is required to return all hints that are supported by the implementation and have default values specified; any user-supplied hints that were not ignored by the implementation; and any additional hints that were set by the implementation. If no such hints exist, a handle to a newly created info object is returned that contains no key/value pair. The user is responsible for freeing info_used via MPI_INFO_FREE.

Motivating Examples 6.5

6.5.1 Current Practice #1

Example #1a:

{

```
int main(int argc, char *argv[])
{
  int me, size;
  . . .
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  (void)printf("Process %d size %d\n", me, size);
  . . .
  MPI_Finalize();
  return 0;
}
```

Example #1a is a do-nothing program that initializes itself, and refers to the "all" communicator, and prints a message. It terminates itself too. This example does not imply that MPI supports printf-like communication itself.

Example #1b (supposing that size is even):

```
int main(int argc, char *argv[])
   int me, size;
   int SOME_TAG = 0;
   . . .
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);
                                          /* local */
  MPI_Comm_size(MPI_COMM_WORLD, &size); /* local */
   if((me % 2) == 0)
   {
```

1

 $\mathbf{2}$

3

4

5

6

7

8

9

10

11

1213

1415

1617

18

19

20

2122

23

24

2526

2728

29

30

31 32

33

34

35

36 37

38

39

40

41

42

43 44

454647

```
1
                 /* send unless highest-numbered process */
2
                 if((me + 1) < size)
3
                    MPI_Send(..., me + 1, SOME_TAG, MPI_COMM_WORLD);
4
             }
5
             else
6
                 MPI_Recv(..., me - 1, SOME_TAG, MPI_COMM_WORLD, &status);
7
8
              . . .
9
             MPI_Finalize();
10
             return 0;
11
          }
12
     Example #1b schematically illustrates message exchanges between "even" and "odd" pro-
13
     cesses in the "all" communicator.
14
15
     6.5.2 Current Practice #2
16
17
         int main(int argc, char *argv[])
18
         ſ
19
           int me, count;
20
           void *data;
21
           . . .
22
23
           MPI_Init(&argc, &argv);
^{24}
           MPI_Comm_rank(MPI_COMM_WORLD, &me);
25
26
           if(me == 0)
27
           {
28
               /* get input, create buffer ''data'' */
29
                . . .
30
           }
^{31}
32
           MPI_Bcast(data, count, MPI_BYTE, 0, MPI_COMM_WORLD);
33
34
            . . .
35
           MPI_Finalize();
36
           return 0;
37
         }
38
39
     This example illustrates the use of a collective communication.
40
41
            (Approximate) Current Practice #3
     6.5.3
42
       int main(int argc, char *argv[])
43
        {
44
45
          int me, count, count2;
46
          void *send_buf, *recv_buf, *send_buf2, *recv_buf2;
47
          MPI_Group group_world, grprem;
          MPI_Comm commWorker;
48
```

```
static int ranks[] = {0};
    . . .
    MPI_Init(&argc, &argv);
    MPI_Comm_group(MPI_COMM_WORLD, &group_world);
    MPI_Comm_rank(MPI_COMM_WORLD, &me); /* local */
    MPI_Group_excl(group_world, 1, ranks, &grprem); /* local */
    MPI_Comm_create(MPI_COMM_WORLD, grprem, &commWorker);
                                                                                      10
    if(me != 0)
                                                                                      11
    {
      /* compute on worker */
                                                                                      12
                                                                                      13
      . . .
                                                                    commWorker);
                                                                                      14
      MPI_Reduce(send_buf,recv_buf,count, MPI_INT, MPI_SUM, 1,
                                                                                      15
                                                                                      16
      MPI_Comm_free(&commWorker);
                                                                                      17
    }
    /* zero falls through immediately to this reduce, others do later...
                                                                                      18
                                                                                      19
    MPI_Reduce(send_buf2, recv_buf2, count2,
                MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
                                                                                      20
                                                                                      21
    MPI_Group_free(&group_world);
                                                                                      22
                                                                                      23
    MPI_Group_free(&grprem);
                                                                                      ^{24}
    MPI_Finalize();
                                                                                      25
    return 0;
                                                                                      26
  }
                                                                                      27
This example illustrates how a group consisting of all but the zeroth process of the "all"
                                                                                      28
```

group is created, and then how a communicator is formed (commWorker) for that new group. The new communicator is used in a collective call, and all processes execute a collective call in the MPI_COMM_WORLD context. This example illustrates how the two communicators (that inherently possess distinct contexts) protect communication. That is, communication in MPI_COMM_WORLD is insulated from communication in commWorker, and vice versa.

In summary, "group safety" is achieved via communicators because distinct contexts within communicators are enforced to be unique on any process.

```
6.5.4
      Example #4
```

The following example is meant to illustrate "safety" between point-to-point and collective communication. MPI guarantees that a single communicator can do safe point-to-point and collective communication.

```
#define TAG_ARBITRARY 12345
#define SOME_COUNT
                          50
int main(int argc, char *argv[])
{
  int me;
  MPI_Request request[2];
```

1

 $\mathbf{2}$

3

5 6

9

47

48

29

30

31

32

33

34

3536

37 38

39

40

```
1
          MPI_Status status[2];
\mathbf{2}
          MPI_Group group_world, subgroup;
3
           int ranks[] = \{2, 4, 6, 8\};
4
           MPI_Comm the_comm;
5
           . . .
6
           MPI_Init(&argc, &argv);
7
           MPI_Comm_group(MPI_COMM_WORLD, &group_world);
8
9
           MPI_Group_incl(group_world, 4, ranks, &subgroup); /* local */
10
           MPI_Group_rank(subgroup, &me);
                                                  /* local */
11
12
          MPI_Comm_create(MPI_COMM_WORLD, subgroup, &the_comm);
13
14
           if(me != MPI_UNDEFINED)
15
           {
16
               MPI_Irecv(buff1, count, MPI_DOUBLE, MPI_ANY_SOURCE, TAG_ARBITRARY,
17
                                   the_comm, request);
               MPI_Isend(buff2, count, MPI_DOUBLE, (me+1)%4, TAG_ARBITRARY,
18
19
                                   the_comm, request+1);
20
               for(i = 0; i < SOME_COUNT; i++)</pre>
21
                 MPI_Reduce(..., the_comm);
22
               MPI_Waitall(2, request, status);
23
24
               MPI_Comm_free(&the_comm);
25
           }
26
           MPI_Group_free(&group_world);
27
28
           MPI_Group_free(&subgroup);
29
          MPI_Finalize();
30
           return 0;
^{31}
        }
32
33
     6.5.5 Library Example #1
34
     The main program:
35
36
         int main(int argc, char *argv[])
37
        {
38
           int done = 0;
39
           user_lib_t *libh_a, *libh_b;
40
           void *dataset1, *dataset2;
41
           . . .
42
           MPI_Init(&argc, &argv);
43
           . . .
44
           init_user_lib(MPI_COMM_WORLD, &libh_a);
45
           init_user_lib(MPI_COMM_WORLD, &libh_b);
46
           . . .
47
           user_start_op(libh_a, dataset1);
48
```

```
1
     user_start_op(libh_b, dataset2);
                                                                                           2
      . . .
                                                                                           3
     while(!done)
     {
                                                                                           4
         /* work */
                                                                                           5
                                                                                           6
         . . .
         MPI_Reduce(..., MPI_COMM_WORLD);
         . . .
                                                                                           9
         /* see if done */
                                                                                          10
         . . .
     }
                                                                                          11
     user_end_op(libh_a);
                                                                                          12
     user_end_op(libh_b);
                                                                                          13
                                                                                          14
                                                                                          15
     uninit_user_lib(libh_a);
                                                                                          16
     uninit_user_lib(libh_b);
                                                                                          17
     MPI_Finalize();
                                                                                          18
     return 0;
                                                                                          19
   }
                                                                                          20
The user library initialization code:
                                                                                          21
   void init_user_lib(MPI_Comm comm, user_lib_t **handle)
                                                                                          22
                                                                                          23
   {
                                                                                          ^{24}
     user_lib_t *save;
                                                                                          25
                                                                                          26
     user_lib_initsave(&save); /* local */
     MPI_Comm_dup(comm, &(save->comm));
                                                                                          27
                                                                                          28
                                                                                          29
     /* other inits */
                                                                                          30
      . . .
                                                                                          ^{31}
                                                                                          32
     *handle = save;
                                                                                          33
   }
                                                                                          34
User start-up code:
                                                                                          35
                                                                                          36
   void user_start_op(user_lib_t *handle, void *data)
                                                                                          37
   {
     MPI_Irecv( ..., handle->comm, &(handle->irecv_handle) );
                                                                                          38
                                                                                          39
     MPI_Isend(..., handle->comm, &(handle->isend_handle) );
   }
                                                                                          40
                                                                                          41
User communication clean-up code:
                                                                                          42
   void user_end_op(user_lib_t *handle)
                                                                                          43
                                                                                          44
   ſ
                                                                                          45
     MPI_Status status;
                                                                                          46
     MPI_Wait(&handle->isend_handle, &status);
                                                                                          47
     MPI_Wait(&handle->irecv_handle, &status);
                                                                                          48
   }
```

```
1
     User object clean-up code:
\mathbf{2}
        void uninit_user_lib(user_lib_t *handle)
3
        {
4
          MPI_Comm_free(&(handle->comm));
5
           free(handle);
6
        }
7
8
     6.5.6 Library Example #2
9
10
     The main program:
11
12
        int main(int argc, char *argv[])
13
        {
14
           int ma, mb;
15
          MPI_Group group_world, group_a, group_b;
16
          MPI_Comm comm_a, comm_b;
17
18
           static int list_a[] = \{0, 1\};
19
     #if defined(EXAMPLE_2B) || defined(EXAMPLE_2C)
20
           static int list_b[] = {0, 2, 3};
21
     #else/* EXAMPLE_2A */
22
          static int list_b[] = \{0, 2\};
23
     #endif
^{24}
           int size_list_a = sizeof(list_a)/sizeof(int);
25
           int size_list_b = sizeof(list_b)/sizeof(int);
26
27
           . . .
28
          MPI_Init(&argc, &argv);
29
          MPI_Comm_group(MPI_COMM_WORLD, &group_world);
30
31
          MPI_Group_incl(group_world, size_list_a, list_a, &group_a);
32
          MPI_Group_incl(group_world, size_list_b, list_b, &group_b);
33
34
          MPI_Comm_create(MPI_COMM_WORLD, group_a, &comm_a);
35
          MPI_Comm_create(MPI_COMM_WORLD, group_b, &comm_b);
36
37
           if(comm_a != MPI_COMM_NULL)
38
              MPI_Comm_rank(comm_a, &ma);
39
           if(comm_b != MPI_COMM_NULL)
40
              MPI_Comm_rank(comm_b, &mb);
41
42
           if(comm_a != MPI_COMM_NULL)
43
              lib_call(comm_a);
44
45
           if(comm_b != MPI_COMM_NULL)
46
           {
47
             lib_call(comm_b);
48
             lib_call(comm_b);
```

```
}
     if(comm_a != MPI_COMM_NULL)
       MPI_Comm_free(&comm_a);
     if(comm_b != MPI_COMM_NULL)
       MPI_Comm_free(&comm_b);
     MPI_Group_free(&group_a);
     MPI_Group_free(&group_b);
     MPI_Group_free(&group_world);
     MPI_Finalize();
     return 0;
   }
The library:
   void lib_call(MPI_Comm comm)
   Ł
     int me, done = 0;
     MPI_Status status;
     MPI_Comm_rank(comm, &me);
     if(me == 0)
        while(!done)
        {
           MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
        }
     else
     {
       /* work */
                     O, ARBITRARY_TAG, comm);
       MPI_Send(...,
     }
#ifdef EXAMPLE_2C
     /* include (resp, exclude) for safety (resp, no safety): */
     MPI_Barrier(comm);
#endif
   }
```

The above example is really three examples, depending on whether or not one includes rank 3 in list_b, and whether or not a synchronize is included in lib_call. This example illustrates that, despite contexts, subsequent calls to lib_call with the same context need not be safe from one another (colloquially, "back-masking"). Safety is realized if the MPI_Barrier is added. What this demonstrates is that libraries have to be written carefully, even with contexts. When rank 3 is excluded, then the synchronize is not needed to get safety from back-masking.

Algorithms like "reduce" and "allreduce" have strong enough source selectivity properties so that they are inherently okay (no back-masking), provided that MPI provides basic guarantees. So are multiple calls to a typical tree-broadcast algorithm with the same root or different roots (see [60]). Here we rely on two guarantees of MPI: pairwise ordering of 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

35

36

37

38

39

40 41

42

43

1 messages between processes in the same context, and source selectivity — deleting either $\mathbf{2}$ feature removes the guarantee that back-masking cannot be required.

3 Algorithms that try to do non-deterministic broadcasts or other calls that include wild-4 card operations will not generally have the good properties of the deterministic implemen- $\mathbf{5}$ tations of "reduce," "allreduce," and "broadcast." Such algorithms would have to utilize 6 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 6.9.

1112

1314

15

16

41

 $\overline{7}$

8

9

10

Inter-Communication 6.6

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.

17All communication described thus far has involved communication between processes 18 that are members of the same group. This type of communication is called "intra-com-19munication" and the communicator used is called an "intra-communicator," as we have 20noted earlier in the chapter. 21

In modular and multi-disciplinary applications, different process groups execute distinct 22modules and processes within different modules communicate with one another in a pipeline 23or a more general module graph. In these applications, the most natural way for a process 24 to specify a target process is by the rank of the target process within the target group. In 25applications that contain internal user-level servers, each server may be a process group that 26provides services to one or more clients, and each client may be a process group that uses the 27services of one or more servers. It is again most natural to specify the target process by rank 28within the target group in these applications. This type of communication is called "inter 29-communication" and the communicator used is called an "inter-communicator," as 30 introduced earlier. 31

An inter-communication is a point-to-point communication between processes in 32 different groups. The group containing a process that initiates an inter-communication 33 operation is called the "local group," that is, the sender in a send and the receiver in a 34receive. The group containing the target process is called the "remote group," that is, 35 the receiver in a send and the sender in a receive. As in intra-communication, the target 36 process is specified using a (communicator, rank) pair. Unlike intra-communication, the rank 37 is relative to a second, remote group.

38 All inter-communicator constructors are blocking except for MPI_COMM_IDUP and 39 require that the local and remote groups be disjoint. 40

Advice to users. The groups must be disjoint for several reasons. Primarily, this 42is the intent of the intercommunicators — to provide a communicator for communication between disjoint groups. This is reflected in the definition of 43

44MPI_INTERCOMM_MERGE, which allows the user to control the ranking of the pro-45cesses in the created intracommunicator; this ranking makes little sense if the groups 46are not disjoint. In addition, the natural extension of collective operations to inter-47 communicators makes the most sense when the groups are disjoint. (End of advice to 48 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.

The routine MPI_COMM_TEST_INTER may be used to determine if a communicator is an inter- or intra-communicator. Inter-communicators can be used as arguments to some of the other communicator access routines. Inter-communicators cannot be used as input to some of the constructor routines for intra-communicators (for instance, MPI_CART_CREATE).

Advice to implementors. For the purpose of point-to-point communication, communicators can be represented in each process by a tuple consisting of:

group send_context receive_context source

For inter-communicators, group describes the remote group, and source is the rank of the process in the local group. For intra-communicators, group is the communicator group (remote=local), source is the rank of the process in this group, and send context and receive context are identical. A group can be represented by a rank-to-absolute-address translation table.

The inter-communicator cannot be discussed sensibly without considering processes in both the local and remote groups. Imagine a process \mathbf{P} in group \mathcal{P} , which has an intercommunicator $\mathbf{C}_{\mathcal{P}}$, and a process \mathbf{Q} in group \mathcal{Q} , which has an inter-communicator $\mathbf{C}_{\mathcal{Q}}$. Then

- $\mathbf{C}_{\mathcal{P}}$.group describes the group \mathcal{Q} and $\mathbf{C}_{\mathcal{Q}}$.group describes the group \mathcal{P} .
- $C_{\mathcal{P}}$.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} .
- $\mathbf{C}_{\mathcal{P}}$.source is rank of **P** in \mathcal{P} and $\mathbf{C}_{\mathcal{Q}}$.source is rank of **Q** in \mathcal{Q} .

Assume that \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} ; **source** and **send_context** are appended to the message.

Assume that \mathbf{Q} posts a receive with an explicit source argument using the intercommunicator. Then \mathbf{Q} matches **receive_context** to the message context and source argument to the message source.

The same algorithm is appropriate for intra-communicators as well.

Unofficial Draft for Comment Only

 24

1 2 3 4 5	sı lo	ipplement this m	odel with additio	ator accessors and onal structures, th lditional safe cont	at store informati	on about the
6 7 8	6.6.1	Inter-Communica	ator Accessors			
9 10	MPI_C	OMM_TEST_INT	ER(comm, flag)			
11	IN	comm		communicator (han	idle)	
12 13	OUT	flag		(logical)		
14 15 16	C bind int MP		ter(MPI_Comm co	omm, int *flag)		
17 18 19 20 21	MPI_Con TY LO	n 2008 binding mm_test_inter(PE(MPI_Comm), 1 GICAL, INTENT(TEGER, OPTIONA)	INTENT(IN) :: (DUT) :: flag	comm		
22 23 24 25 26	MPI_CO IN	n binding MM_TEST_INTER((TEGER COMM, IE GICAL FLAG		RROR)		
27 28 29	commu otherwi	nicator or an int se false.	ra-communicator	cess to determine . It returns true d as an input argu	if it is an inter-co	ommunicator,
30 31 32				nunication, the foll		
33 34 35 36		MPI_COMM_ MPI_COMM_ MPI_COMM_	GROUP returns	s the size of the loos s the local group. s the rank in the lo		
37 38	ſ	Table 6.1: MPI_C	OMM_* Function	Behavior (in Inte	r-Communication	Mode)
 39 40 41 42 43 44 45 46 47 	commu Both co MPI_CC because Th	nicators must be orresponding loca ONGRUENT or MP e either the local	either intra- or in al and remote gr I_SIMILAR. In pa or remote groups sors provide con	COMPARE is valid ter-communicators roups must compa- articular, it is poss were similar but n asistent access to l operations.	s, or else MPI_UNE are correctly to ge sible for MPI_SIMI not identical.	QUAL results. et the results LAR to result

MPI_COMM_REMOTE_SIZE(comm, size)			1
IN	comm	inter-communicator (handle)	2
OUT	size	number of processes in the remote group of comm (integer)	3
		(integer)	5 6
C bindir	ıg		7
	Comm_remote_size(MPI_Comm	comm, int *size)	8
Fortran	2008 binding		9
	_remote_size(comm, size, :	ierror)	10
	(MPI_Comm), INTENT(IN) ::		11 12
INTE	GER, INTENT(OUT) :: size		12
INTE	GER, OPTIONAL, INTENT(OUT)) :: ierror	14
Fortran	binding		15
MPI_COMM	_REMOTE_SIZE(COMM, SIZE, I	IERROR)	16
INTE	GER COMM, SIZE, IERROR		17
			18
			19 20
MPI_CON	IM_REMOTE_GROUP(comm,	group)	21
IN	comm	inter-communicator (handle)	22
OUT	group	remote group corresponding to comm (handle)	23
			24
C binding			
int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)			26 27
Fortran	2008 binding		27
MPI_Comm	_remote_group(comm, group	, ierror)	29
	C(MPI_Comm), INTENT(IN) ::		30
	(MPI_Group), INTENT(OUT)		31
INTE	GER, OPTIONAL, INTENT(OUT)) :: ierror	32
Fortran	binding		33
	L_REMOTE_GROUP(COMM, GROUP	, IERROR)	34
INTE	GER COMM, GROUP, IERROR		35 36
			37
	0	o both the local and remote groups of an inter-	38
		s function, as well as MPI_COMM_REMOTE_SIZE	39
hav	e been provided. (End of ratio	nale.)	40
	•••• Communitation (41
6.6.2 In	ter-Communicator Operations		42
	0	nter-communicator operations.	43
		bind two intra-communicators into an inter-com-	44 45
		MM_CREATE_FROM_GROUPS constructs an inter-	40
	1 V	ined disjoint groups; the function	47

MPI_INTERCOMM_MERGE creates an intra-communicator by merging the local and remote

Unofficial Draft for Comment Only

groups of an inter-communicator. The functions MPI_COMM_DUP and MPI_COMM_FREE,
 introduced previously, duplicate and free an inter-communicator, respectively.

³ Overlap of local and remote groups that are bound into an inter-communicator is ⁴ prohibited. If there is overlap, then the program is erroneous and is likely to deadlock. (If ⁵ a process is multithreaded, and MPI calls block only a thread, rather than a process, then ⁶ "dual membership" can be supported. It is then the user's responsibility to make sure that ⁷ calls on behalf of the two "roles" of a process are executed by two independent threads.)

⁸ The function MPI_INTERCOMM_CREATE can be used to create an inter-communicator ⁹ from two existing intra-communicators, in the following situation: At least one selected ¹⁰ member from each group (the "group leader") has the ability to communicate with the ¹¹ selected member from the other group; that is, a "peer" communicator exists to which both ¹² leaders belong, and each leader knows the rank of the other leader in this peer communicator. ¹³ Furthermore, members of each group know the rank of their leader.

¹⁴ Construction of an inter-communicator from two intra-communicators requires separate
 ¹⁵ collective operations in the local group and in the remote group, as well as a point-to-point
 ¹⁶ communication between a process in the local group and a process in the remote group.

¹⁷ When using the World Model, the MPI_COMM_WORLD communicator (or preferably a ¹⁸ dedicated duplicate thereof) can be this peer communicator. For applications that have used ¹⁹ the Sessions Model, spawn, or join it may be necessary to first create an intracommunicator ²⁰ to be used as peer.

The application topology functions described in Chapter 7 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 carte sian topology capabilities to that intra-communicator, creating an appropriate topology oriented intra-communicator. Alternatively, it may be reasonable to devise one's own application topology mechanisms for this case, without loss of generality.

²⁹ MPI_INTERCOMM_CREATE(local_comm, local_leader, peer_comm, remote_leader, tag, ³⁰ newintercomm)

00			
31	IN	local_comm	local intra-communicator (handle)
32 33	IN	local_leader	${\rm rank} \ {\rm of} \ {\rm local} \ {\rm group} \ {\rm leader} \ {\rm in} \ {\rm local_comm} \ ({\rm integer})$
34 35	IN	peer_comm	"peer" communicator; significant only at the local_leader (handle)
36 37	IN	remote_leader	rank of remote group leader in peer_comm; significant only at the local_leader (integer)
38 39	IN	tag	tag (integer)
40	OUT	newintercomm	new inter-communicator (handle)
41			
42	C bindin	g	
43	int MPI_	Intercomm_create(MPI_Comm	<pre>local_comm, int local_leader,</pre>
44		MPI_Comm peer_comm,	int remote_leader, int tag,
45		MPI_Comm *newinterco	mm)
46	Danta a		
47	Fortran A	2008 binding	
48			

```
1
MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader,
                                                                                          2
               tag, newintercomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: local_comm, peer_comm
    INTEGER, INTENT(IN) :: local_leader, remote_leader, tag
    TYPE(MPI_Comm), INTENT(OUT) :: newintercomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER,
               TAG, NEWINTERCOMM, IERROR)
                                                                                         10
    INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
                                                                                         11
                NEWINTERCOMM, IERROR
                                                                                         12
                                                                                         13
This call creates an inter-communicator. It is collective over the union of the local and
                                                                                         14
remote groups. MPI processes should provide identical local_comm and
                                                                                         15
local_leader arguments within each group. Wildcards are not permitted for remote_leader,
                                                                                         16
local_leader, and tag.
                                                                                         17
                                                                                         18
MPI_INTERCOMM_CREATE_FROM_GROUPS(local_group, local_leader, remote_group,
                                                                                         19
               remote_leader, stringtag, info, errhandler, newintercomm)
                                                                                         20
                                                                                         21
  IN
           local_group
                                       local group (handle)
                                                                                         22
  IN
           local_leader
                                       rank of local group leader in local_group (integer)
                                                                                         23
  IN
            remote_group
                                       remote group, significant only at local_leader (handle)
                                                                                         24
                                                                                         25
  IN
            remote_leader
                                       rank of remote group leader in remote_group,
                                                                                         26
                                       significant only at local_leader (integer)
                                                                                         27
           stringtag
                                       unique idenitifier for this operation (string)
  IN
                                                                                         28
  IN
            info
                                       info object (handle)
                                                                                         29
  IN
           errhandler
                                       error handler to be attached to new
                                                                                         30
                                                                                         31
                                       inter-communicator (handle)
                                                                                         32
            newintercomm
  OUT
                                       new inter-communicator (handle)
                                                                                         33
                                                                                         34
C binding
                                                                                         35
int MPI_Intercomm_create_from_groups(MPI_Group local_group,
                                                                                         36
               int local_leader, MPI_Group remote_group, int remote_leader,
                                                                                         37
               const char *stringtag, MPI_Info info,
                                                                                         38
               MPI_Errhandler errhandler, MPI_Comm *newintercomm)
                                                                                         39
                                                                                         40
Fortran 2008 binding
                                                                                         41
MPI_Intercomm_create_from_groups(local_group, local_leader, remote_group,
                                                                                         42
               remote_leader, stringtag, info, errhandler, newintercomm,
               ierror)
                                                                                         43
                                                                                         44
    TYPE(MPI_Group), INTENT(IN) :: local_group, remote_group
                                                                                         45
    INTEGER, INTENT(IN) :: local_leader, remote_leader
                                                                                         46
    CHARACTER(LEN=*), INTENT(IN) :: stringtag
                                                                                         47
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                         48
    TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
```

```
1
          TYPE(MPI_Comm), INTENT(OUT) :: newintercomm
\mathbf{2}
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     Fortran binding
4
     MPI_INTERCOMM_CREATE_FROM_GROUPS(LOCAL_GROUP, LOCAL_LEADER, REMOTE_GROUP,
5
                    REMOTE_LEADER, STRINGTAG, INFO, ERRHANDLER, NEWINTERCOMM,
6
                    IERROR)
7
          INTEGER LOCAL_GROUP, LOCAL_LEADER, REMOTE_GROUP, REMOTE_LEADER, INFO,
8
                     ERRHANDLER, NEWINTERCOMM, IERROR
9
          CHARACTER*(*) STRINGTAG
10
11
     This call creates an inter-communicator. Unlike MPI_Intercomm_create, this function uses
12
     as input previously defined, disjoint local and remote groups. The calling MPI process
13
     must be a member of the local group. The call is collective over the union of the local
14
     and remote groups. All involved MPI processes shall provide an identical value for the
15
     stringtag argument. Within each group, all MPI processes shall provide identical local_group,
16
     local_leader arguments. Wildcards are not permitted for the remote_leader or local_leader
17
     arguments. The stringtag argument serves the same purpose as the stringtag used in the
18
     MPI_COMM_CREATE_FROM_GROUP function; it differentiates concurrent calls in a multi-
19
     threaded environment. The stringtag shall not exceed MPI_MAX_FROM_GROUP_STRINGTAG
20
     characters in length. For C, this includes space for a null terminating character. In the
21
     event that MPI_GROUP_EMPTY is supplied as the local_group or remote_group or both, then
22
     the call is a local operation and MPI_COMM_NULL is returned as the newintercomm.
23
^{24}
     MPI_INTERCOMM_MERGE(intercomm, high, newintracomm)
25
26
       IN
                 intercomm
                                             Inter-Communicator (handle)
27
       IN
                 high
                                             (logical)
28
       OUT
                 newintracomm
                                             new intra-communicator (handle)
29
30
31
     C binding
32
     int MPI_Intercomm_merge(MPI_Comm intercomm, int high,
33
                    MPI_Comm *newintracomm)
34
     Fortran 2008 binding
35
     MPI_Intercomm_merge(intercomm, high, newintracomm, ierror)
36
          TYPE(MPI_Comm), INTENT(IN) :: intercomm
37
          LOGICAL, INTENT(IN) :: high
38
          TYPE(MPI_Comm), INTENT(OUT) :: newintracomm
39
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
^{41}
     Fortran binding
42
     MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR)
43
          INTEGER INTERCOMM, NEWINTRACOMM, IERROR
44
          LOGICAL HIGH
45
     This function creates an intra-communicator from the union of the two groups that are
46
     associated with intercomm. All processes should provide the same high value within each
```

 $_{47}$ associated with intercomm. An processes should provide the same light value within each $_{48}$ of the two groups. If processes in one group provided the value high = false and processes

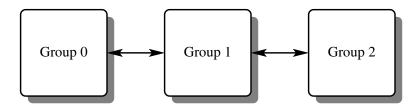


Figure 6.4: Three-group pipeline

in the other group provided the value high = true then the union orders the "low" group before the "high" group. If all processes provided the same high argument then the order of the union is arbitrary. This call is blocking and collective within the union of the two groups.

The error handler on the new intercommunicator in each process is inherited from the communicator that contributes the local group. Note that this can result in different processes in the same communicator having different error handlers.

Advice to implementors. The implementation of MPI_INTERCOMM_MERGE, MPI_COMM_FREE, and MPI_COMM_DUP are similar to the implementation of MPI_INTERCOMM_CREATE, except that contexts private to the input inter-communicator are used for communication between group leaders rather than contexts inside a bridge communicator. (*End of advice to implementors.*)

6.6.3 Inter-Communication Examples

```
Example 1: Three-Group "Pipeline"
```

Groups 0 and 1 communicate. Groups 1 and 2 communicate. Therefore, group 0 requires one inter-communicator, group 1 requires two inter-communicators, and group 2 requires 1 inter-communicator.

```
30
int main(int argc, char *argv[])
                                                                                 31
{
                                                                                 32
                            /* intra-communicator of local sub-group */
 MPI_Comm
             myComm;
                                                                                 33
 MPI_Comm
             myFirstComm;
                            /* inter-communicator */
                                                                                 34
             mySecondComm; /* second inter-communicator (group 1 only) */
 MPI_Comm
                                                                                 35
  int membershipKey;
                                                                                 36
  int rank;
                                                                                 37
                                                                                 38
 MPI_Init(&argc, &argv);
                                                                                 39
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
                                                                                 40
                                                                                 41
  /* User code must generate membershipKey in the range [0, 1, 2] */
                                                                                 42
 membershipKey = rank % 3;
                                                                                 43
                                                                                 44
  /* Build intra-communicator for local sub-group */
                                                                                 45
  MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);
                                                                                 46
                                                                                 47
  /* Build inter-communicators. Tags are hard-coded. */
                                                                                 48
```

2

6

9

10

11

12

13

14

15

16 17

18

19

20

21

22 23

24 25

26

27

28

```
1
2
3
4
                           Group 0
                                                               Group 2
                                             Group 1
5
6
7
                                  Figure 6.5: Three-group ring
8
9
10
           if (membershipKey == 0)
11
           {
                                    /* Group 0 communicates with group 1. */
12
             MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1,
13
                                     1, &myFirstComm);
14
           }
15
           else if (membershipKey == 1)
16
                            /* Group 1 communicates with groups 0 and 2. */
           {
17
             MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 0,
18
                                     1, &myFirstComm);
19
             MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 2,
20
                                     12, &mySecondComm);
21
           }
22
           else if (membershipKey == 2)
23
                                    /* Group 2 communicates with group 1. */
           {
^{24}
             MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1,
25
                                     12, &myFirstComm);
26
           }
27
28
           /* Do work ... */
29
30
           switch(membershipKey)
                                     /* free communicators appropriately */
31
           {
32
           case 1:
33
              MPI_Comm_free(&mySecondComm);
34
           case 0:
35
           case 2:
36
              MPI_Comm_free(&myFirstComm);
37
              break;
38
           }
39
40
           MPI_Finalize();
41
           return 0;
42
        }
43
44
     Example 2: Three-Group "Ring"
45
     Groups 0 and 1 communicate. Groups 1 and 2 communicate. Groups 0 and 2 communicate.
46
47
     Therefore, each requires two inter-communicators.
48
```

```
1
int main(int argc, char *argv[])
                                                                                  2
{
                                                                                  3
                           /* intra-communicator of local sub-group */
 MPI_Comm
             myComm;
 MPI_Comm
             myFirstComm; /* inter-communicators */
             mySecondComm;
 MPI_Comm
                                                                                  5
  int membershipKey;
                                                                                  6
  int rank;
  MPI_Init(&argc, &argv);
                                                                                  9
                                                                                 10
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
                                                                                 11
  . . .
                                                                                 12
  /* User code must generate membershipKey in the range [0, 1, 2] */
                                                                                 13
                                                                                 14
  membershipKey = rank % 3;
                                                                                 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
                                                                                 21
  {
                 /* Group 0 communicates with groups 1 and 2. */
    MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1,
                                                                                 22
                                                                                 23
                           1, &myFirstComm);
                                                                                 24
    MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 2,
                                                                                 25
                          2, &mySecondComm);
                                                                                 26
  }
  else if (membershipKey == 1)
                                                                                 27
            /* Group 1 communicates with groups 0 and 2. */
                                                                                 28
  {
    MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 0,
                                                                                 29
                                                                                 30
                           1, &myFirstComm);
                                                                                 31
    MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 2,
                          12, &mySecondComm);
                                                                                 32
                                                                                 33
                                                                                 34
  else if (membershipKey == 2)
           /* Group 2 communicates with groups 0 and 1. */
                                                                                 35
  ſ
    MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 0,
                                                                                 36
                                                                                 37
                          2, &myFirstComm);
                                                                                 38
    MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1,
                                                                                 39
                          12, &mySecondComm);
  }
                                                                                 40
                                                                                 41
                                                                                 42
  /* Do some work ... */
                                                                                 43
                                                                                 44
  /* Then free communicators before terminating... */
  MPI_Comm_free(&myFirstComm);
                                                                                 45
                                                                                 46
  MPI_Comm_free(&mySecondComm);
                                                                                 47
  MPI_Comm_free(&myComm);
                                                                                 48
  MPI_Finalize();
```

```
1
            return 0;
2
         }
3
4
     6.7
            Caching
5
6
      MPI provides a "caching" facility that allows an application to attach arbitrary pieces of
7
      information, called attributes, to three kinds of MPI objects, communicators, windows,
8
      and datatypes. More precisely, the caching facility allows a portable library to do the
9
      following:
10
11
         • pass information between calls by associating it with an MPI intra- or inter-commu-
12
           nicator, window, or datatype,
13
         • quickly retrieve that information, and
14
15
         • be guaranteed that out-of-date information is never retrieved, even if the object is
16
           freed and its handle subsequently reused by MPI.
17
18
          The caching capabilities, in some form, are required by built-in MPI routines such as
19
      collective communication and application topology. Defining an interface to these capa-
     bilities as part of the MPI standard is valuable because it permits routines like collective
20
21
      communication and application topologies to be implemented as portable code, and also
22
      because it makes MPI more extensible by allowing user-written routines to use standard
23
      MPI calling sequences.
^{24}
                             The communicator MPI_COMM_SELF is a suitable choice for post-
           Advice to users.
25
26
           ing process-local attributes, via this attribute-caching mechanism. (End of advice to
           users.)
27
28
           Rationale. In one extreme one can allow caching on all opaque handles. The other
29
           extreme is to only allow it on communicators. Caching has a cost associated with it
30
           and should only be allowed when it is clearly needed and the increased cost is modest.
^{31}
           This is the reason that windows and datatypes were added but not other handles.
32
           (End of rationale.)
33
34
          One difficulty is the potential for size differences between Fortran integers and C
35
      pointers.
                For this reason, the Fortran versions of these routines use integers of kind
36
      MPI_ADDRESS_KIND.
37
38
           Advice to implementors. High-quality implementations should raise an error when
39
           a keyval that was created by a call to MPI_XXX_CREATE_KEYVAL is used with an
40
           object of the wrong type with a call to MPI_YYY_GET_ATTR, MPI_YYY_SET_ATTR,
41
           MPI_YYY_DELETE_ATTR, or MPI_YYY_FREE_KEYVAL. To do so, it is necessary to
42
           maintain, with each keyval, information on the type of the associated user function.
43
           (End of advice to implementors.)
44
45
      6.7.1 Functionality
46
47
      Attributes can be attached to communicators, windows, and datatypes. Attributes are local
```

48 to the process and specific to the communicator to which they are attached. Attributes are

not propagated by MPI from one communicator to another except when the communicator is duplicated using MPI_COMM_DUP or MPI_COMM_IDUP (and even then the application must give specific permission through callback functions for the attribute to be copied).

Advice to users. Attributes in C are of type void*. Typically, such an attribute will be a pointer to a structure that contains further information, or a handle to an MPI object. In Fortran, attributes are of type INTEGER. Such attribute can be a handle to an MPI object, or just an integer-valued attribute. (*End of advice to users.*)

Advice to implementors. Attributes are scalar values, equal in size to, or larger than a C-language pointer. Attributes can always hold an MPI handle. (*End of advice to implementors.*)

The caching interface defined here requires that attributes be stored by MPI opaquely within a communicator, window, or datatype. Accessor functions include the following:

- obtain a key value (used to identify an attribute); the user specifies "callback" functions by which MPI informs the application when the communicator is destroyed or copied.
- store and retrieve the value of an attribute;

Advice to implementors. Caching and callback functions are only called synchronously, in response to explicit application requests. This avoids problems that result from repeated crossings between user and system space. (This synchronous calling rule is a general property of MPI.)

The choice of key values is under control of MPI. This allows MPI to optimize its implementation of attribute sets. It also avoids conflict between independent modules caching information on the same communicators.

A much smaller interface, consisting of just a callback facility, would allow the entire caching facility to be implemented by portable code. However, with the minimal callback interface, some form of table searching is implied by the need to handle arbitrary communicators. In contrast, the more complete interface defined here permits rapid access to attributes through the use of pointers in communicators (to find the attribute table) and cleverly chosen key values (to retrieve individual attributes). In light of the efficiency "hit" inherent in the minimal interface, the more complete interface defined here is seen to be superior. (*End of advice to implementors.*)

MPI provides the following services related to caching. They are all process local.

6.7.2 Communicators

Functions for caching on communicators are:

 $\mathbf{2}$

 $\overline{7}$

 24

308 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

```
1
     MPI_COMM_CREATE_KEYVAL(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval,
\mathbf{2}
                    extra_state)
3
       IN
                comm_copy_attr_fn
                                           copy callback function for comm_keyval (function)
4
       IN
                comm_delete_attr_fn
                                           delete callback function for comm_keyval (function)
5
6
       OUT
                comm_keyval
                                           key value for future access (integer)
7
       IN
                                           extra state for callback function
                extra_state
8
9
     C binding
10
     int MPI_Comm_create_keyval(MPI_Comm_copy_attr_function *comm_copy_attr_fn,
11
                    MPI_Comm_delete_attr_function *comm_delete_attr_fn,
12
                    int *comm_keyval, void *extra_state)
13
14
     Fortran 2008 binding
15
     MPI_Comm_create_keyval(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval,
16
                    extra_state, ierror)
17
         PROCEDURE(MPI_Comm_copy_attr_function), INTENT(IN) :: comm_copy_attr_fn
18
         PROCEDURE(MPI_Comm_delete_attr_function), INTENT(IN) ::
19
                     comm_delete_attr_fn
20
         INTEGER, INTENT(OUT) :: comm_keyval
21
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
     Fortran binding
24
     MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL,
25
                    EXTRA_STATE, IERROR)
26
         EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN
27
         INTEGER COMM_KEYVAL, IERROR
28
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
29
30
         Generates a new attribute key. Keys are locally unique in a process, and opaque to
31
     user, though they are explicitly stored in integers. Once allocated, the key value can be
32
     used to associate attributes and access them on any locally defined communicator.
33
     The C callback functions are:
34
     typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm, int comm_keyval,
35
                    void *extra_state, void *attribute_val_in,
36
                    void *attribute_val_out, int *flag);
37
     and
38
     typedef int MPI_Comm_delete_attr_function(MPI_Comm comm, int comm_keyval,
39
                    void *attribute_val, void *extra_state);
40
41
     which are the same as the MPI-1.1 calls but with a new name. The old names are deprecated.
42
     With the mpi_f08 module, the Fortran callback functions are:
43
     ABSTRACT INTERFACE
44
       SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,
45
                     attribute_val_in, attribute_val_out, flag, ierror)
46
         TYPE(MPI_Comm) :: oldcomm
47
         INTEGER :: comm_keyval, ierror
48
```

```
1
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                        \mathbf{2}
               attribute_val_out
                                                                                        3
    LOGICAL :: flag
                                                                                        4
and
                                                                                        5
ABSTRACT INTERFACE
                                                                                        6
  SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval,
               attribute_val, extra_state, ierror)
    TYPE(MPI_Comm) :: comm
                                                                                        9
    INTEGER :: comm_keyval, ierror
                                                                                       10
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
                                                                                       11
                                                                                       12
With the mpi module and mpif.h, the Fortran callback functions are:
                                                                                       13
SUBROUTINE COMM_COPY_ATTR_FUNCTION(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
                                                                                       14
              ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
                                                                                       15
    INTEGER OLDCOMM, COMM_KEYVAL, IERROR
                                                                                       16
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
                                                                                       17
               ATTRIBUTE_VAL_OUT
                                                                                       18
    LOGICAL FLAG
                                                                                       19
and
                                                                                       20
SUBROUTINE COMM_DELETE_ATTR_FUNCTION(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
                                                                                       21
              EXTRA_STATE, IERROR)
                                                                                       22
    INTEGER COMM, COMM_KEYVAL, IERROR
                                                                                       23
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
                                                                                       ^{24}
                                                                                       25
    The comm_copy_attr_fn function is invoked when a communicator is duplicated by
                                                                                       26
MPI_COMM_DUP or MPI_COMM_IDUP. comm_copy_attr_fn should be of type
                                                                                       27
MPI_Comm_copy_attr_function. The copy callback function is invoked for each key value in
                                                                                       28
oldcomm in arbitrary order. Each call to the copy callback is made with a key value and its
                                                                                       29
corresponding attribute. If it returns flag = 0 or .FALSE., then the attribute is deleted in
                                                                                       30
the duplicated communicator. Otherwise (flag = 1 or .TRUE.), the new attribute value is
                                                                                       31
```

set to the value returned in attribute_val_out. The function returns MPI_SUCCESS on success and an error code on failure (in which case MPI_COMM_DUP or MPI_COMM_IDUP will fail). The argument comm_copy_attr_fn may be specified as MPI_COMM_NULL_COPY_FN or MPI_COMM_DUP_FN from either C or Fortran. MPI_COMM_NULL_COPY_FN is a

or MPI_COMM_DUP_FN from either C or Fortran. MPI_COMM_NULL_COPY_FN is a function that does nothing other than returning flag = 0 or .FALSE. (depending on whether the keyval was created with a C or Fortran binding to MPI_COMM_CREATE_KEYVAL) and MPI_SUCCESS. MPI_COMM_DUP_FN is a simple-minded copy function that sets flag = 1 or .TRUE., returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS. These replace the MPI-1 predefined callbacks MPI_NULL_COPY_FN and MPI_DUP_FN, whose use is deprecated.

Advice to users. Even though both formal arguments attribute_val_in and attribute_val_out are of type void*, their usage differs. The C copy function is passed by MPI in attribute_val_in the value of the attribute, and in attribute_val_out the address of the attribute, so as to allow the function to return the (new) attribute value. The use of type void* for both is to avoid messy type casts.

Unofficial Draft for Comment Only

32

33

34

35

36

37

38

39

40

41

42

43 44

 $45 \\ 46$

47

1	A valid copy function is one that completely duplicates the information by making
2	a full duplicate copy of the data structures implied by an attribute; another might
3	just make another reference to that data structure, while using a reference-count
4	mechanism. Other types of attributes might not copy at all (they might be specific
5	to oldcomm only). (End of advice to users.)
6	
7	Advice to implementors. A C interface should be assumed for copy and delete
8	functions associated with key values created in C; a Fortran calling interface should
9	be assumed for key values created in Fortran. (End of advice to implementors.)
10	
11	Analogous to comm_copy_attr_fn is a callback deletion function, defined as follows.
12	The comm_delete_attr_fn function is invoked when a communicator is deleted by
13	MPI_COMM_FREE or when a call is made explicitly to MPI_COMM_DELETE_ATTR.
14	comm_delete_attr_fn should be of type MPI_Comm_delete_attr_function.
15	This function is called by MPI_COMM_FREE, MPI_COMM_DELETE_ATTR, and
16	MPI_COMM_SET_ATTR to do whatever is needed to remove an attribute. The function
17	returns MPI_SUCCESS on success and an error code on failure (in which case
18	MPI_COMM_FREE will fail).
19	The argument comm_delete_attr_fn may be specified as
20	MPI_COMM_NULL_DELETE_FN from either C or Fortran.
20	MPI_COMM_NULL_DELETE_FN is a function that does nothing, other than returning
21	MPI_SUCCESS. MPI_COMM_NULL_DELETE_FN replaces MPI_NULL_DELETE_FN, whose
22	use is deprecated.
	If an attribute copy function or attribute delete function returns other than
24	MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_COMM_FREE),
25	is erroneous.
26	The special key value MPI_KEYVAL_INVALID is never returned by
27	MPI_COMM_CREATE_KEYVAL. Therefore, it can be used for static initialization of key
28	values.
29	values.
30	Advice to implementors. The predefined Fortran functions
31	MPI_COMM_NULL_COPY_FN, MPI_COMM_DUP_FN, and
32	MPI_COMM_NULL_DELETE_FN are defined in the mpi module (and mpif.h) and
33	the mpi_f08 module with the same name, but with different interfaces. Each function
34	can coexist twice with the same name in the same MPI library, one routine as an
35	implicit interface outside of the mpi module, i.e., declared as EXTERNAL, and the other
36	routine within mpi_f08 declared with CONTAINS. These routines have different link
37	names, which are also different to the link names used for the routines used in C.
38	(End of advice to implementors.)
39	(Lina of addice to implementors.)
40	Advice to users. Callbacks, including the predefined Fortran functions
41	MPI_COMM_NULL_COPY_FN, MPI_COMM_DUP_FN, and
42	MPI_COMM_NULL_DELETE_FN should not be passed from one application routine
43	that uses the mpi_f08 module to another application routine that uses the mpi module
44	or mpif.h, and vice versa; see also the advice to users on page 762. (End of advice to
45	users.)
46	
47	
48	

		N	_
MPI_COM	M_FREE_KEYVAL(comm_key	val)	1
INOUT	comm_keyval	key value (integer)	3
			4
C binding			
int MPI_C	Comm_free_keyval(int *com	n_keyval)	6
Fortran 2	2008 binding		7
MPI_Comm_free_keyval(comm_keyval, ierror)			8
	ER, INTENT(INOUT) :: com		9
INTEG	ER, OPTIONAL, INTENT(OUT)) :: ierror	10
Fortran h	anding		11
	FREE_KEYVAL(COMM_KEYVAL,	IERROR)	12 13
	ER COMM_KEYVAL, IERROR		13
			15
		is function sets the value of keyval to	16
		ot erroneous to free an attribute key that is in use, re until after all references (in other communicators	17
	-	d. These references need to be explicitly freed by the	18
-	,	1_DELETE_ATTR that free one attribute instance,	19
· · ·		ree all attribute instances associated with the freed	20
communic		ee an attribute instances associated with the need	21
community			22
			23
MPI_COM	M_SET_ATTR(comm, comm_	keyval, attribute_val)	24
INOUT	comm	communicator to which attribute will be attached	25
		(handle)	26
IN	comm_keyval	key value (integer)	27 28
IN	attribute_val	attribute value	20
	attribute_var		30
C binding			31
		nm, int comm_keyval, void *attribute_val)	32
		mm, ind comm_Reyvar, voia (abbiibabe_var)	33
	2008 binding		34
		al, attribute_val, ierror)	35
	(MPI_Comm), INTENT(IN) ::		36
	ER, INTENT(IN) :: comm_ke		37
	ER, OPTIONAL, INTENT(OUT)), INTENT(IN) :: attribute_val	38
	ER, UPIIONAL, INIENI(UOI,) :: leffor	39
Fortran b	oinding		40
MPI_COMM_	SET_ATTR(COMM, COMM_KEYV	AL, ATTRIBUTE_VAL, IERROR)	41
	ER COMM, COMM_KEYVAL, IEI		42 43
INTEC	ER(KIND=MPI_ADDRESS_KIND)) ATTRIBUTE_VAL	43 44
This f	unction stores the stipulated a	attribute value attribute_val for subsequent retrieval	44 45
	-	lue is already present, then the outcome is as if	46

by MPI_COMM_GET_ATTR. If the value is already present, then the outcome is as if MPI_COMM_DELETE_ATTR was first called to delete the previous value (and the callback function comm_delete_attr_fn was executed), and a new value was next stored. The call

is erroneous if there is no key with value keyval; in particular MPI_KEYVAL_INVALID is an
 erroneous key value. The call will fail if the comm_delete_attr_fn function returned an error
 code other than MPI_SUCCESS.

```
5
      MPI_COMM_GET_ATTR(comm, comm_keyval, attribute_val, flag)
6
\overline{7}
       IN
                                              communicator to which the attribute is attached
                 comm
8
                                              (handle)
9
                 comm_keyval
       IN
                                              key value (integer)
10
       OUT
                  attribute_val
                                              attribute value, unless flag = false
11
12
        OUT
                 flag
                                              false if no attribute is associated with the key
13
                                              (logical)
14
15
      C binding
16
      int MPI_Comm_get_attr(MPI_Comm comm, int comm_keyval, void *attribute_val,
17
                     int *flag)
18
     Fortran 2008 binding
19
      MPI_Comm_get_attr(comm, comm_keyval, attribute_val, flag, ierror)
20
          TYPE(MPI_Comm), INTENT(IN) :: comm
21
          INTEGER, INTENT(IN) :: comm_keyval
22
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
23
          LOGICAL, INTENT(OUT) :: flag
24
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26
      Fortran binding
27
     MPI_COMM_GET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
28
          INTEGER COMM, COMM_KEYVAL, IERROR
29
          INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
30
          LOGICAL FLAG
31
          Retrieves attribute value by key. The call is erroneous if there is no key with value
32
      keyval. On the other hand, the call is correct if the key value exists, but no attribute is
33
34
      attached on comm for that key; in such case, the call returns flag = false. In particular
      MPI_KEYVAL_INVALID is an erroneous key value.
35
36
           Advice to users.
                             The call to MPI_Comm_set_attr passes in attribute_val the value
37
           of the attribute; the call to MPI_Comm_get_attr passes in attribute_val the address
38
           of the location where the attribute value is to be returned. Thus, if the attribute
39
           value itself is a pointer of type void*, then the actual attribute_val parameter to
40
           MPI_Comm_set_attr will be of type void* and the actual attribute_val parameter to
41
           MPI_Comm_get_attr will be of type void**. (End of advice to users.)
42
43
           Rationale. The use of a formal parameter attribute_val of type void* (rather than
44
           void**) avoids the messy type casting that would be needed if the attribute value is
45
           declared with a type other than void*. (End of rationale.)
46
47
48
```

MPI_COMM_DELETE_ATTR(comm, comm_keyval) ¹				
INOUT	comm	communicator from which the attribute is deleted (handle)	2 3 4	
IN	comm_keyval	key value (integer)	4 5	
C bindin int MPI_(g Comm_delete_attr(MPI_Comm	comm, int comm_keyval)	6 7 8	
MPI_Comm_ TYPE INTEC	2008 binding _delete_attr(comm, comm_k (MPI_Comm), INTENT(IN) :: GER, INTENT(IN) :: comm_k GER, OPTIONAL, INTENT(OUT	comm eyval	9 10 11 12 13 14	
	Dinding _DELETE_ATTR(COMM, COMM_K GER COMM, COMM_KEYVAL, IE		15 16 17 18	
comm_dele comm_dele When MPI_COM invoked (i	Delete attribute from cache by key. This function invokes the attribute delete function comm_delete_attr_fn specified when the keyval was created. The call will fail if the comm_delete_attr_fn function returns an error code other than MPI_SUCCESS. Whenever a communicator is replicated using the function MPI_COMM_DUP or MPI_COMM_IDUP, all call-back copy functions for attributes that are currently set are invoked (in arbitrary order). Whenever a communicator is deleted using the function MPI_COMM_FREE all callback delete functions for attributes that are currently set are			
	ndows ions for caching on windows a	re:	27 28 29 30	
MPI_WIN	_CREATE_KEYVAL(win_copy_ extra_state)	_attr_fn, win_delete_attr_fn, win_keyval,	31 32 33	
IN IN OUT IN	win_copy_attr_fn win_delete_attr_fn win_keyval extra_state	copy callback function for win_keyval (function) delete callback function for win_keyval (function) key value for future access (integer) extra state for callback function	34 35 36 37 38 39	
	C binding int MPI_Win_create_keyval(MPI_Win_copy_attr_function *win_copy_attr_fn, MPI_Win_delete_attr_function *win_delete_attr_fn, int *win_keyval, void *extra_state)			
MPI_Win_o	extra_state, ierror)	<pre>tr_fn, win_delete_attr_fn, win_keyval, unction), INTENT(IN) :: win_copy_attr_fn</pre>	44 45 46 47 48	

```
1
         PROCEDURE(MPI_Win_delete_attr_function), INTENT(IN) ::
\mathbf{2}
                    win_delete_attr_fn
3
         INTEGER, INTENT(OUT) :: win_keyval
4
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     Fortran binding
7
     MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,
8
                   EXTRA_STATE, IERROR)
9
         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-minded 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,
20
     other than returning MPI_SUCCESS.
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
48
```

```
1
SUBROUTINE WIN_COPY_ATTR_FUNCTION(OLDWIN, WIN_KEYVAL, EXTRA_STATE,
                                                                                        \mathbf{2}
              ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
    INTEGER OLDWIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
               ATTRIBUTE_VAL_OUT
    LOGICAL FLAG
and
SUBROUTINE WIN_DELETE_ATTR_FUNCTION(WIN, WIN_KEYVAL, ATTRIBUTE_VAL,
              EXTRA_STATE, IERROR)
                                                                                        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
           win_keyval
 INOUT
                                      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
           win
                                      window to which attribute will be attached (handle)
                                                                                       35
                                                                                       36
           win_keyval
 IN
                                      key value (integer)
                                                                                       37
           attribute_val
 IN
                                      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
```

```
1
     Fortran binding
\mathbf{2}
     MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR)
3
          INTEGER WIN, WIN_KEYVAL, IERROR
4
          INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
5
6
\overline{7}
     MPI_WIN_GET_ATTR(win, win_keyval, attribute_val, flag)
8
       IN
                                            window to which the attribute is attached (handle)
                win
9
                win_keyval
10
       IN
                                            key value (integer)
11
       OUT
                attribute_val
                                            attribute value, unless flag = false
12
       OUT
                flag
                                            false if no attribute is associated with the key
13
                                            (logical)
14
15
     C binding
16
     int MPI_Win_get_attr(MPI_Win win, int win_keyval, void *attribute_val,
17
                    int *flag)
18
19
     Fortran 2008 binding
20
     MPI_Win_get_attr(win, win_keyval, attribute_val, flag, ierror)
21
          TYPE(MPI_Win), INTENT(IN) :: win
22
          INTEGER, INTENT(IN) :: win_keyval
23
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
24
          LOGICAL, INTENT(OUT) :: flag
25
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
     Fortran binding
27
     MPI_WIN_GET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
28
          INTEGER WIN, WIN_KEYVAL, IERROR
29
          INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
30
          LOGICAL FLAG
31
32
33
34
     MPI_WIN_DELETE_ATTR(win, win_keyval)
35
       INOUT
                                            window from which the attribute is deleted (handle)
                 win
36
       IN
                win_keyval
                                            key value (integer)
37
38
     C binding
39
     int MPI_Win_delete_attr(MPI_Win win, int win_keyval)
40
41
     Fortran 2008 binding
42
     MPI_Win_delete_attr(win, win_keyval, ierror)
43
          TYPE(MPI_Win), INTENT(IN) :: win
44
          INTEGER, INTENT(IN) :: win_keyval
45
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
     Fortran binding
48
     MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR)
```

I	NTEGER WIN, WIN_KEYVAL, IER	ROR	$\frac{1}{2}$
c - .			3
6.7.4	Datatypes		4
The new functions for caching on datatypes are:		5 6	
			7
MPI_7	YPE_CREATE_KEYVAL(type_cc extra_state)	py_attr_fn, type_delete_attr_fn, type_keyval,	8 9
IN	type_copy_attr_fn	copy callback function for <code>type_keyval</code> (function)	10
IN	type_delete_attr_fn	delete callback function for type_keyval (function)	11 12
OUT	type_keyval	key value for future access (integer)	13
IN	extra_state	extra state for callback function	14
	-		15
C bin	ding		16 17
int M	PI_Type_create_keyval(MPI_T	<pre>ype_copy_attr_function *type_copy_attr_fn,</pre>	17
		r_function *type_delete_attr_fn,	19
	int *type_keyval, v	oid *extra_state)	20
Fortr	an 2008 binding		21
MPI_T	ype_create_keyval(type_copy	_attr_fn, type_delete_attr_fn, type_keyval,	22
	extra_state, ierror		23
		_function), INTENT(IN) :: type_copy_attr_fn	24
<pre>PROCEDURE(MPI_Type_delete_attr_function), INTENT(IN) ::</pre>			25
type_delete_attr_fn			26
INTEGER, INTENT(OUT) :: type_keyval INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state			27
			28
1	NTEGER, OPTIONAL, INTENT(OU	[) :: lerror	29 30
Fortr	an binding		31
MPI_T	YPE_CREATE_KEYVAL(TYPE_COPY	_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL,	32
	EXTRA_STATE, IERROR)	33
	XTERNAL TYPE_COPY_ATTR_FN, T	TYPE_DELETE_ATTR_FN	34
	NTEGER TYPE_KEYVAL, IERROR		35
I	NTEGER(KIND=MPI_ADDRESS_KINI	D) EXTRA_STATE	36
Т	he argument type_copy_attr_fn n	nay be specified as MPI_TYPE_NULL_COPY_FN or	37
		Fortran. MPI_TYPE_NULL_COPY_FN is a function	38
that d	oes nothing other than returning	g flag = 0 and MPI_SUCCESS. MPI_TYPE_DUP_FN	39
is a sin	nple-minded copy function that a	sets $flag = 1$, returns the value of attribute_val_in in	40
attribu	te_val_out, and returns MPI_SUC	CESS.	41
	с <u>,</u>	may be specified as MPI_TYPE_NULL_DELETE_FN	42
		NULL_DELETE_FN is a function that does nothing,	43
	than returning MPI_SUCCESS.		44
	callback functions are:		45 46
typed		<pre>inction(MPI_Datatype oldtype, id to the set of the</pre>	40 47
	<pre>int type_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag);</pre>		

```
1
     and
\mathbf{2}
     typedef int MPI_Type_delete_attr_function(MPI_Datatype datatype,
3
                    int type_keyval, void *attribute_val, void *extra_state);
4
     With the mpi_f08 module, the Fortran callback functions are:
5
     ABSTRACT INTERFACE
6
       SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state,
7
                    attribute_val_in, attribute_val_out, flag, ierror)
8
         TYPE(MPI_Datatype) :: oldtype
9
         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
20
         INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
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
       INOUT
                type_keyval
                                           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
```

INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	1 2
Fortran b	Fortran binding		
	MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)		
INTEG	ER TYPE_KEYVAL, IERROR		5
			6
	_SET_ATTR(datatype, type_k	overal attribute val)	7
		- ,	8 9
INOUT	datatype	datatype to which attribute will be attached (handle)	10
IN	type_keyval	key value (integer)	11
IN	attribute_val	attribute value	12
			13
C binding			14
int MPI_T	ype_set_attr(MPI_Datatype void *attribute_val)	a datatype, int type_keyval,	15 16
Fortran 2	008 binding		17
	6	eyval, attribute_val, ierror)	18
• -	MPI_Datatype), INTENT(IN)	-	19 20
INTEG	ER, INTENT(IN) :: type_ke	yval	20 21
		, INTENT(IN) :: attribute_val	22
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	23
Fortran b	Fortran binding		
MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)			25
	ER DATATYPE, TYPE_KEYVAL,		26
INTEG	ER(KIND=MPI_ADDRESS_KIND)	ATTRIBUTE_VAL	27
			28 29
		and attribute val flag)	30
	_GET_ATTR(datatype, type_k		31
IN	datatype	datatype to which the attribute is attached (handle)	32
IN	type_keyval	key value (integer)	33
OUT	attribute_val	attribute value, unless $flag = false$	34
OUT	flag	false if no attribute is associated with the key	35 36
		(logical)	37
			38
C binding	;		39
int MPI_T		datatype, int type_keyval,	40
	<pre>void *attribute_val,</pre>	int *flag)	41
Fortran 2	008 binding		42
MPI_Type_	get_attr(datatype, type_k	eyval, attribute_val, flag, ierror)	43
	MPI_Datatype), INTENT(IN)	• =	44 45
	ER, INTENT(IN) :: type_ke	•	46
		, INTENT(OUT) :: attribute_val	47
LUGIC	AL, INTENT(OUT) :: flag		48

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     Fortran binding
3
     MPI_TYPE_GET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
4
         INTEGER DATATYPE, TYPE_KEYVAL, IERROR
5
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
6
         LOGICAL FLAG
7
8
9
     MPI_TYPE_DELETE_ATTR(datatype, type_keyval)
10
11
       INOUT
                                            datatype from which the attribute is deleted (handle)
                datatype
12
       IN
                type_keyval
                                            key value (integer)
13
14
     C binding
15
     int MPI_Type_delete_attr(MPI_Datatype datatype, int type_keyval)
16
17
     Fortran 2008 binding
18
     MPI_Type_delete_attr(datatype, type_keyval, ierror)
19
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
20
         INTEGER, INTENT(IN) :: type_keyval
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
     Fortran binding
23
     MPI_TYPE_DELETE_ATTR(DATATYPE, TYPE_KEYVAL, IERROR)
24
         INTEGER DATATYPE, TYPE_KEYVAL, IERROR
25
26
27
            Error Class for Invalid Keyval
     6.7.5
28
     Key values for attributes are system-allocated, by
29
     MPI_{XXX}_CREATE_KEYVAL. Only such values can be passed to the functions that use
30
     key values as input arguments. In order to signal that an erroneous key value has been
^{31}
     passed to one of these functions, there is a new MPI error class: MPI_ERR_KEYVAL. It can
32
     be returned by MPI_ATTR_PUT, MPI_ATTR_GET, MPI_ATTR_DELETE,
33
34
     MPI_KEYVAL_FREE,
     MPI_{XXX}_DELETE_ATTR,
35
     MPI_{XXX}_SET_ATTR,
36
     MPI_{XXX}_GET_ATTR,
37
     MPI_{XXX}_FREE_KEYVAL, MPI_COMM_DUP, MPI_COMM_IDUP,
38
     MPI_COMM_DISCONNECT, and MPI_COMM_FREE. The last four are included because
39
     keyval is an argument to the copy and delete functions for attributes.
40
41
42
     6.7.6 Attributes Example
43
                              This example shows how to write a collective communication
          Advice to users.
44
          operation that uses caching to be more efficient after the first call. (End of advice to
45
          users.)
46
47
48
```

```
1
/* key for this module's stuff: */
                                                                                  2
static int gop_key = MPI_KEYVAL_INVALID;
typedef struct
ſ
                                                                                  5
   int ref_count;
                            /* reference count */
                                                                                  6
   /* other stuff, whatever else we want */
} gop_stuff_type;
                                                                                  9
                                                                                  10
void Efficient_Collective_Op(MPI_Comm comm, ...)
                                                                                  11
Ł
  gop_stuff_type *gop_stuff;
                                                                                  12
                                                                                  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
  Ł
    if ( ! MPI_Comm_create_keyval(gop_stuff_copier,
                                                                                 20
                                                                                 21
                               gop_stuff_destructor,
                               &gop_key, (void *)0)) {
                                                                                 22
    /* get the key while assigning its copy and delete callback
                                                                                 23
                                                                                 ^{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.
       We will use the cached information */
                                                                                  32
                                                                                 33
                                                                                 34
  else
  { /* This is a group that we have not yet cached anything in.
                                                                                 35
       We will now do so.
                                                                                 36
                                                                                 37
    */
                                                                                 38
                                                                                 39
    /* First, allocate storage for the stuff we want,
       and initialize the reference count */
                                                                                  40
                                                                                 41
                                                                                 42
    gop_stuff = (gop_stuff_type *) malloc(sizeof(gop_stuff_type));
    if (gop_stuff == NULL) { /* abort on out-of-memory error */ }
                                                                                 43
                                                                                 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 */
```

```
1
2
            /* Third, store gop_stuff as the attribute value */
3
            MPI_Comm_set_attr(comm, gop_key, gop_stuff);
4
          }
5
          /* Then, in any case, use contents of *gop_stuff
6
             to do the global op ... */
7
        }
8
9
        /* The following routine is called by MPI when a group is freed */
10
11
        int gop_stuff_destructor(MPI_Comm comm, int keyval, void *gop_stuffP,
12
                                  void *extra)
13
        {
14
          gop_stuff_type *gop_stuff = (gop_stuff_type *)gop_stuffP;
15
          if (keyval != gop_key) { /* abort -- programming error */ }
16
17
          /* The group's being freed removes one reference to gop_stuff */
18
          gop_stuff->ref_count -= 1;
19
          /* If no references remain, then free the storage */
20
21
          if (gop_stuff->ref_count == 0) {
22
            free((void *)gop_stuff);
23
          }
24
          return MPI_SUCCESS;
25
        }
26
27
        /* The following routine is called by MPI when a group is copied */
28
        int gop_stuff_copier(MPI_Comm comm, int keyval, void *extra,
29
                        void *gop_stuff_inP, void *gop_stuff_outP, int *flag)
30
        {
31
          gop_stuff_type *gop_stuff_in = (gop_stuff_type *)gop_stuff_inP;
32
          gop_stuff_type **gop_stuff_out = (gop_stuff_type **)gop_stuff_outP;
33
          if (keyval != gop_key) { /* abort -- programming error */ }
34
35
          /* The new group adds one reference to this gop_stuff */
36
          gop_stuff_in->ref_count += 1;
37
          *gop_stuff_out = gop_stuff_in;
38
          return MPI_SUCCESS;
39
        }
40
41
```

6.8 Naming Objects

42

There are many occasions on which it would be useful to allow a user to associate a printable
 identifier with an MPI communicator, window, or datatype, for instance error reporting,
 debugging, and profiling. The names attached to opaque objects do not propagate when
 the object is duplicated or copied by MPI routines. For communicators this can be achieved
 using the following two functions.

MPI_COMM_SET_NAME(comm, comm_name)			
INOUT	comm	communicator whose identifier is to be set (handle)	2
IN	comm_name	the character string which is remembered as the	3 4
		name (string)	5
			6
C bindin	•		7
int MPI_	Comm_set_name(MPI_Comm cor	nm, const char *comm_name)	8
Fortran	2008 binding		9
MPI_Comm	_set_name(comm, comm_name)	, ierror)	10 11
	(MPI_Comm), INTENT(IN) ::		12
	ACTER(LEN=*), INTENT(IN)		13
INTE	GER, OPTIONAL, INTENT(OUT)) :: lerror	14
Fortran	binding		15
	_SET_NAME(COMM, COMM_NAME)	, IERROR)	16
	GER COMM, IERROR		17
CHAR	ACTER*(*) COMM_NAME		18 19
		ser to associate a name string with a communicator.	20
		MPI_COMM_SET_NAME will be saved inside the	21
	5 (aller immediately after the call, or allocated on the	22
/	0	ficant but trailing ones are not. (non-collective) operation, which only affects the	23
		process which made the MPI_COMM_SET_NAME	24
		ame (or any) name be assigned to a communicator	25 26
	rocess where it exists.		20
			28
		IM_SET_NAME is provided to help debug code, it	29
	ts, to avoid confusion. (<i>End of</i>	to a communicator in all of the processes where it	30
CAIS	is, to avoid confusion. (<i>End of</i>		31
The	length of the name which can	be stored is limited to the value of	32
MPI_MAX	OBJECT_NAME in Fortran and	d MPI_MAX_OBJECT_NAME-1 in C to allow for the	33 34
		es longer than this will result in truncation of the	35
name. MF	PI_MAX_OBJECT_NAME must h	ave a value of at least 64.	36
Adv	ice to users. Under circumsta	nces of store exhaustion an attempt to put a name	37
		the value of MPI_MAX_OBJECT_NAME should be	38
		d on the name length, not a guarantee that setting	39
nam	es of less than this length will	always succeed. (End of advice to users.)	40
A 1	· , · , , , , , , , , , , , , , , , , ,		41 42
		entations which pre-allocate a fixed size space for a allocation as the value of MPI_MAX_OBJECT_NAME.	43
	_	anocation as the value of MFT_MAA_OBJECT_NAME.	44
-	-	relatively small value, since the user has to allocate	45
		e when calling MPI_COMM_GET_NAME. (End of	46
advi	ce to implementors.)		47
			48

Unofficial Draft for Comment Only

MPI_COMM_GET_NAME(comm, comm_name, resultlen)

2	-		· · · · · · · · · · · · · · · · · · ·			
3	IN	comm	communicator whose name is to be returned (handle)			
4	OUT	comm_name	the name previously stored on the communicator, or			
5			an empty string if no such name exists (string)			
6	OUT	resultlen	length of returned name (integer)			
7						
8	C binding					
9 10	int MPI_Co	omm_get_name(MPI_Comm com	m, char *comm_name, int *resultlen)			
11	Fortran 20	008 binding				
12		get_name(comm, comm_name,				
13		<pre>IPI_Comm), INTENT(IN) ::</pre>				
14			AME), INTENT(OUT) :: comm_name			
15		ER, INTENT(OUT) :: result				
16	INIEGE	ER, OPTIONAL, INTENT(OUT)	:: lerror			
17	Fortran bi	inding				
18 19	MPI_COMM_C	GET_NAME(COMM, COMM_NAME,	RESULTLEN, IERROR)			
20		ER COMM, RESULTLEN, IERRO	R			
20	CHARAC	CTER*(*) COMM_NAME				
22	MPI_C	OMM_GET_NAME returns th	he last name which has previously been associated			
23			may be set and retrieved from any language. The			
24	same name	will be returned independent	t of the language used. name should be allocated			
25	so that it c	an hold a resulting string of I	length $MPI_MAX_OBJECT_NAME$ characters.			
26	MPI_COMM_GET_NAME returns a copy of the set name in name.					
27	In C, a null character is additionally stored at $name[resultlen]$. The value of $resultlen$					
28			T_NAME-1. In Fortran, name is padded on the			
29	right with blank characters. The value of resultlen cannot be larger than					
30		DBJECT_NAME.				
31 32			ne with a communicator, or an error occurs, empty string (all spaces in Fortran, "" in C). The			
33			we predefined names associated with them. Thus,			
34			COMM_SELF, and the communicator returned by			
35			_COMM_NULL) will have the default of			
36			nd MPI_COMM_PARENT. The fact that the system			
37			to a communicator does not prevent the user from			
38	setting a na	ame on the same communicat	tor; doing this removes the old name and assigns			
39	the new one	e.				
40	D					
41	Ratio		nctions for setting and getting the name of a com-			
42			viding a predefined attribute key for the following			
43	reasor	18:				
44	• I	t is not, in general, possible t	o store a string as an attribute from Fortran.			
45 46			te function for a string attribute unless it is known			
40		to have been allocated from the	0			
48			*			

- To make the attribute key useful additional code to call strdup is necessary. If this is not standardized then users have to write it. This is extra unneeded work which we can easily eliminate.
- The Fortran binding is not trivial to write (it will depend on details of the Fortran compilation system), and will not be portable. Therefore it should be in the library rather than in user code.

(End of rationale.)

Advice to users. The above definition means that it is safe simply to print the string returned by MPI_COMM_GET_NAME, as it is always a valid string even if there was no name.

Note that associating a name with a communicator has no effect on the semantics of an MPI program, and will (necessarily) increase the store requirement of the program, since the names must be saved. Therefore there is no requirement that users use these functions to associate names with communicators. However debugging and profiling MPI applications may be made easier if names are associated with communicators, since the debugger or profiler should then be able to present information in a less cryptic manner. (*End of advice to users.*)

The following functions are used for setting and getting names of datatypes. The constant MPI_MAX_OBJECT_NAME also applies to these names.

MPI_TYPE_SET_NAME(datatype, type_name) 2526INOUT datatype datatype whose identifier is to be set (handle) 27IN type_name the character string which is remembered as the 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) 35TYPE(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) 40INTEGER DATATYPE, IERROR 41 CHARACTER*(*) TYPE_NAME 4243

1

2

3

4

5

6

7

8

10

11

12 13

14

15

16

17

18

19

20 21

22

23 24

326 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

1 MPI_TYPE_GET_NAME(datatype, type_name, resultlen) 2 IN datatype datatype whose name is to be returned (handle) 3 OUT type_name the name previously stored on the datatype, or an 4 empty string if no such name exists (string) 56 OUT resultlen length of returned name (integer) 7 8 C binding 9 int MPI_Type_get_name(MPI_Datatype datatype, char *type_name, 10 int *resultlen) 11 Fortran 2008 binding 12MPI_Type_get_name(datatype, type_name, resultlen, ierror) 13 TYPE(MPI_Datatype), INTENT(IN) :: datatype 14CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: type_name 15INTEGER, INTENT(OUT) :: resultlen 16INTEGER, OPTIONAL, INTENT(OUT) :: ierror 17 18 Fortran binding 19MPI_TYPE_GET_NAME(DATATYPE, TYPE_NAME, RESULTLEN, IERROR) 20INTEGER DATATYPE, RESULTLEN, IERROR 21CHARACTER*(*) TYPE_NAME 22 Named predefined datatypes have the default names of the datatype name. For exam-23ple, MPI_WCHAR has the default name of MPI_WCHAR. 24The following functions are used for setting and getting names of windows. The con-25stant MPI_MAX_OBJECT_NAME also applies to these names. 262728MPI_WIN_SET_NAME(win, win_name) 29 INOUT window whose identifier is to be set (handle) 30 win 31IN the character string which is remembered as the win_name 32 name (string) 33 34C binding 35 int MPI_Win_set_name(MPI_Win win, const char *win_name) 36 37 Fortran 2008 binding 38 MPI_Win_set_name(win, win_name, ierror) TYPE(MPI_Win), INTENT(IN) :: win 39 40 CHARACTER(LEN=*), INTENT(IN) :: win_name 41 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 42Fortran binding 43 MPI_WIN_SET_NAME(WIN, WIN_NAME, IERROR) 44 INTEGER WIN, IERROR 45 CHARACTER*(*) WIN_NAME 46 47 48

MPI_WIN_GET_NAME(win, win_name, resultlen)				
IN	win	window whose name is to be returned (handle)		
OUT	win_name	the name previously stored on the window, or an empty string if no such name exists (string)		
OUT	resultlen	length of returned name (integer)		

C binding

int MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)

Fortran 2008 binding

```
MPI_Win_get_name(win, win_name, resultlen, ierror)
   TYPE(MPI_Win), INTENT(IN) :: win
   CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: win_name
   INTEGER, INTENT(OUT) :: resultlen
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_WIN_GET_NAME(WIN, WIN_NAME, RESULTLEN, IERROR)
INTEGER WIN, RESULTLEN, IERROR
CHARACTER*(*) WIN_NAME
```

6.9 Formalizing the Loosely Synchronous Model

In this section, we make further statements about the loosely synchronous model, with particular attention to intra-communication.

6.9.1 Basic Statements

When a caller passes a communicator (that contains a context and group) to a callee, that communicator must be free of side effects throughout execution of the subprogram: there should be no active operations on that communicator that might involve the process. This provides one model in which libraries can be written, and work "safely." For libraries so designated, the callee has permission to do whatever communication it likes with the communicator, and under the above guarantee knows that no other communicators will interfere. Since we permit good implementations to create new communicators without synchronization (such as by preallocated contexts on communicators), this does not impose a significant overhead.

This form of safety is analogous to other common computer-science usages, such as passing a descriptor of an array to a library routine. The library routine has every right to expect such a descriptor to be valid and modifiable.

6.9.2 Models of Execution

In the loosely synchronous model, transfer of control to a **parallel procedure** is effected by having each executing process invoke the procedure. The invocation is a collective operation: it is executed by all processes in the execution group, and invocations are similarly ordered at all processes. However, the invocation need not be synchronized.

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 44

45

46

47

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.

6 7

8

Static Communicator Allocation

⁹ This covers the case where, at any point in time, at most one invocation of a parallel ¹⁰ procedure can be active at any process, and the group of executing processes is fixed. For ¹¹ example, all invocations of parallel procedures involve all processes, processes are single-¹² 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.

18

¹⁹ 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).
- 40 41 42

34

35

36 37

38

39

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

Chapter 7

Process Topologies

7.1 Introduction

This chapter discusses the MPI topology mechanism. A topology is an extra, optional attribute that one can give to an intra-communicator; topologies cannot be added to 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 6, a process group in MPI is a collection of n processes. Each process in the group is assigned a rank between 0 and n-1. In many parallel applications a linear ranking of processes does not adequately reflect the logical communication pattern of the processes (which is usually determined by the underlying problem geometry and the numerical algorithm used). Often the processes are arranged in topological patterns such as two- or three-dimensional grids. More generally, the logical process arrangement is described by a graph. In this chapter we will refer to this logical process arrangement as the "virtual topology."

A clear distinction must be made between the virtual process topology and the topology of the underlying, physical hardware. The virtual topology can be exploited by the system in the assignment of processes to physical processors, if this helps to improve the communication performance on a given machine. How this mapping is done, however, is outside the scope of MPI. The description of the virtual topology, on the other hand, depends only on the application, and is machine-independent. The functions that are described in this chapter deal with machine-independent mapping and communication on virtual process topologies.

Rationale. Though physical mapping is not discussed, the existence of the virtual topology information may be used as advice by the runtime system. There are well-known techniques for mapping grid/torus structures to hardware topologies such as hypercubes or grids. For more complicated graph structures good heuristics often yield nearly optimal results [46]. 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

Besides possible performance benefits, the virtual topology can function as a convenient, process-naming structure, with significant benefits for program readability and notational power in message-passing programming. (*End of rationale.*)

7.2 Virtual Topologies

The communication pattern of a set of processes can be represented by a graph. The nodes represent processes, and the edges connect processes that communicate with each other. MPI provides message-passing between any pair of processes in a group. There is no requirement for opening a channel explicitly. Therefore, a "missing link" in the user-defined process graph does not prevent the corresponding processes from exchanging messages. It means rather that this connection is neglected in the virtual topology. This strategy implies that the topology gives no convenient way of naming this pathway of communication. Another possible consequence is that an automatic mapping tool (if one exists for the runtime environment) will not take account of this edge when mapping.

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-19tion 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 (2×2) grid is as follows.

coord $(0,0)$:	rank 0
coord $(0,1)$:	rank 1
coord $(1,0)$:	rank 2
coord $(1,1)$:	$\operatorname{rank} 3$

7.3 Embedding in MPI

The support for virtual topologies as defined in this chapter is consistent with other parts of MPI, and, whenever possible, makes use of functions that are defined elsewhere. Topology information is associated with communicators. It is added to communicators using the caching mechanism described in Chapter 6.

7.4 Overview of the Functions

⁴³ MPI supports three topology types: **Cartesian**, **graph**, and **distributed graph**. The ⁴⁴ function MPI_CART_CREATE is used to create Cartesian topologies, the function

⁴⁶ MPI_GRAPH_CREATE is used to create graph topologies, and the functions

⁴⁰ MPI_DIST_GRAPH_CREATE_ADJACENT and MPI_DIST_GRAPH_CREATE are used to cre-

 $_{48}$ ate distributed graph topologies. These topology creation functions are collective. As with

1

2

3

4 5

> 6 7

> > 8

9

10

11

12

13

14

15

25

26

27

35 36

37

38

39

40 41

other collective calls, the program must be written to work correctly, whether the call synchronizes or not.

The topology creation functions take as input an existing communicator comm_old, which defines the set of processes on which the topology is to be mapped. For MPI_GRAPH_CREATE and MPI_CART_CREATE, all input arguments must have identical values on all processes of the group of comm_old. When calling MPI_GRAPH_CREATE, each process specifies all nodes and edges in the graph. In contrast, the functions MPI_DIST_GRAPH_CREATE_ADJACENT or MPI_DIST_GRAPH_CREATE are used to specify the graph in a distributed fashion, whereby each process only specifies a subset of the edges in the graph such that the entire graph structure is defined collectively across the set of processes. Therefore the processes provide different values for the arguments specifying the graph. However, all processes must give the same value for reorder and the info argument. In all cases, a new communicator comm_topol is created that carries the topological structure as cached information (see Chapter 6). In analogy to function MPI_COMM_CREATE, no cached information propagates from **comm_old** to **comm_topol**.

MPI_CART_CREATE can be used to describe Cartesian structures of arbitrary dimension. For each coordinate direction one specifies whether the process structure is periodic or not. Note that an *n*-dimensional hypercube is an *n*-dimensional torus with 2 processes per coordinate direction. Thus, special support for hypercube structures is not necessary. The local auxiliary function MPI_DIMS_CREATE can be used to compute a balanced distribution of processes among a given number of dimensions.

MPI defines functions to query a communicator for topology information. The function MPI_TOPO_TEST is used to query for the type of topology associated with a communicator. Depending on the topology type, different information can be extracted. For a graph topology, the functions MPI_GRAPHDIMS_GET and MPI_GRAPH_GET return the values that were specified in the call to MPI_GRAPH_CREATE. Additionally, the functions MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS can be used to obtain the neighbors of an arbitrary node in the graph. For a distributed graph topology, the functions MPI_DIST_GRAPH_NEIGHBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS can be used to obtain the neighbors of the calling process. For a Cartesian topology, the functions MPI_CARTDIM_GET and MPI_CART_GET return the values that were specified in the call to MPI_CART_CREATE. Additionally, the functions MPI_CART_RANK and MPI_CART_COORDS translate Cartesian coordinates into a group rank, and vice-versa. 34The function MPI_CART_SHIFT provides the information needed to communicate with neighbors along a Cartesian dimension. All of these query functions are local.

For Cartesian topologies, the function MPI_CART_SUB can be used to extract a Cartesian subspace (analogous to MPI_COMM_SPLIT). This function is collective over the input communicator's group.

The two additional functions, MPI_GRAPH_MAP and MPI_CART_MAP, are, in general, not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 6, they are sufficient to implement all other topology functions. Section 7.5.8 outlines such an implementation.

The neighborhood collective communication routines MPI_NEIGHBOR_ALLGATHER, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL,

MPI_NEIGHBOR_ALLTOALLV, and MPI_NEIGHBOR_ALLTOALLW communicate with the 4546nearest neighbors on the topology associated with the communicator. The nonblocking 47variants are MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV, 48 MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6 $\overline{7}$

8 9

10 11

12

13

14

15

16

17

18

1920

21

22

23 24

2526

2728

2930

 31

32 33

35

36 37

38

39

40 41

42

43

1 2	MPI_INEIGHBOR_ALLTOALLW.			
3 4	7.5 Topology Constructors			
5 6 7	7.5.1 Car	tesian Constructor		
8 9	MPI_CART	_CREATE(comm_old, ndims,	dims, periods, reorder, comm_cart)	
10	IN	comm_old	input communicator (handle)	
11	IN	ndims	number of dimensions of Cartesian grid (integer)	
12 13 14	IN	dims	integer array of size ndims specifying the number of processes in each dimension	
15 16	IN	periods	logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension	
17 18	IN	reorder	ranking may be reordered (true) or not (false) (logical)	
19 20	OUT	comm_cart	communicator with new Cartesian topology (handle)	
23 24 25 26 27 28 29 30 31 32	Fortran 2 MPI_Cart_ TYPE(INTEG LOGIC TYPE(<pre>const int periods[], 008 binding</pre>	dims(ndims) s(ndims), reorder comm_cart	
33 34 35 36	INTEG	J	DIMS, PERIODS, REORDER, COMM_CART, IERROR) (*), COMM_CART, IERROR	
 37 38 39 40 41 42 43 44 45 46 47 48 	topology in new group the process the physica the group of MPI_COM	nformation is attached. If rec is identical to its rank in the ses (possibly so as to choose al machine). If the total size of comm_old, then some proce M_SPLIT. If ndims is zero ther	lle to a new communicator to which the Cartesian order = false then the rank of each process in the e old group. Otherwise, the function may reorder a good embedding of the virtual topology onto of the Cartesian grid is smaller than the size of esses are returned MPI_COMM_NULL, in analogy to a zero-dimensional Cartesian topology is created. d that is larger than the group size or if ndims is	

7.5.2 Cartesian Convenien	nce Function: MPI_DIMS_CREATE	1
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 <i>n</i> -dimensional topology.		
MPI_DIMS_CREATE(nnodes	s. ndims. dims)	8 9
IN nnodes	number of nodes in a grid (integer)	10
IN ndims	number of Cartesian dimensions (integer)	11 12
INOUT dims	integer array of size ndims specifying the number of nodes in each dimension	13 14 15
		16
C binding int MPI Dims create(int	nnodes, int ndims, int dims[])	17
		18 19
Fortran 2008 binding MPI_Dims_create(nnodes, ndims, dims, ierror)		
INTEGER, INTENT(IN) :: nnodes, ndims		
INTEGER, INTENT(INOUT) :: dims(ndims)		
INTEGER, OPTIONAL,	INTENT(OUT) :: ierror	23
Fortran binding		24 25
MPI_DIMS_CREATE(NNODES,		26
INTEGER NNODES, NDI	MS, DIMS(*), IERROR	27
The entries in the array	dims are set to describe a Cartesian grid with ndims dimensions	28
and a total of nnodes nodes. The dimensions are set to be as close to each other as possible,		
	ility algorithm. The caller may further constrain the operation	30 31
	g elements of array dims. If dims[i] is set to a positive number,	32
dims[i] = 0 are modified by	the number of nodes in dimension i; only those entries where the call	33
	of dims[i] are erroneous. An error will occur if nnodes is not a	34
multiple of		35
	$\prod dims[i].$	36
	$i,dims[i]{ eq}0$	37
For dims[i] set by the c	all, dims[i] will be ordered in non-increasing order. Array dims	38 39
-	to routine MPI_CART_CREATE. MPI_DIMS_CREATE is local.	39 40
If ndims is zero and nnodes	is one, MPI_DIMS_CREATE returns MPI_SUCCESS.	41
Example 7.1		

Example 7.1

Unofficial Draft for Comment Only

```
function call
                                                                 dims
1
                   dims
2
                   before call
                                                                 on return
3
                   (0,0)
                                MPI_DIMS_CREATE(6, 2, dims)
                                                                 (3,2)
                                MPI_DIMS_CREATE(7, 2, dims)
4
                   (0,0)
                                                                 (7,1)
                                MPI_DIMS_CREATE(6, 3, dims)
5
                    (0,3,0)
                                                                 (2,3,1)
6
                   (0,3,0)
                                MPI_DIMS_CREATE(7, 3, dims)
                                                                 erroneous call
7
8
            Graph Constructor
     7.5.3
9
10
11
     MPI_GRAPH_CREATE(comm_old, nnodes, index, edges, reorder, comm_graph)
12
       IN
                 comm_old
                                             input communicator (handle)
13
14
       IN
                 nnodes
                                             number of nodes in graph (integer)
15
                 index
                                             array of integers describing node degrees (see below)
       IN
16
                 edges
                                             array of integers describing graph edges (see below)
       IN
17
18
       IN
                 reorder
                                             ranking may be reordered (true) or not (false)
19
                                             (logical)
20
       OUT
                 comm_graph
                                             communicator with graph topology added (handle)
21
22
     C binding
23
     int MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int index[],
24
                     const int edges[], int reorder, MPI_Comm *comm_graph)
25
26
     Fortran 2008 binding
27
     MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph,
28
                     ierror)
29
          TYPE(MPI_Comm), INTENT(IN) :: comm_old
30
          INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
31
          LOGICAL, INTENT(IN) :: reorder
32
          TYPE(MPI_Comm), INTENT(OUT) :: comm_graph
33
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
     Fortran binding
35
     MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH,
36
                     IERROR)
37
          INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR
38
          LOGICAL REORDER
39
40
          MPI_GRAPH_CREATE returns a handle to a new communicator to which the graph
41
     topology information is attached. If reorder = false then the rank of each process in the
42
     new group is identical to its rank in the old group. Otherwise, the function may reorder the
43
     processes. If the size, nnodes, of the graph is smaller than the size of the group of comm_old,
44
     then some processes are returned MPI_COMM_NULL, in analogy to MPI_CART_CREATE
45
     and MPI_COMM_SPLIT. If the graph is empty, i.e., nnodes == 0, then MPI_COMM_NULL
46
     is returned in all processes. The call is erroneous if it specifies a graph that is larger than
47
     the group size of the input communicator.
48
```

The three parameters nnodes, index and edges define the graph structure. nnodes is the number of nodes of the graph. The nodes are numbered from 0 to nnodes-1. The i-th entry of array index stores the total number of neighbors of the first i graph nodes. The lists of neighbors of nodes 0, 1, ..., nnodes-1 are stored in consecutive locations in array edges. The array edges is a flattened representation of the edge lists. The total number of entries in index is nnodes and the total number of entries in edges is equal to the number of graph edges.

The definitions of the arguments nnodes, index, and edges are illustrated with the following simple example.

Example 7.2 Assume there are four processes 0, 1, 2, 3 with the following adjacency matrix:

process	neighbors
0	1, 3
1	0
2	3
3	0, 2

Then, the input arguments are:

$$\begin{array}{ll} \text{nnodes} = & 4 \\ \text{index} = & 2, \, 3, \, 4, \, 6 \\ \text{edges} = & 1, \, 3, \, 0, \, 3, \, 0, \, 2 \end{array}$$

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:

- Type of topology (Cartesian/graph),
 For a Cartesian topology:
 1. ndims (number of dimensions),
 - 2. dims (numbers of processes per coordinate direction),

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\overline{7}$

1	3. periods (periodicity information),
2 3	4. own_position (own position in grid, could also be computed from rank and dims)
4	• For a graph topology:
5	1. index,
6 7	2. edges,
8	which are the vectors defining the graph structure.
9	which are the vectors denning the graph structure.
10	For a graph structure the number of nodes is equal to the number of processes in
11	the group. Therefore, the number of nodes does not have to be stored explicitly. An additional zero entry at the start of array index simplifies access to the topology
12 13	information. (End of advice to implementors.)
14	
15	7.5.4 Distributed Graph Constructor
16 17 18 19 20 21 22 23 24 25 26 27 28 29 30	MPI_GRAPH_CREATE requires that each process passes the full (global) communication graph to the call. This limits the scalability of this constructor. With the distributed graph interface, the communication graph is specified in a fully distributed fashion. Each process specifies only the part of the communication graph of which it is aware. Typically, this could be the set of processes from which the process will eventually receive or get data, or the set of processes to which the process will send or put data, or some combination of such edges. Two different interfaces can be used to create a distributed graph topology. MPI_DIST_GRAPH_CREATE_ADJACENT creates a distributed graph communicator with each process specifying each of its incoming and outgoing (adjacent) edges in the logical communication graph and thus requires minimal communication during creation. MPI_DIST_GRAPH_CREATE provides full flexibility such that any process can indicate that communication will occur between any pair of processes in the graph. To provide better possibilities for optimization by the MPI library, the distributed graph constructors permit weighted communication edges and take an info argument that
31 32	can further influence process reordering or other optimizations performed by the MPI library. For example, hints can be provided on how edge weights are to be interpreted, the quality
32 33	of the reordering, and/or the time permitted for the MPI library to process the graph.
34	
35	
36	
37 38	
39	
40	*

MPI_DIST_GRAPH_CREATE_ADJACENT(comm_old, indegree, sources, sourceweights, outdegree, destinations, destweights, info, reorder, comm_dist_graph)

	outdegree, destinations, d	cstweights, 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)	11 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
<pre>INTEGER, INTENT(IN) :: indegree, sources(indegree), sourceweights(*),</pre>	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,	45
OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER,	46

COMM_DIST_GRAPH, IERROR)

INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE, DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR LOGICAL REORDER

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.4. (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
IN	sources	array containing the n source nodes for which this	10
	3001003	process specifies edges (array of non-negative	11 12
		integers)	12
IN	dogroop		14
IIN	degrees	array specifying the number of destinations for each source node in the source node array (array of	15
		non-negative integers)	16
	destinations		17
IN	destinations	destination nodes for the source nodes in the source node array (array of non-negative integers)	18
			19
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 25
		(logical)	25 26
OUT	comm_dist_graph	communicator with distributed graph topology	27
		added (handle)	28
			29
C binding	g		30
int MPI_D	ist_graph_create(MPI_Comm	n comm_old, int n, const int sources[],	31
		<pre>const int destinations[],</pre>	32
		MPI_Info info, int reorder,	33
	MPI_Comm *comm_dist_	graph)	34
Fortran 2	008 binding		35
	J	, sources, degrees, destinations, weights,	36
	info, reorder, comm_	dist_graph, ierror)	37 38
TYPE(MPI_Comm), INTENT(IN) ::	comm_old	39
INTEG		<pre>rces(n), degrees(n), destinations(*),</pre>	40
	weights(*)		41
	<pre>MPI_Info), INTENT(IN) ::</pre>		42
	AL, INTENT(IN) :: reorder		43
	MPI_Comm), INTENT(OUT) :		44
LINIEG	ER, OPTIONAL, INTENT(OUT)	/ ICIIOI	45
Fortran b	inding		46
MPI_DIST_	GRAPH_CREATE(COMM_OLD, N	, SOURCES, DEGREES, DESTINATIONS, WEIGHTS,	47

INFO, REORDER, COMM_DIST_GRAPH, IERROR)

INTEGER COMM_OLD, N, SOURCES(*), DEGREES(*), DESTINATIONS(*), WEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR

LOGICAL REORDER

MPI_DIST_GRAPH_CREATE returns a handle to a new communicator to which the 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

3

1

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.4. (*End of rationale.*)

The meaning of the weights argument can be influenced by the info argument. Info arguments can be used to guide the mapping; possible options include minimizing the maximum number of edges between processes on different SMP nodes, or minimizing the sum of all such edges. An MPI implementation is not obliged to follow specific hints, and it is valid for an MPI implementation not to do any reordering. An MPI implementation may specify more info key-value pairs. All processes must specify the same set of key-value info pairs.

Advice to implementors. MPI implementations must document any additionally supported key-value info pairs. MPI_INFO_NULL is always valid, and may indicate the default creation of the distributed graph topology to the MPI library.

An implementation does not explicitly need to construct the topology from its distributed parts. However, all processes can construct the full topology from the distributed specification and use this in a call to MPI_GRAPH_CREATE to create the topology. This may serve as a reference implementation of the functionality, and may be acceptable for small communicators. However, a scalable high-quality implementation would save the topology graph in a distributed way. (*End of advice to implementors.*)

Example 7.3 As for Example 7.2, assume there are four processes 0, 1, 2, 3 with the following adjacency matrix and unit edge weights:

process	neighbors
0	1, 3
1	0
2	3
3	0, 2

With MPI_DIST_GRAPH_CREATE, this graph could be constructed in many different ways. One way would be that each process specifies its outgoing edges. The arguments per process would be:

process	n	sources	degrees	destinations	weights
0	1	0	2	1,3	1,1
1	1	1	1	0	1
2	1	2	1	3	1
3	1	3	2	0,2	1,1

Another way would be to pass the whole graph on process 0, which could be done with the following arguments per process:

process	n	sources	degrees	destinations	weights
0	4	0,1,2,3	2,1,1,2	1,3,0,3,0,2	$1,\!1,\!1,\!1,\!1,\!1,\!1$
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 7.4 A two-dimensional PxQ torus where all processes communicate along the dimensions and along the diagonal edges. This cannot be modeled with Cartesian topologies, but can easily be captured with MPI_DIST_GRAPH_CREATE as shown in the following code. In this example, the communication along the dimensions is twice as heavy as the communication along the diagonals:

```
/*
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;
35
     destinations[1] = P*y+(P+x-1)%P; weights[1] = 2;
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		
7.5.5 To	pology Inquiry Functions		4		
-	gy has been defined with one obtained 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 Fopo_test(MPI_Comm comm,	int setatus)	15		
	-	Int "Status)	16		
	2008 binding _test(comm, status, ierr	or)	17 18		
-	(MPI_Comm), INTENT(IN) :		19		
	GER, INTENT(OUT) :: stat		20		
	GER, OPTIONAL, INTENT(OU	1) :: lerror	21 22		
Fortran	6		23		
	_TEST(COMM, STATUS, IERR 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		
The o	output value status is one of t	the following:	28		
MPI_GR	АРН	graph topology	29		
MPI_CA		Cartesian topology	30 31		
	T_GRAPH	distributed graph topology	32		
MPI_UN	DEFINED	no topology	33		
			34		
MPI_GRA	PHDIMS_GET(comm, nnodes	s, nedges)	$\frac{35}{36}$		
IN	comm	communicator for group with graph structure	37		
		(handle)	38		
OUT	nnodes	number of nodes in graph (same as number of	39 40		
		processes in the group) (integer)	41		
OUT	nedges	number of edges in graph (integer)	42		
Chindin	a.		$\frac{43}{44}$		
C binding int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges) 45					
46					
	Fortran 2008 binding MPI_Graphdims_get(comm, nnodes, nedges, ierror) 47				
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
         Functions MPI_GRAPHDIMS_GET and MPI_GRAPH_GET retrieve the graph-topology
9
     information that was associated with a communicator by MPI_GRAPH_CREATE.
10
         The information provided by MPI_GRAPHDIMS_GET can be used to dimension the
11
     vectors index and edges correctly for 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
       IN
                 maxedges
                                             length of vector edges in the calling program (integer)
18
19
       OUT
                 index
                                             array of integers containing the graph structure (for
20
                                             details see the definition of MPI_GRAPH_CREATE)
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
27
     Fortran 2008 binding
28
     MPI_Graph_get(comm, maxindex, maxedges, index, edges, ierror)
29
          TYPE(MPI_Comm), INTENT(IN) :: comm
30
          INTEGER, INTENT(IN) :: maxindex, maxedges
^{31}
          INTEGER, INTENT(OUT) :: index(maxindex), edges(maxedges)
32
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
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
                                             communicator with Cartesian structure (handle)
                 comm
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
```

```
1
MPI_Cartdim_get(comm, ndims, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                          2
    INTEGER, INTENT(OUT) :: ndims
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                          5
Fortran binding
                                                                                          6
MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
    INTEGER COMM, NDIMS, IERROR
                                                                                          9
    The functions MPI_CARTDIM_GET and MPI_CART_GET return the Cartesian topol-
                                                                                          10
ogy information that was associated with a communicator by MPI_CART_CREATE. If comm
                                                                                          11
is associated with a zero-dimensional Cartesian topology, MPI_CARTDIM_GET returns
ndims = 0 and MPI_CART_GET will keep all output arguments unchanged.
                                                                                          12
                                                                                          13
                                                                                          14
MPI_CART_GET(comm, maxdims, dims, periods, coords)
                                                                                          15
                                                                                          16
  IN
                                       communicator with Cartesian structure (handle)
           comm
                                                                                          17
                                       length of vectors dims, periods, and coords in the
  IN
            maxdims
                                                                                          18
                                       calling program (integer)
                                                                                          19
  OUT
           dims
                                       number of processes for each Cartesian dimension
                                                                                          20
                                        (array of integers)
                                                                                         21
                                                                                          22
  OUT
            periods
                                       periodicity (true/false) for each Cartesian dimension
                                                                                          23
                                        (array of logicals)
                                                                                          24
  OUT
           coords
                                       coordinates of calling process in Cartesian structure
                                                                                          25
                                        (array of integers)
                                                                                          26
                                                                                          27
C binding
                                                                                          28
int MPI_Cart_get(MPI_Comm comm, int maxdims, int dims[], int periods[],
                                                                                          29
               int coords[])
                                                                                          30
                                                                                          31
Fortran 2008 binding
                                                                                          32
MPI_Cart_get(comm, maxdims, dims, periods, coords, ierror)
                                                                                          33
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                          34
   INTEGER, INTENT(IN) :: maxdims
                                                                                          35
    INTEGER, INTENT(OUT) :: dims(maxdims), coords(maxdims)
                                                                                          36
    LOGICAL, INTENT(OUT) :: periods(maxdims)
                                                                                          37
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                          38
Fortran binding
                                                                                          39
MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
                                                                                          40
    INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
                                                                                          41
    LOGICAL PERIODS(*)
                                                                                          42
                                                                                          43
                                                                                          44
                                                                                          45
                                                                                          46
                                                                                          47
```

```
1
     MPI_CART_RANK(comm, coords, rank)
2
       IN
                                               communicator with Cartesian structure (handle)
                  comm
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 process group with Cartesian structure, the function MPI_CART_RANK trans-
21
     lates the logical process coordinates to process ranks as they are used by the point-to-point
22
      routines.
23
          For dimension i with periods(i) = true, if the coordinate, coords(i), is out of range, that
24
      is, coords(i) < 0 or coords(i) \ge dims(i), it is shifted back to the interval
25
      0 \leq coords(i) < dims(i) automatically. Out-of-range coordinates are erroneous for non-
26
      periodic dimensions.
27
          If comm is associated with a zero-dimensional Cartesian topology, coords is not signif-
28
      icant and 0 is returned in rank.
29
30
^{31}
     MPI_CART_COORDS(comm, rank, maxdims, coords)
32
       IN
                                               communicator with Cartesian structure (handle)
33
                  comm
34
       IN
                  rank
                                               rank of a process within group of comm (integer)
35
       IN
                  maxdims
                                               length of vector coords in the calling program
36
                                               (integer)
37
       OUT
                                               integer array (of size maxdims) containing the
38
                  coords
                                               Cartesian coordinates of specified process (array of
39
40
                                               integers)
41
42
     C binding
43
      int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])
44
      Fortran 2008 binding
45
      MPI_Cart_coords(comm, rank, maxdims, coords, ierror)
46
          TYPE(MPI_Comm), INTENT(IN) :: comm
47
          INTEGER, INTENT(IN) :: rank, maxdims
48
```

INTEGER, INTENT(OUT) :: coords(maxdims) INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
Fortran	0		4		
	_COORDS(COMM, RANK, MAXDI		5		
LNIE	GER COMM, RANK, MAXDIMS,	CUURDS(*), IERRUR	6		
The	inverse mapping, rank-to-coor	rdinates translation is provided by	7		
	T_COORDS.		8		
		-dimensional Cartesian topology,	9 10		
coords will be unchanged.					
			11 12		
MPI_GRA	PH_NEIGHBORS_COUNT(cor	nm, rank, nneighbors)	12		
IN	comm	communicator with graph topology (handle)	14		
IN	rank	rank of process in group of comm (integer)	15		
OUT	nneighbors	number of neighbors of specified process (integer)	16 17		
	0		18		
C bindin	1g		19		
	0	_Comm comm, int rank, int *nneighbors)	20		
D			21		
	2008 binding h_neighbors_count(comm, r	ank projektore jerrer)	22		
-	(MPI_Comm), INTENT(IN) ::		23		
	GER, INTENT(IN) :: rank		24 25		
INTEGER, INTENT(IN) TAIK INTEGER, INTENT(OUT) :: nneighbors					
INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
Fortran binding					
	DINGING H_NEIGHBORS_COUNT(COMM, R	ANV MNETCUDADE TEDDAD)	28 29		
	GER COMM, RANK, NNEIGHBOR		30		
1010	delit obinit, itawa, aneroneoit		31		
			32		
MPI GRA	PH_NEIGHBORS(comm, rank,	maxneighbors, neighbors)	33		
IN		,	34		
	comm	communicator with graph topology (handle)	35		
IN	rank	rank of process in group of comm (integer)	36		
IN	maxneighbors	size of array neighbors (integer)	37 38		
OUT	neighbors	ranks of processes that are neighbors to specified	39		
		process (array of integers)	40		
			41		
C bindin	lg		42		
int MPI_	Graph_neighbors(MPI_Comm	comm, int rank, int maxneighbors,	43		
int neighbors[]) 4					
Fortran	2008 binding		45		
	6	axneighbors, neighbors, ierror)	46		
-	(MPI_Comm), INTENT(IN) ::	e	47		
48					

1	INTEGED INTENT(IN) mania maunaiathana				
2	INTEGER, INTENT(IN) :: rank, maxneighbors INTEGER, INTENT(OUT) :: neighbors(maxneighbors)				
3	INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
4	INTEGER, OFFICIAL, INTENT(COT) TETTOT				
5	Fortran binding				
6	MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)				
7	INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR				
8	MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS provide adjacency				
9	information for a graph topology. The returned count and array of neighbors for the queried				
10	rank will both include <i>all</i> neighbors and reflect the same edge ordering as was specified by				
11	the original call to MPI_GRAPH_CREATE. Specifically, MPI_GRAPH_NEIGHBORS_COUNT				
12	and MPI_GRAPH_NEIGHBORS will return values based on the original index and edges array				
13	passed to MPI_GRAPH_CREATE (for the purpose of this example, we assume that index[-1]				
14	is zero):				
15					
16	• The number of neighbors (nneighbors) returned from				
17	$MPI_GRAPH_NEIGHBORS_COUNT \text{ will be } (index[rank] - index[rank-1]).$				
18	• The neighbors array returned from MPI_GRAPH_NEIGHBORS will be edges[index[rank-				
19	1]] through edges[index[rank]-1].				
20					
21 22	Example 7.5 Assume there are four processes 0, 1, 2, 3 with the following adjacency				
22	matrix (note that some neighbors are listed multiple times):				
23	matrix (note that some neighbors are insted matriple times).				
25	process neighbors				
26	0 1, 1, 3				
27	1 0, 0				
28	$\begin{vmatrix} 2 \\ 3 \end{vmatrix}$				
29	$\begin{array}{ c c c c }\hline 3 & 0, 2, 2 \\\hline \end{array}$				
30	Thus, the input arguments to MPI_GRAPH_CREATE are:				
31					
32	$\begin{array}{rcl} \text{nnodes} = & 4 \\ \text{index} = & 3, 5, 6, 9 \end{array}$				
33	ddex = 3, 5, 6, 9 ddges = 1, 1, 3, 0, 0, 3, 0, 2, 2				
34					
35	Therefore, calling MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS for				
36	each of the 4 processes will return:				
37	Input rank Count Neighbors				
38 39	$\begin{array}{c c} \hline 0 \\ \hline 0 \\ \hline \end{array} \\ \hline 3 \\ \hline 1, 1, 3 \\ \hline \end{array}$				
40	1 2 0, 0				
41	2 1 3				
42	3 3 0, 2, 2				
43					
44	Example 7.6 Suppose that comm is a communicator with a shuffle-exchange topology.				
45	The group has 2^n members. Each process is labeled by a_1, \ldots, a_n with $a_i \in \{0, 1\}$, and has				
46	three neighbors: exchange $(a_1, \ldots, a_n) = a_1, \ldots, a_{n-1}, \bar{a}_n$ ($\bar{a} = 1 - a$), shuffle $(a_1, \ldots, a_n) = a_1, \ldots, a_{n-1}, \bar{a}_n$				
47	a_1, \ldots, a_n, a_1 , and $unshuffle(a_1, \ldots, a_n) = a_1, \ldots, a_{n-1}, a_n$ ($a = 1$ w), 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				
48	illustrated below for $n = 3$.				

Unofficial Draft for Comment Only

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.

! assume: each process has stored a real number A.	
! extract neighborhood information	
CALL MPI_COMM_RANK(comm, myrank, ierr)	
CALL MPI_GRAPH_NEIGHBORS(comm, myrank, 3, neighbors, ierr)	
! perform exchange permutation	
CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(1), 0, &	
<pre>neighbors(1), 0, comm, status, ierr)</pre>	
! perform shuffle permutation	
CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(2), 0, &	
neighbors(3), 0, comm, status, ierr)	
! perform unshuffle permutation	
CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(3), 0, &	
neighbors(2), 0, comm, status, ierr)	

```
MPI_DIST_GRAPH_NEIGHBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS pro-
vide adjacency information for a distributed graph topology.
```

MPI_DIST_GRAPH_NEIGHBORS_COUNT(comm, indegree, outdegree, weighted) ³³						
IN	comm	communicator with distributed graph topology	34			
		(handle)	35			
OUT	indegree	number of edges into this process (non-negative	36			
001	macgree	integer)	37 38			
OUT	outdegree	number of edges out of this process (non-negative	39			
001	outdegree	integer)	40			
OUT	weighted	false if MPI_UNWEIGHTED was supplied during	41			
		creation, true otherwise (logical)	42			
			43			
C hinding	•		44			

C binding int MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree, int *outdegree, int *weighted)

Fortran 2008 binding

 31

1 2 3 4 5 6	<pre>MPI_Dist_graph_neighbors_count(comm, indegree, outdegree, weighted, ierror) TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, INTENT(OUT) :: indegree, outdegree LOGICAL, INTENT(OUT) :: weighted INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>						
7 8 9 10 11	INTEG		T(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR) UTDEGREE, IERROR				
12 13 14	MPI_DIST		omm, maxindegree, sources, sourceweights, nations, destweights)				
15 16	IN	comm	communicator with distributed graph topology (handle)				
17 18 19	IN	maxindegree	size of sources and sourceweights arrays (non-negative integer)				
20 21	OUT	sources	processes for which the calling process is a destination (array of non-negative integers)				
22 23	OUT	sourceweights	weights of the edges into the calling process (array of non-negative integers)				
24 25 26	IN	maxoutdegree	size of destinations and destweights arrays (non-negative integer)				
20 27 28	OUT	destinations	processes for which the calling process is a source (array of non-negative integers)				
29 30	OUT	destweights	weights of the edges out of the calling process (array of non-negative integers)				
31 32 33 34 35	C binding int MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[], int sourceweights[], int maxoutdegree, int destinations[], int destweights[])						
36 37 38 39 40 41 42 43 44 45	<pre>Fortran 2008 binding MPI_Dist_graph_neighbors(comm, maxindegree, sources, sourceweights,</pre>						
46 47 48	Fortran binding MPI_DIST_GRAPH_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, SOURCEWEIGHTS, MAXOUTDEGREE, DESTINATIONS, DESTWEIGHTS, IERROR)						

INTEGER COMM, MAXINDEGREE, SOURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE, DESTINATIONS(*), DESTWEIGHTS(*), IERROR

These calls are local. The number of edges into and out of the process returned by MPI_DIST_GRAPH_NEIGHBORS_COUNT are the total number of such edges given in the call to MPI_DIST_GRAPH_CREATE_ADJACENT or MPI_DIST_GRAPH_CREATE (potentially by processes other than the calling process in the case of MPI_DIST_GRAPH_CREATE). Multiply defined edges are all counted and returned by MPI_DIST_GRAPH_NEIGHBORS in some order. If MPI_UNWEIGHTED is supplied for sourceweights or destweights or both, or if MPI_UNWEIGHTED was supplied during the construction of the graph then no weight information is returned in that array or those arrays. If the communicator was created with MPI_DIST_GRAPH_CREATE_ADJACENT then for each rank in comm, the order of the values in sources and destinations is identical to the input that was used by the process with the same rank in **comm_old** in the creation call. If the communicator was created with MPI_DIST_GRAPH_CREATE then the only requirement on the order of values in sources and destinations is that two calls to the routine with same input argument comm will return the same sequence of edges. If maxindegree or maxoutdegree is smaller than the numbers returned by MPI_DIST_GRAPH_NEIGHBORS_COUNT, then only the first part of the full list is returned.

Advice to implementors. Since the query calls are defined to be local, each process needs to store the list of its neighbors with incoming and outgoing edges. Communication is required at the collective MPI_DIST_GRAPH_CREATE call in order to compute the neighbor lists for each process from the distributed graph specification. (*End of advice to implementors.*)

7.5.6 Cartesian Shift Coordinates

If the process topology is a Cartesian structure, an MPI_SENDRECV operation may be used along a coordinate direction to perform a shift of data. As input, MPI_SENDRECV takes the rank of a source process for the receive, and the rank of a destination process for the send. If the function MPI_CART_SHIFT is called for a Cartesian process group, it provides the calling process with the above identifiers, which then can be passed to MPI_SENDRECV. The user specifies the coordinate direction and the size of the step (positive or negative). The function is local.

MPI_CART_SHIFT(comm, direction, disp, rank_source, rank_dest)			37
IN	comm	communicator with Cartesian structure (handle)	38
IN	direction	coordinate dimension of shift (integer)	39 40
IN	disp	displacement (> 0: upwards shift, < 0: downwards	41
		shift) (integer)	42
OUT	rank_source	rank of source process (integer)	43
OUT	rank_dest	rank of destination process (integer)	44
			$45 \\ 46$
C binding			
			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

```
1
     int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
\mathbf{2}
                     int *rank_source, int *rank_dest)
3
     Fortran 2008 binding
4
     MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror)
5
          TYPE(MPI_Comm), INTENT(IN) :: comm
6
          INTEGER, INTENT(IN) :: direction, disp
7
          INTEGER, INTENT(OUT) :: rank_source, rank_dest
8
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10
     Fortran binding
11
     MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)
12
          INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR
13
          The direction argument indicates the coordinate dimension to be traversed by the shift.
14
     The dimensions are numbered from 0 to ndims-1, where ndims is the number of dimensions.
15
          Depending on the periodicity of the Cartesian group in the specified coordinate direc-
16
     tion, MPI_CART_SHIFT provides the identifiers for a circular or an end-off shift. In the case
17
     of an end-off shift, the value MPI_PROC_NULL may be returned in rank_source or rank_dest,
18
     indicating that the source or the destination for the shift is out of range.
19
          It is erroneous to call MPI_CART_SHIFT with a direction that is either negative or
20
     greater than or equal to the number of dimensions in the Cartesian communicator. This
21
     implies that it is erroneous to call MPI_CART_SHIFT with a comm that is associated with
22
     a zero-dimensional Cartesian topology.
23
^{24}
     Example 7.7 The communicator, comm, has a two-dimensional, periodic, Cartesian topol-
25
     ogy associated with it. A two-dimensional array of REALs is stored one element per process,
26
     in variable A. One wishes to skew this array, by shifting column i (vertically, i.e., along the
27
     column) by i steps.
28
29
30
     ! find process rank
^{31}
     CALL MPI_COMM_RANK(comm, rank, ierr)
32
     ! find Cartesian coordinates
33
     CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
34
     ! compute shift source and destination
35
     CALL MPI_CART_SHIFT(comm, 0, coords(2), source, dest, ierr)
36
     ! skew array
37
     CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0, source, 0, comm, &
38
                                    status, ierr)
39
40
           Advice to users. In Fortran, the dimension indicated by DIRECTION = i has DIMS(i+1)
41
           nodes, where DIMS is the array that was used to create the grid. In C, the dimension
42
           indicated by direction = i is the dimension specified by dims[i]. (End of advice to users.)
43
44
45
46
47
48
```

7.5.7 Partitioning of Cartesian Structures

MPI_CART_SUB(d	comm, remain_dims, n	iewcomm)
----------------	----------------------	----------

IN remain_dims the i-th entry of remain_dims specifies whether the	
i-th dimension is kept in the subgrid (true) or is	
dropped (false) (array of logicals)	
OUT newcomm communicator containing the subgrid that include the calling process (handle)	\$

C binding

int MPI_Cart_sub(MPI_Comm comm, const int remain_dims[], MPI_Comm *newcomm)

Fortran 2008 binding

MPI_Cart_sub(comm, remain_dims, newcomm, ierror)
TYPE(MPI_Comm), INTENT(IN) :: comm
LOGICAL, INTENT(IN) :: remain_dims(*)
TYPE(MPI_Comm), INTENT(OUT) :: newcomm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

MPI_	_CART_SU	B(COMM,	REMAIN_	DIMS,	NEWCOMM,	IERROR)
	INTEGER	COMM,	NEWCOMM,	IERR	DR	
	LOGICAL	REMATN	DTMS(*)			r

If a Cartesian topology has been created with MPI_CART_CREATE, the function MPI_CART_SUB can be used to partition the communicator group into subgroups that form lower-dimensional Cartesian subgrids, and to build for each subgroup a communicator with the associated subgrid Cartesian topology. If all entries in remain_dims are false or comm is already associated with a zero-dimensional Cartesian topology. (This function is closely related to MPI_COMM_SPLIT.)

Example 7.8 Assume that MPI_CART_CREATE(..., comm) has defined a $(2 \times 3 \times 4)$ grid. Let remain_dims = (true, false, true). Then a call to

MPI_CART_SUB(comm, remain_dims, comm_new);

will create three communicators each with eight processes in a 2×4 Cartesian topology. If remain_dims = (false, false, true) then the call to MPI_CART_SUB(comm, remain_dims, comm_new) will create six non-overlapping communicators, each with four processes, in a one-dimensional Cartesian topology.

7.5.8 Low-Level Topology Functions

The two additional functions introduced in this section can be used to implement all other topology functions. In general they will not be called by the user directly, unless he or she

Unofficial Draft for Comment Only

```
1
      is creating additional virtual topology capability other than that provided by MPI. The two
\mathbf{2}
      calls are both local.
3
4
      MPI_CART_MAP(comm, ndims, dims, periods, newrank)
5
6
       IN
                 comm
                                               input communicator (handle)
7
       IN
                  ndims
                                               number of dimensions of Cartesian structure (integer)
8
                 dims
       IN
                                               integer array of size ndims specifying the number of
9
                                               processes in each coordinate direction
10
       IN
                  periods
                                               logical array of size ndims specifying the periodicity
11
                                               specification in each coordinate direction
12
13
        OUT
                                               reordered rank of the calling process;
                  newrank
14
                                               MPI_UNDEFINED if calling process does not belong
15
                                               to grid (integer)
16
17
      C binding
18
      int MPI_Cart_map(MPI_Comm comm, int ndims, const int dims[],
19
                     const int periods[], int *newrank)
20
21
      Fortran 2008 binding
      MPI_Cart_map(comm, ndims, dims, periods, newrank, ierror)
22
          TYPE(MPI_Comm), INTENT(IN) :: comm
23
          INTEGER, INTENT(IN) :: ndims, dims(ndims)
^{24}
          LOGICAL, INTENT(IN) :: periods(ndims)
25
26
          INTEGER, INTENT(OUT) :: newrank
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
      Fortran binding
29
     MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)
30
          INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR
^{31}
          LOGICAL PERIODS(*)
32
          MPI_CART_MAP computes an "optimal" placement for the calling process on the phys-
33
34
      ical machine. A possible implementation of this function is to always return the rank of the
      calling process, that is, not to perform any reordering.
35
36
           Advice to implementors.
                                       The function MPI_CART_CREATE(comm, ndims, dims,
37
           periods, reorder, comm_cart), with reorder = true can be implemented by calling
38
           MPI_CART_MAP(comm, ndims, dims, periods, newrank), then calling
39
           MPI_COMM_SPLIT(comm, color, key, comm_cart), with color = 0 if newrank \neq
40
           MPI_UNDEFINED, color = MPI_UNDEFINED otherwise, and key = newrank. If ndims
41
           is zero then a zero-dimensional Cartesian topology is created.
42
           The function MPI_CART_SUB(comm, remain_dims, comm_new) can be implemented
43
           by a call to MPI_COMM_SPLIT(comm, color, key, comm_new), using a single number
44
           encoding of the lost dimensions as color and a single number encoding of the preserved
45
           dimensions as key.
46
47
           All other Cartesian topology functions can be implemented locally, using the topology
48
           information that is cached with the communicator. (End of advice to implementors.)
```

The corresponding function for graph structures is as follows. MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank) IN comm input communicator (handle) IN nnodes number of graph nodes (integer) IN index integer array specifying the graph structure, see MPI_GRAPH_CREATE 10 IN edges integer array specifying the graph structure 11 OUT newrank reordered rank of the calling process; 12MPI_UNDEFINED if the calling process does not 13 belong to graph (integer) 1415C binding 16int MPI_Graph_map(MPI_Comm comm, int nnodes, const int index[], 17 const int edges[], int *newrank) 18 19 Fortran 2008 binding 20MPI_Graph_map(comm, nnodes, index, edges, newrank, ierror) 21TYPE(MPI_Comm), INTENT(IN) :: comm 22 INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*) 23INTEGER, INTENT(OUT) :: newrank 24INTEGER, OPTIONAL, INTENT(OUT) :: ierror 25Fortran binding 26MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR) 27INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR 2829 Advice to implementors. The function MPI_GRAPH_CREATE(comm, nnodes, index, 30 31edges, reorder, comm_graph), with reorder = true can be implemented by calling 32 MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank), then calling 33 MPI_COMM_SPLIT(comm, color, key, comm_graph), with color = 0 if newrank \neq 34 MPI_UNDEFINED, color = MPI_UNDEFINED otherwise, and key = newrank. 35All other graph topology functions can be implemented locally, using the topology 36 information that is eached with the communicator. (End of advice to implementors.)

7.6 Neighborhood Collective Communication on Process Topologies

MPI process topologies specify a communication graph, but they implement no communication function themselves. Many applications require sparse nearest neighbor communications that can be expressed as graph topologies. We now describe several collective operations that perform communication along the edges of a process topology. All of these functions are collective; i.e., they must be called by all processes in the specified communicator. See Section 5 for an overview of other dense (global) collective communication operations and the semantics of collective operations.

4748

37 38

39 40

41

42

43

44

45

46

1

2

5 6

¹ If the graph was created with MPI_DIST_GRAPH_CREATE_ADJACENT with sources ² and destinations containing 0, ..., n-1, where n is the number of processes in the group ³ of comm_old (i.e., the graph is fully connected and also includes an edge from each node ⁴ to itself), then the sparse neighborhood communication routine performs the same data ⁵ exchange as the corresponding dense (fully-connected) collective operation. In the case of a ⁶ Cartesian communicator, only nearest neighbor communication is provided, corresponding ⁷ to rank_source and rank_dest in MPI_CART_SHIFT with input disp = 1.

8 9

10

11

12

13

14 15 Rationale. Neighborhood collective communications enable communication on a process topology. This high-level specification of data exchange among neighboring processes enables optimizations in the MPI library because the communication pattern is known statically (the topology). Thus, the implementation can compute optimized message schedules during creation of the topology [36]. This functionality can significantly simplify the implementation of neighbor exchanges [32]. (End of rationale.)

For a distributed graph topology, created with MPI_DIST_GRAPH_CREATE, the se-16quence of neighbors in the send and receive buffers at each process is defined as the sequence 17returned by MPI_DIST_GRAPH_NEIGHBORS for destinations and sources, respectively. For 18 a general graph topology, created with MPI_GRAPH_CREATE, the use of neighborhood col-19lective communication is restricted to adjacency matrices, where the number of edges be-20tween any two processes is defined to be the same for both processes (i.e., with a symmetric 21adjacency matrix). In this case, the order of neighbors in the send and receive buffers is 22 defined as the sequence of neighbors as returned by MPI_GRAPH_NEIGHBORS. Note that 23general graph topologies should generally be replaced by the distributed graph topologies. 24 For a Cartesian topology, created with MPI CART_CREATE, the sequence of neigh-25bors in the send and receive buffers at each process is defined by order of the dimensions, 26first the neighbor in the negative direction and then in the positive direction with dis-27placement 1. The numbers of sources and destinations in the communication routines are 282^{*}ndims with ndims defined in MPI_CART_CREATE. If a neighbor does not exist, i.e., at 29 the border of a Cartesian topology in the case of a non-periodic virtual grid dimension (i.e., 30 periods[...]==false), then this neighbor is defined to be MPI_PROC_NULL. 31

If a neighbor in any of the functions is MPI_PROC_NULL, then the neighborhood collective communication behaves like a point-to-point communication with MPI_PROC_NULL in this direction. That is, the buffer is still part of the sequence of neighbors but it is neither communicated nor updated.

36 37

7.6.1 Neighborhood Gather

In this function, each process i gathers data items from each process j if an edge (j, i) exists in the topology graph, and each process i sends the same data items to all processes j where an edge (i, j) exists. The send buffer is sent to each neighboring process and the l-th block in the receive buffer is received from the l-th neighbor.

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

MPI_NEIGHBOR_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 1 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 IN sendtype data type of send buffer elements (handle) 7 OUT recvbuf starting address of receive buffer (choice) 9 number of elements received from each neighbor IN recvcount 10 (non-negative integer) 11 IN recvtype data type of receive buffer elements (handle) 1213 IN comm communicator with topology structure (handle) 1415C binding 16int MPI_Neighbor_allgather(const void *sendbuf, int sendcount, 17 MPI_Datatype sendtype, void *recvbuf, int recvcount, 18 MPI_Datatype recvtype, MPI_Comm comm) 19 Fortran 2008 binding 2021MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, 22 recvtype, comm, ierror) 23TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf 24 INTEGER, INTENT(IN) :: sendcount, recvcount 25TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 26TYPE(*), DIMENSION(..) :: recvbuf 27TYPE(MPI_Comm), INTENT(IN) :: comm 28 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 29 Fortran binding 30 MPI_NEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 31RECVTYPE, COMM, IERROR) 32 <type> SENDBUF(*), RECVBUF(*) 33 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR 34 35This function supports Cartesian communicators, graph communicators, and distributed 36 graph communicators as described in Section 7.6. If comm is a distributed graph commu-37 nicator, the outcome is as if each process executed sends to each of its outgoing neighbors 38 and receives from each of its incoming neighbors: 39 40 MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted); 41 int *srcs=(int*)malloc(indegree*sizeof(int)); 42int *dsts=(int*)malloc(outdegree*sizeof(int)); MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED, 43 44 outdegree, dsts, MPI_UNWEIGHTED); 45int k,l; 4647/* assume sendbuf and recvbuf are of type (char*) */

for(k=0; k<outdegree; ++k)</pre>

1 2	<pre>MPI_Isend(sendbuf, sendcount, sendtype,dsts[k],);</pre>
2 3 4 5 6	<pre>for(1=0; l<indegree; ++1)="" mpi_irecv(recvbuf+l*recvcount*extent(recvtype),="" recvcount,="" recvtype,<="" td=""></indegree;></pre>
7 8	<pre>MPI_Waitall();</pre>
9 10 11 12 13	Figure 7.1 shows the neighborhood gather communication of one process with outgoing neighbors $d_0 \ldots d_3$ and incoming neighbors $s_0 \ldots s_5$. The process will send its sendbuf to all four destinations (outgoing neighbors) and it will receive the contribution from all six sources (incoming neighbors) into separate locations of its receive buffer.
14 15 16 17	s_0 d_2, s_4
18 19 20 21	d_1 s_1 s_3 s_3
22 23 24 25	sendbuf d_3, s_5
26 27 28 29	$\begin{array}{c ccccccccccccccccccccccccccccccccccc$
30 31	Figure 7.1: Neighborhood gather communication example.
32 33 34 35 36 37 38 39 40 41 42 43 44	All arguments are significant on all processes and the argument comm must have iden- tical values on all processes. The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at all other processes. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed.
46 47 48	

MPI_NE		(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,	1	
	recvtype, comm)		2 3	
IN	sendbuf	starting address of send buffer (choice)	4	
IN	sendcount	number of elements sent to each neighbor (non-negative integer)	5 6	
IN	sendtype	data type of send buffer elements (handle)	7	
OUT	recvbuf	starting address of receive buffer (choice)	8	
IN	recvcounts	non-negative integer array (of length indegree) containing the number of elements that are received from each neighbor	9 10 11 12	
IN	displs	integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i	13 14 15 16	
IN	recvtype	data type of receive buffer elements (handle)	10	
IN	comm	communicator with topology structure (handle)	18	
Fortrar MPI_Nei TYF INT TYF TYF	MPI_Datatype s const int disp a 2008 binding .ghbor_allgatherv(sen displs, recvty PE(*), DIMENSION(), TEGER, INTENT(IN) ::	IN) :: comm	22 23 24 25 26 27 28 29 30 31 32	
			33 34	
<pre>Fortran binding MPI_NEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,</pre>				
graph contractor, and reconstruction MPI_Dis	ommunicators as describ the outcome is as if each eives from each of its incost_graph_neighbors_co	unt(comm, &indegree, &outdegree, &weighted);	40 41 42 43 44 45 46	
	<pre>nt *srcs=(int*)malloc(indegree*sizeof(int)); 47</pre>			

```
int *dsts=(int*)malloc(outdegree*sizeof(int));
```

```
1
     MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
\mathbf{2}
                                    outdegree, dsts, MPI_UNWEIGHTED);
3
      int k,l;
4
5
      /* assume sendbuf and recvbuf are of type (char*) */
6
      for(k=0; k<outdegree; ++k)</pre>
\overline{7}
        MPI_Isend(sendbuf, sendcount, sendtype, dsts[k],...);
8
9
      for(l=0; l<indegree; ++l)</pre>
10
        MPI_Irecv(recvbuf+displs[1]*extent(recvtype), recvcounts[1], recvtype,
11
                    srcs[1],...);
12
13
     MPI_Waitall(...);
14
          The type signature associated with sendcount, sendtype, at process j must be equal
15
      to the type signature associated with recvcounts[l], recvtype at any other process with
16
      srcs[l] = j. This implies that the amount of data sent must be equal to the amount of
17
      data received, pairwise between every pair of communicating processes. Distinct type maps
18
      between sender and receiver are still allowed. The data received from the l-th neighbor is
19
      placed into recvbuf beginning at offset displs[l] elements (in terms of the recvtype).
20
          The "in place" option is not meaningful for this operation.
21
          All arguments are significant on all processes and the argument comm must have iden-
22
      tical values on all processes.
23
^{24}
      7.6.2
             Neighbor Alltoall
25
26
      In this function, each process i receives data items from each process j if an edge (j,i)
27
      exists in the topology graph or Cartesian topology. Similarly, each process i sends data
28
      items to all processes j where an edge (i, j) exists. This call is more general than
29
      MPI_NEIGHBOR_ALLGATHER in that different data items can be sent to each neighbor.
30
      The k-th block in send buffer is sent to the k-th neighboring process and the l-th block in
^{31}
      the receive buffer is received from the l-th neighbor.
32
33
      MPI_NEIGHBOR_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
34
                      comm)
35
36
        IN
                  sendbuf
                                                starting address of send buffer (choice)
37
        IN
                  sendcount
                                                number of elements sent to each neighbor
38
                                                (non-negative integer)
39
        IN
                  sendtype
                                                data type of send buffer elements (handle)
40
41
        OUT
                  recvbuf
                                                starting address of receive buffer (choice)
42
        IN
                                                number of elements received from each neighbor
                  recvcount
43
                                                (non-negative integer)
44
        IN
                                                data type of receive buffer elements (handle)
                  recvtype
45
        IN
                                                communicator with topology structure (handle)
46
                  comm
47
```

```
^{48} C binding
```

```
1
int MPI_Neighbor_alltoall(const void *sendbuf, int sendcount,
                                                                                       \mathbf{2}
              MPI_Datatype sendtype, void *recvbuf, int recvcount,
                                                                                       3
              MPI_Datatype recvtype, MPI_Comm comm)
Fortran 2008 binding
MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                       6
              recvtype, comm, ierror)
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
    INTEGER, 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
                                                                                       14
Fortran binding
                                                                                       15
MPI_NEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
                                                                                       16
              RECVTYPE, COMM, IERROR)
                                                                                       17
    <type> SENDBUF(*), RECVBUF(*)
                                                                                       18
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
                                                                                       19
    This function supports Cartesian communicators, graph communicators, and distributed
                                                                                       20
graph communicators as described in Section 7.6. If comm is a distributed graph commu-
                                                                                       21
nicator, the outcome is as if each process executed sends to each of its outgoing neighbors
                                                                                       22
and receives from each of its incoming neighbors:
                                                                                       23
                                                                                       ^{24}
MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
                                                                                       25
int *srcs=(int*)malloc(indegree*sizeof(int));
                                                                                       26
int *dsts=(int*)malloc(outdegree*sizeof(int));
                                                                                       27
MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
                                                                                       28
                           outdegree, dsts, MPI_UNWEIGHTED);
                                                                                       29
int k,l;
                                                                                       30
                                                                                       31
/* assume sendbuf and recvbuf are of type (char*) */
                                                                                       32
for(k=0; k<outdegree; ++k)</pre>
                                                                                       33
  MPI_Isend(sendbuf+k*sendcount*extent(sendtype), sendcount, sendtype,
                                                                                       34
             dsts[k],...);
                                                                                       35
                                                                                       36
for(l=0; l<indegree; ++1)</pre>
                                                                                       37
  MPI_Irecv(recvbuf+l*recvcount*extent(recvtype), recvcount, recvtype,
                                                                                       38
             srcs[1],...);
                                                                                       39
                                                                                       40
MPI_Waitall(...);
                                                                                       41
                                                                                       42
    The type signature associated with sendcount, sendtype, at a process must be equal to
                                                                                       43
```

the type signature associated with schucount, schucype, 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.

Unofficial Draft for Comment Only

44

45

46

47

```
1
          All arguments are significant on all processes and the argument comm must have iden-
\mathbf{2}
     tical values on all processes.
3
          The vector variant of MPI_NEIGHBOR_ALLTOALL allows sending/receiving different
4
     numbers of elements to and from each neighbor.
5
6
     MPI_NEIGHBOR_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
7
                     rdispls, recvtype, comm)
8
9
       IN
                 sendbuf
                                               starting address of send buffer (choice)
10
       IN
                 sendcounts
                                               non-negative integer array (of length outdegree)
11
                                               specifying the number of elements to send to each
12
                                               neighbor
13
       IN
                 sdispls
                                               integer array (of length outdegree). Entry j specifies
14
                                               the displacement (relative to sendbuf) from which to
15
                                               send the outgoing data to neighbor j
16
17
       IN
                 sendtype
                                               data type of send buffer elements (handle)
18
       OUT
                 recvbuf
                                               starting address of receive buffer (choice)
19
                  recvcounts
       IN
                                               non-negative integer array (of length indegree)
20
                                               specifying the number of elements that are received
21
                                               from each neighbor
22
23
       IN
                                               integer array (of length indegree). Entry i specifies
                  rdispls
^{24}
                                               the displacement (relative to recvbuf) at which to
25
                                               place the incoming data from neighbor i
26
       IN
                                               data type of receive buffer elements (handle)
                  recvtype
27
                                               communicator with topology structure (handle)
       IN
                 comm
28
29
     C binding
30
     int MPI_Neighbor_alltoallv(const void *sendbuf, const int sendcounts[],
^{31}
32
                     const int sdispls[], MPI_Datatype sendtype, void *recvbuf,
                     const int recvcounts[], const int rdispls[],
33
34
                     MPI_Datatype recvtype, MPI_Comm comm)
35
     Fortran 2008 binding
36
     MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
37
                     recvcounts, rdispls, recvtype, comm, ierror)
38
          TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
39
          INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
40
                      rdispls(*)
41
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
42
          TYPE(*), DIMENSION(...) :: recvbuf
43
          TYPE(MPI_Comm), INTENT(IN) :: comm
44
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
     Fortran binding
47
     MPI_NEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
48
                     RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR)
```

```
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
RECVTYPE, COMM, IERROR
```

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6. If comm is a distributed graph communicator, the outcome is as if each process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

```
MPI_Waitall(...);
```

The type signature associated with sendcounts[k], sendtype with dsts[k]==j at process i must be equal to the type signature associated with recvcounts[l], recvtype with srcs[l]==iat process j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed. The data in the sendbuf beginning at offset sdispls[k] elements (in terms of the sendtype) is sent to the k-th outgoing neighbor. The data received from the l-th incoming neighbor is placed into recvbuf beginning at offset rdispls[l]elements (in terms of the recvtype).

The "in place" option is not meaningful for this operation.

All arguments are significant on all processes and the argument **comm** must have identical values on all processes.

MPI_NEIGHBOR_ALLTOALLW allows one to send and receive with different datatypes to and from each neighbor.

 31

12	MPI_NEIGHBOR_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm)			
3 4	IN	sendbuf	starting address of send buffer (choice)	
5 6 7	IN	sendcounts	non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor	
8 9 10 11	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)	
12 13 14 15	IN	sendtypes	array of datatypes (of length outdegree). Entry j specifies the type of data to send to neighbor j (array of handles)	
16	OUT	recvbuf	starting address of receive buffer (choice)	
17 18 19	IN	recvcounts	non-negative integer array (of length indegree) specifying the number of elements that are received from each neighbor	
20 21 22 23 24	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)	
25 26 27	IN	recvtypes	array of datatypes (of length indegree). Entry i specifies the type of data received from neighbor i (array of handles)	
28 29	IN	comm	communicator with topology structure (handle)	
30 31	C bindir	3		
32 33	<pre>int MPI_Neighbor_alltoallw(const void *sendbuf, const int sendcounts[],</pre>			
34 35 36	const MPI_Aint rdispls[], const MPI_Datatype recvtypes[], MPI_Comm comm)			
37	Fortran	2008 binding		
38 39	<pre>MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm, ierror)</pre>			
40	TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf			
41	<pre>INTEGER, INTENT(IN) :: sendcounts(*), recvcounts(*) INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)</pre>			
42 43			N) :: sendtypes(*), recvtypes(*)	
44	TYPE	(*), DIMENSION() :: red	cvbuf	
45		(MPI_Comm), INTENT(IN) :		
46 47	1 N'I'E	GER, OPTIONAL, INTENT(OUT	1) :: lerror	
47 48	Fortran	binding		

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6. If comm is a distributed graph communicator, the outcome is as if each process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

```
MPI_Irecv(recvbuf+rdispls[1], recvcounts[1], recvtypes[1], srcs[1],...);
```

MPI_Waitall(...);

The type signature associated with sendcounts[k], sendtypes[k] with dsts[k]==j at process i must be equal to the type signature associated with recvcounts[l], recvtypes[l] with srcs[l]==i at process j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed.

The "in place" option is not meaningful for this operation.

All arguments are significant on all processes and the argument **comm** must have identical values on all processes.

7.7 Nonblocking Neighborhood Communication on Process Topologies

Nonblocking variants of the neighborhood collective operations allow relaxed synchronization and overlapping of computation and communication. The semantics are similar to nonblocking collective operations as described in Section 5.12. $\mathbf{2}$

	366		CHAPTER 7. PROCESS TOPOLOGIES
1 2 3	7.7.1 Nonblocking Neighborhood Gather		
4 5	MPI_INEIG	HBOR_ALLGATHER(sendbuf, comm, request)	sendcount, sendtype, recvbuf, recvcount, recvtype,
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	data type 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	data type of receive buffer elements (handle)
16 17	IN	comm	communicator with topology structure (handle)
18	OUT	request	communication request (handle)
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	C binding int MPI_Ineighbor_allgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) Fortran 2008 binding MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf INTEGER, INTENT(IN) :: sendcount, recvcount TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Comm), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_INEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR) (type> SENDBUF(*), RECVBUF(*) INTEGER SENDFOUNT, SENDTYPE, COMM, REQUEST, IERROR This call starts a nonblocking variant of MPI_NEIGHBOR_ALLGATHER.		

MPI_INEIG	HBOR_ALLGATHERV(sendb recvtype, comm, request	uf, sendcount, sendtype, recvbuf, recvcounts, displs,)	1 2
IN	sendbuf	, starting address of send buffer (choice)	3
IN	sendcount	number of elements sent to each neighbor (non-negative integer)	4 5 6
IN	sendtype	data type of send buffer elements (handle)	7
OUT	recvbuf	starting address of receive buffer (choice)	8
IN	recvcounts	non-negative integer array (of length indegree) containing the number of elements that are received from each neighbor	9 10 11 12
IN	displs	integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i	13 14 15
IN	recvtype	data type of receive buffer elements (handle)	16 17
IN	comm	communicator with topology structure (handle)	18
OUT	request	communication request (handle)	19
MPI_Ineig TYPE(INTEG TYPE(TYPE(INTEG TYPE(INTEG INTEG Fortran b	<pre>const int displs[], MPI_Request *request 008 binding hbor_allgatherv(sendbuf, displs, recvtype, co *), DIMENSION(), INTEN ER, INTENT(IN) :: sendco MPI_Datatype), INTENT(IN *), DIMENSION(), ASYNCH ER, INTENT(IN), ASYNCHRO MPI_Comm), INTENT(IN) :: MPI_Comm), INTENT(IN) :: MPI_Request), INTENT(OUT ER, OPTIONAL, INTENT(OUT inding HBOR_ALLGATHERV(SENDBUF,</pre>	<pre>sendcount, sendtype, recvbuf, recvcounts, omm, request, ierror) T(IN), ASYNCHRONOUS :: sendbuf unt) :: sendtype, recvtype HRONOUS :: recvbuf NOUS :: recvcounts(*), displs(*) comm) :: request) :: ierror SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,</pre>	23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
<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 This call starts a nonblocking variant of MPI_NEIGHBOR_ALLGATHERV.</type></pre>			40 41 42 43 44 45 46 47

	368		CHAPTER 7.	PROCESS TOPOLOGIES
1 2 3	7.7.2 Nonblocking Neighborhood Alltoall			
4 5	MPI_INEIGHBOR_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)			cvbuf, recvcount, recvtype,
6 7	IN	sendbuf	starting address of sen	d buffer (choice)
8 9	IN	sendcount	number of elements set (non-negative integer)	nt to each neighbor
10	IN	sendtype	data type of send buffe	er elements (handle)
11 12	OUT	recvbuf	starting address of rec	eive buffer (choice)
13 14	IN	recvcount	number of elements red (non-negative integer)	ceived from each neighbor
15	IN	recvtype	data type of receive bu	uffer elements (handle)
16 17	IN	comm	communicator with to	pology structure (handle)
18	OUT	request	communication reques	t (handle)
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	<pre>Int WF1_InFighton_alltoall(const void *sendoul, int sendoult, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) Fortran 2008 binding MPI_Ineighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request, iarror) 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_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_INEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDEUF(*), RECVBUF(*) INTEGER SENDEUF(*), RECVBUF(*) INTEGER a nonblocking variant of MPI_NEIGHBOR_ALLTOALL.</type></pre>			

MPI_INEIGHBOR_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, ¹ rdispls, recvtype, comm, request) 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 send the outgoing data to neighbor j	8 9 10	
IN	sendtype	data type of send buffer elements (handle)	11 12	
OUT	recvbuf	starting address of receive buffer (choice)	13	
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 recvbuf) at which to place the incoming data from neighbor i	17 18 19 20	
IN	recvtype	data type of receive buffer elements (handle)	21	
IN	comm	communicator with topology structure (handle)	22	
OUT	request	communication request (handle)	23 24	
C binding int MPI_Ineighbor_alltoallv(const void *sendbuf, const int sendcounts[], const int sdispls[], MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)				
Fortran 2	2008 binding		31	
<pre>MPI_Ineighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*), recvcounts(*), rdispls(*) TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>				
<pre>Fortran binding MPI_INEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,</pre>			42 43 44 45 46 47 48	

1	This ca	all starts a nonblocking variar	at of MPI_NEIGHBOR_ALLTOALLV.
2			
3 4 5	MPI_INEIG	HBOR_ALLTOALLW(sendbuf, rdispls, recvtypes, comm,	sendcounts, sdispls, sendtypes, recvbuf, recvcounts, request)
6	IN	sendbuf	starting address of send buffer (choice)
7 8 9 10	IN	sendcounts	non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor
11 12 13 14	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)
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	OUT	recvbuf	starting address of receive buffer (choice)
20 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 26	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)
27 28 29 30	IN	recvtypes	array of datatypes (of length indegree). Entry i specifies the type of data received from neighbor i (array of handles)
31	IN	comm	communicator with topology structure (handle)
32 33 34	OUT	request	communication request (handle)
35	C binding		<pre>void *sendbuf, const int sendcounts[],</pre>
36 37	1110 111 1_11		<pre>Ls[], const MPI_Datatype sendtypes[],</pre>
38 39 40		const MPI_Aint rdisp] MPI_Comm comm, MPI_Re	ls[], const MPI_Datatype recvtypes[], equest *request)
41	Fortran 2	008 binding	
42	MPI_Ineight		endcounts, sdispls, sendtypes, recvbuf,
43		-	recvtypes, comm, request, ierror)
44			'(IN), ASYNCHRONOUS :: sendbuf OUS :: sendcounts(*), recvcounts(*)
45 46			, INTENT(IN), ASYNCHRONOUS :: sdispls(*),
40	1.1100	rdispls(*)	,, _,, _
48		L	

TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*), recvtypes(*) TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_INEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM, 12REQUEST, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*) 14This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALLW. Persistent Neighborhood Communication on Process Topologies 7.8 19Persistent variants of the neighborhood collective operations can offer significant perfor-20mance benefits for programs with repetitive communication patterns. The semantics are 21similar to persistent collective operations as described in Section 5.13. 22

7.8.1 Persistent Neighborhood Gather

MPI_NEIGHBOR_ALLGATHER_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, info, request)

IN	sendbuf	starting address of send buffer (choice)	29
IN	sendcount	number of elements sent to each neighbor (non-negative integer)	30 31 32
IN	sendtype	data type of send buffer elements (handle)	33
OUT	recvbuf	starting address of receive buffer (choice)	34
IN	recvcount	number of elements received from each neighbor (non-negative integer)	35 36 37
IN	recvtype	data type of receive buffer elements (handle)	38
IN	comm	communicator with topology structure (handle)	39
IN	info	info argument (handle)	40 41
OUT	request	communication request (handle)	42
			43

C binding

int	<pre>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)

Unofficial Draft for Comment Only

1

2

3

4

5

6

9

10

11

13

151617

18

23

 24 2526

27

28

44

```
1
     Fortran 2008 binding
\mathbf{2}
     MPI_Neighbor_allgather_init(sendbuf, sendcount, sendtype, recvbuf,
3
                     recvcount, recvtype, comm, info, request, ierror)
4
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
5
          INTEGER, INTENT(IN) :: sendcount, recvcount
6
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
7
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
8
          TYPE(MPI_Comm), INTENT(IN) :: comm
9
          TYPE(MPI_Info), INTENT(IN) :: info
10
          TYPE(MPI_Request), INTENT(OUT) :: request
11
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
     Fortran binding
13
     MPI_NEIGHBOR_ALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
14
                     RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR)
15
          <type> SENDBUF(*), RECVBUF(*)
16
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
17
                      IERROR
18
19
          Creates a persistent collective communication request for the neighborhood allgather
20
     operation.
21
22
     MPI_NEIGHBOR_ALLGATHERV_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
23
                     displs, recvtype, comm, info, request)
24
25
       IN
                 sendbuf
                                              starting address of send buffer (choice)
26
       IN
                 sendcount
                                              number of elements sent to each neighbor
27
                                              (non-negative integer)
28
       IN
                 sendtype
                                              data type of send buffer elements (handle)
29
30
       OUT
                 recvbuf
                                              starting address of receive buffer (choice)
31
       IN
                                              non-negative integer array (of length indegree)
                 recycounts
32
                                              containing the number of elements that are received
33
                                              from each neighbor
34
       IN
                 displs
                                              integer array (of length indegree). Entry i specifies
35
                                              the displacement (relative to recvbuf) at which to
36
                                              place the incoming data from neighbor i
37
38
       IN
                 recvtype
                                              data type of receive buffer elements (handle)
39
       IN
                 comm
                                              communicator with topology structure (handle)
40
       IN
                 info
                                              info argument (handle)
41
       OUT
                 request
                                              communication request (handle)
42
43
44
     C binding
45
     int MPI_Neighbor_allgatherv_init(const void *sendbuf, int sendcount,
46
                     MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
47
                     const int displs[], MPI_Datatype recvtype, MPI_Comm comm,
48
                     MPI_Info info, MPI_Request *request)
```

Fortran 2008 binding				
MPI_Neighbor_allgatherv_init(sendbuf, sendcount, sendtype, recvbuf,				
	-	recvtype, comm, info, request, ierror)	3	
		(IN), ASYNCHRONOUS :: sendbuf	4 5	
	ER, INTENT(IN) :: sendcou	-	6	
	MPI_Datatype), INTENT(IN)	VI VI	7	
	*), DIMENSION(), ASYNCH		8	
	ER, INTENT(IN), ASYNCHRON		9	
	<pre>MPI_Comm), INTENT(IN) :: MPI_Info), INTENT(IN) ::</pre>		10	
	MPI_INIO), INTENI(IN) :: MPI_Request), INTENT(OUT)		11	
	ER, OPTIONAL, INTENT(OUT)	-	12	
	ER, OFFICIAL, INTENT(001)	161101	13	
Fortran b	inding		14	
MPI_NEIGH		UF, SENDCOUNT, SENDTYPE, RECVBUF,	15	
		RECVTYPE, COMM, INFO, REQUEST, IERROR)	16	
	> SENDBUF(*), RECVBUF(*)		17	
INTEG		ECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,	18	
INFO, REQUEST, IERROR			19	
Creates a persistent collective communication request for the neighborhood allgatherv			20	
operation.			21	
			22	
7.8.2 Per	sistent Neighborhood Alltoall		23 24	
			25	
MPI_NEIGI	HBOR_ALLTOALL_INIT(sendb	uf, sendcount, sendtype, recvbuf, recvcount,	26 27	
	recvtype, comm, info, req	uest)	27	
IN	sendbuf	starting address of send buffer (choice)	29	
IN	sendcount	number of elements sent to each neighbor	30	
		(non-negative integer)	31	
IN	sendtype	data type of send buffer elements (handle)	32	
			33	
OUT	recvbuf	starting address of receive buffer (choice)	34	
IN	recvcount	number of elements received from each neighbor	35	
		(non-negative integer)	36	
IN	recvtype	data type of receive buffer elements (handle)	37 38	
IN	comm	communicator with topology structure (handle)	39	
IN	info	info argument (handle)	40	
	-			

C binding int MPI_Neighbor_alltoall_init(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, MPI_Request *request) 43 44 45 46 47 48

communication request (handle)

OUT

request

```
1
      Fortran 2008 binding
\mathbf{2}
      MPI_Neighbor_alltoall_init(sendbuf, sendcount, sendtype, recvbuf,
3
                      recvcount, recvtype, comm, info, request, ierror)
4
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
5
          INTEGER, INTENT(IN) :: sendcount, recvcount
6
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
7
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
8
          TYPE(MPI_Comm), INTENT(IN) :: comm
9
          TYPE(MPI_Info), INTENT(IN) :: info
10
          TYPE(MPI_Request), INTENT(OUT) :: request
11
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
      Fortran binding
13
      MPI_NEIGHBOR_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
14
                      RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR)
15
          <type> SENDBUF(*), RECVBUF(*)
16
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
17
                      IERROR
18
19
          Creates a persistent collective communication request for the neighborhood alltoall
20
      operation.
21
22
      MPI_NEIGHBOR_ALLTOALLV_INIT(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
23
                      recvcounts, rdispls, recvtype, comm, info, request)
^{24}
25
        IN
                  sendbuf
                                                starting address of send buffer (choice)
26
        IN
                  sendcounts
                                                non-negative integer array (of length outdegree)
27
                                                specifying the number of elements to send to each
28
                                                neighbor
29
        IN
                  sdispls
                                                integer array (of length outdegree). Entry j specifies
30
                                                the displacement (relative to sendbuf) from which
^{31}
                                                send the outgoing data to neighbor j
32
33
        IN
                  sendtype
                                                data type of send buffer elements (handle)
34
        OUT
                  recvbuf
                                                starting address of receive buffer (choice)
35
        IN
                                                non-negative integer array (of length indegree)
                  recvcounts
36
                                                specifying the number of elements that are received
37
                                                from each neighbor
38
                                                integer array (of length indegree). Entry i specifies
        IN
                  rdispls
39
                                                the displacement (relative to recvbuf) at which to
40
                                                place the incoming data from neighbor i
41
42
        IN
                  recvtype
                                                data type of receive buffer elements (handle)
43
        IN
                  comm
                                                communicator with topology structure (handle)
44
        IN
                  info
                                                info argument (handle)
45
        OUT
                                                communication request (handle)
                  request
46
47
48
```

```
<sup>8</sup> C binding
```

```
1
int MPI_Neighbor_alltoallv_init(const void *sendbuf,
                                                                                      2
              const int sendcounts[], const int sdispls[],
                                                                                      3
              MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
              const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm,
                                                                                      \mathbf{4}
              MPI_Info info, MPI_Request *request)
                                                                                      5
                                                                                      6
Fortran 2008 binding
MPI_Neighbor_alltoallv_init(sendbuf, sendcounts, sdispls, sendtype,
              recvbuf, recvcounts, rdispls, recvtype, comm, info, request,
                                                                                      a
              ierror)
                                                                                      10
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                      11
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
                                                                                      12
               recvcounts(*), rdispls(*)
                                                                                      13
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                      14
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                      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
Fortran binding
                                                                                     21
MPI_NEIGHBOR_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE,
                                                                                      22
              RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST,
                                                                                     23
              IERROR)
                                                                                      24
    <type> SENDBUF(*), RECVBUF(*)
                                                                                      25
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
                                                                                      26
               RECVTYPE, COMM, INFO, REQUEST, IERROR
                                                                                      27
    Creates a persistent collective communication request for the neighborhood alltoally
                                                                                      28
operation.
                                                                                      29
                                                                                      30
                                                                                      31
                                                                                      32
                                                                                      33
                                                                                      34
                                                                                      35
                                                                                      36
                                                                                      37
                                                                                      38
                                                                                      39
                                                                                      40
                                                                                      41
                                                                                      42
                                                                                      43
                                                                                      44
                                                                                      45
                                                                                      46
                                                                                      47
```

1 2	MPI_NEIGHBOR_ALLTOALLW_INIT(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm, info, request)		
3 4	IN	sendbuf	starting address of send buffer (choice)
5 6 7	IN	sendcounts	non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor
8 9 10 11	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)
12 13 14 15	IN	sendtypes	array of datatypes (of length outdegree). Entry j specifies the type of data to send to neighbor j (array of handles)
16	OUT	recvbuf	starting address of receive buffer (choice)
17 18 19	IN	recvcounts	non-negative integer array (of length indegree) specifying the number of elements that are received from each neighbor
20 21 22 23 24	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)
25 26 27	IN	recvtypes	array of datatypes (of length indegree). Entry i specifies the type of data received from neighbor i (array of handles)
28	IN	comm	communicator with topology structure (handle)
29 30	IN	info	info argument (handle)
31	OUT	request	communication request (handle)
32 33 34 35 36 37 38 39	C bindin int MPI_N	Veighbor_alltoallw const int sen const MPI_Dat const int red	<pre>y_init(const void *sendbuf, ndcounts[], const MPI_Aint sdispls[], tatype sendtypes[], void *recvbuf, cvcounts[], const MPI_Aint rdispls[], tatype recvtypes[], MPI_Comm comm, MPI_Info info, *request)</pre>
40 41 42 43 44 45 46 47 48	MPI_Neigh TYPE INTEC	recvbuf, recv ierror) (*), DIMENSION() GER, INTENT(IN), A	t(sendbuf, sendcounts, sdispls, sendtypes, acounts, rdispls, recvtypes, comm, info, request, , INTENT(IN), ASYNCHRONOUS :: sendbuf SYNCHRONOUS :: sendcounts(*), recvcounts(*) SS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),

```
TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
        recvtypes(*)
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Comm), INTENT(IN) :: info
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
MPI_NEIGHBOR_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES,
        RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST,
        IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNTS(*), SENDTYPES(*), COMM,
        INFO, REQUEST, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
```

Creates a persistent collective communication request for the neighborhood all toallw operation.

7.9 An Application Example

Example 7.9 The example in Figures 7.2-7.5 shows how the grid definition and inquiry functions can be used in an application program. A partial differential equation, for instance the Poisson equation, is to be solved on a rectangular domain. First, the processes organize themselves in a two-dimensional structure. Each process then inquires about the ranks of its neighbors in the four directions (up, down, right, left). The numerical problem is solved by an iterative method, the details of which are hidden in the subroutine relax.

In each relaxation step each process computes new values for the solution grid function at the points u(1:100,1:100) owned by the process. Then the values at inter-process boundaries have to be exchanged with neighboring processes. For example, the newly calculated values in u(1,1:100) must be sent into the halo cells u(101,1:100) of the left-hand neighbor with coordinates (own_coord(1)-1,own_coord(2)).

1

2

4

5

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

```
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 7.2: 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)
REAL u(0:101,0:101)
INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
REAL sndbuf(100,num_neigh), rcvbuf(100,num_neigh)
INTEGER ierr
sndbuf(1:100,1) = u( 1,1:100)
sndbuf(1:100,2) = u(100,1:100)
sndbuf(1:100,3) = u(1:100, 1)
sndbuf(1:100,4) = u(1:100,100)
CALL MPI_NEIGHBOR_ALLTOALL(sndbuf, 100, MPI_REAL, rcvbuf, 100, MPI_REAL, &
                           comm_cart, ierr)
! instead of
! DO i=1,num_neigh
    CALL MPI_IRECV(rcvbuf(1,i), 100, MPI_REAL, neigh_rank(i),..., &
i
1
                   rq(2*i-1), ierr)
    CALL MPI_ISEND(sndbuf(1,i), 100, MPI_REAL, neigh_rank(i),...
!
                   rq(2*i ), ierr)
L
! END DO
! CALL MPI_WAITALL(2*num_neigh, rq, statuses, ierr)
u(0,1:100) = rcvbuf(1:100,1)
u(101,1:100) = rcvbuf(1:100,2)
u(1:100, 0) = rcvbuf(1:100,3)
u(1:100,101) = rcvbuf(1:100,4)
END
Figure 7.3: Communication routine with local data copying and sparse neighborhood all-
to-all.
```

2

3

4

5

6

10

11

12

13

14

15

16

17

18

19 20

21

22

23

 24

25 26 27

28

```
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 7.4: 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
!
                                                                                   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
!
                                                                                    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 7.5: Two-dimensional parallel Poisson solver with persistent sparse neighborhood all-to-all-w and without local data copying.

47



Chapter 8

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 8.1

8.1.1 Version Inquiries

In order to cope with changes to the MPI Standard, there are both compile-time and runtime ways to determine which version of the standard is in use in the environment one is using.

The "version" will be represented by two separate integers, for the version and subversion: In C,

```
#define MPI_VERSION
                        3
#define MPI_SUBVERSION 1
```

in Fortran.			
III FOILIAII,			
INTEGER :: MPI_VERSION, MP	I_SUBVERSION	33	
PARAMETER (MPI_VERSION	= 3)	34	
PARAMETER (MPI_SUBVERSION	= 1)	35	
		36	
For runtime determination,		37	
		38	
MPI_GET_VERSION(version, subvers	ion)	39	
Υ.	,	40	
OUT version	version number (integer)	41	
OUT subversion	subversion number (integer)	42	
		43	
C binding		44	
6	int MPI_Get_version(int *version, int *subversion)		
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 10.6. 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 10.6.
```

CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT

384

8.1.2 Environmental Inquiries	1
When using the World Model (Section 10.2), 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 6.7 and in Section 18.2.7. 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
MPI_HOST Host process rank, if such exists, MPI_PROC_NULL, otherwise.	11
MPI_IO rank of a node that has regular I/O facilities (possibly myrank). Nodes in the same	12 13
communicator may return different values for this parameter.	13
	15
MPI_WTIME_IS_GLOBAL Boolean variable that indicates whether clocks are synchronized.	16
When using the Sessions Model (Section 10.3), only the MPI_TAG_UB attribute is avail-	17
able.	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
Advise to seems. Note that in the C hinding, the value neturned by these attributes	23
Advice to users. Note that in the C binding, the value returned by these attributes is a <i>pointer</i> to an int containing the requested value. (End of advice to users.)	24
is a pointer to an int containing the requested value. (End of dubice to users.)	25
The required parameter values are discussed in more detail below:	26 27
	28
Tag Values	29
The values names from 0 to the value naturned for MDL TAC UD inclusive. These values are	30
Tag values range from 0 to the value returned for MPI_TAG_UB, inclusive. These values are guaranteed to be unchanging during the execution of an MPI program. In addition, the tag	31
upper bound value must be <i>at least</i> 32767. An MPI implementation is free to make the	32
value of MPI_TAG_UB larger than this; for example, the value $2^{30} - 1$ is also a valid value	33
for MPI_TAG_UB.	34
In the Sessions Model, the attribute MPI_TAG_UB is attached to all communicators cre-	35
ated by MPI_COMM_CREATE_FROM_GROUP and MPI_INTERCOMM_CREATE_FROM_GROUP	Ů₽́S.
with the same value on all MPI processes in the communicator. In the World Model, the	37
attribute MPI_TAG_UB has the same value on all processes of MPI_COMM_WORLD.	38
	39
Host Rank	40 41
The value returned for MPI_HOST gets the rank of the HOST process in the group associated	42
with communicator MPI_COMM_WORLD, if there is such. MPI_PROC_NULL is returned if	43
there is no host. MPI does not specify what it means for a process to be a <i>HOST</i> , nor does	44
it requires that a <i>HOST</i> exists.	45
The attribute MPI_HOST has the same value on all processes of MPI_COMM_WORLD.	46
• –	47
	48

386	CHAPTER 8.	MPI ENVIRO	ONMENTAL MAN	AGEMENT
IO Rank				
The value returned for MPI_IO I/O facilities. For Fortran, thi (e.g., OPEN, REWIND, WRITE). supported (e.g., fopen, fprin If every process can provi will be returned. Otherwise, then its rank will be returned I/O then the rank of one su returned by all processes. If n MPI_PROC_NULL will be return	is means that all For C, this means tf, lseek). ide language-sta , if the calling l. Otherwise, if , ch process will to process can pro-	l of the Fortran ans that all of ndard I/O, the process can p some process be returned.	n I/O operations and the ISO C I/O op en the value MPI_A rovide language-static can provide langua The same value	NY_SOURCE andard I/O, age-standard need not be
Advice to users. Note the which process can or do	-	<i>'</i>		<i>not</i> indicate
Clock Synchronization				
The value returned for MPI_W			-	
MDI COMM MODID are gung	bronized () oth	α	leation of clocks in	a considered

19MPI_COMM_WORLD are synchronized, 0 otherwise. A collection of clocks is considered 20synchronized if explicit effort has been taken to synchronize them. The expectation is that 21the variation in time, as measured by calls to MPI_WTIME, will be less then one half the 22round-trip time for an MPI message of length zero. If time is measured at a process just 23before a send and at another process just after a matching receive, the second time should 24 be always higher than the first one.

25The attribute MPI_WTIME_IS_GLOBAL need not be present when the clocks are not 26synchronized (however, the attribute key MPI_WTIME_IS_GLOBAL is always valid). This 27attribute may be associated with communicators other then MPI_COMM_WORLD.

```
28
         The attribute MPI_WTIME_IS_GLOBAL has the same value on all processes of
29
     MPI_COMM_WORLD.
30
```

Inquire Processor Name

1 2

3

4

56

7

8

9

10

11 1213

14

 31

32 33 34

35

MPI_GET_PROCESSOR_NAME(name, resultlen)

36 37	OUT	name	A unique specifier for the actual (as opposed to virtual) node.		
38 39	OUT	resultlen	Length (in printable characters) of the result returned in name		
40					
41	C binding				
42	<pre>int MPI_Get_processor_name(char *name, int *resultlen)</pre>				
43					
44	Fortran 2008 binding				
45	MPI_Get_processor_name(name, resultlen, ierror)				
46	CHARACTER(LEN=MPI_MAX_PROCESSOR_NAME), INTENT(OUT) :: name				
47	тмтг		·· rogultion		

```
INTEGER, INTENT(OUT) :: resultlen
47
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

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.)

8.2 Memory Allocation

In some systems, message-passing and remote-memory-access (RMA) operations run faster when accessing specially allocated memory (e.g., memory that is shared by the other processes in the communicating group on an SMP). MPI provides a mechanism for allocating and freeing such special memory. The use of such memory for message-passing or RMA is not mandatory, and this memory can be used without restrictions as any other dynamically allocated memory. However, implementations may restrict the use of some RMA functionality as defined in Section 11.5.3.

MPI_ALLOC_MEM(size, info, baseptr)

IN	size	size of memory segment in bytes (non-negative	40
		$\operatorname{integer})$	41
IN	info	info argument (handle)	42
OUT	baseptr	pointer to beginning of memory segment allocated	43
		Ferrer of a solution of memory colonical and and a	44

C binding int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)

Fortran 2008 binding

 $\mathbf{2}$

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

45

46

```
1
     MPI_Alloc_mem(size, info, baseptr, ierror)
\mathbf{2}
          USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
3
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
4
          TYPE(MPI_Info), INTENT(IN) :: info
5
          TYPE(C_PTR), INTENT(OUT) :: baseptr
6
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
     Fortran binding
8
     MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
9
          INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
10
          INTEGER INFO, IERROR
11
12
         If the Fortran compiler provides TYPE(C_PTR), then the following generic interface must
13
     be provided in the mpi module and should be provided in mpif.h through overloading,
14
     i.e., with the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND)
15
     BASEPTR, but with a different specific procedure name:
16
17
     INTERFACE MPI_ALLOC_MEM
18
          SUBROUTINE MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
19
              IMPORT :: MPI_ADDRESS_KIND
              INTEGER INFO, IERROR
20
21
              INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
22
          END SUBROUTINE
23
          SUBROUTINE MPI_ALLOC_MEM_CPTR(SIZE, INFO, BASEPTR, IERROR)
^{24}
              USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
25
              IMPORT :: MPI_ADDRESS_KIND
26
              INTEGER :: INFO, IERROR
              INTEGER(KIND=MPI_ADDRESS_KIND) ::
27
                                                      SIZE
              TYPE(C_PTR) :: BASEPTR
28
29
          END SUBROUTINE
30
     END INTERFACE
31
          The base procedure name of this overloaded function is MPI_ALLOC_MEM_CPTR. The
32
     implied specific procedure names are described in Section 18.1.5.
33
34
         By default, the allocated memory shall be aligned to at least the alignment required
     for load/store accesses of any datatype corresponding to a predefined MPI datatype. The
35
     info argument may be used to specify a desired alternative minimum alignment in bytes for
36
     the allocated memory by setting the value of the key mpi_minimum_memory_alignment to an
37
     integral number equal to a power of two. An implementation may ignore values smaller than
38
     the default required alignment. The info argument can also be used to provide directives
39
     that control the desired location of the allocated memory. Such a directive does not affect
40
     the semantics of the call. The corresponding info values are implementation-dependent. A
41
     null directive value of info = MPI_INFO_NULL is always valid.
42
          The function MPI_ALLOC_MEM may return an error code of class MPI_ERR_NO_MEM
43
     to indicate it failed because memory is exhausted.
44
45
46
47
48
```

shape = (/100, 100/)

1

 $\mathbf{2}$

4 5

6

9

10

11

12

13

14

15

1617

18

19

40 41

43

44

45

46

47

48

MPI_FREE_MEM(base) IN base initial address of memory segment allocated by MPI_ALLOC_MEM (choice) C binding int MPI_Free_mem(void *base) 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 2021to 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 2223less level of indirection). Both arguments are declared to be of same type void* so 24as to facilitate type casting. The Fortran binding is consistent with the C bindings: 25the Fortran MPI_ALLOC_MEM call returns in baseptr the TYPE(C_PTR) pointer or 26the (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 2728 stored at that location. (*End of rationale.*) 29 If MPI_ALLOC_MEM allocates special memory, then a Advice to implementors. 30 design similar to the design of C malloc and free functions has to be used, in order 31to find out the size of a memory segment, when the segment is freed. If no special 32 memory is used, MPI_ALLOC_MEM simply invokes malloc, and MPI_FREE_MEM 33 invokes free. 34 35A call to MPI_ALLOC_MEM can be used in shared memory systems to allocate mem-36 ory in a shared memory segment. (End of advice to implementors.) 37 38 **Example 8.1** Example of use of MPI_ALLOC_MEM, in Fortran with TYPE(C_PTR) pointers. 39 We assume 4-byte REALs. USE mpi_f08 ! or USE mpi (not guaranteed with INCLUDE 'mpif.h') 42USE, 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

```
1
       size = 4 * \text{shape}(1) * \text{shape}(2)
                                                             ! assuming 4 bytes per REAL
\mathbf{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 8.2 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 8.2 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 8.3 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
     8.3
44
45
```

An MPI implementation cannot or may choose not to handle some errors that occur during
 MPI calls. These can include errors that generate exceptions or traps, such as floating point
 errors or access violations. The set of errors that are handled by MPI is implementation dependent. Each such error generates an MPI exception.

 $\mathbf{2}$

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

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 MPI exception 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 10.8.4). 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 36 default initial error handler and associated with predefined communicators. Thus, if the 37 user chooses not to control error handling, every error that MPI handles is treated as fatal. 38 Since (almost) all MPI calls return an error code, a user may choose to handle errors in its 39 main code, by testing the return code of MPI calls and executing a suitable recovery code 40 when the call was not successful. In this case, the error handler MPI_ERRORS_RETURN will 41 be used. Usually it is more convenient and more efficient not to test for errors after each 42MPI call, and have such error handled by a non-trivial MPI error handler. Note that unlike 43 predefined communicators, windows and files do not inherit from the initial error handler, 44as defined in Sections 11.6 and 13.7 respectively. 45

After an error is detected, MPI will provide the user as much information as possible 46 about that error using error classes. Some errors might prevent MPI from completing 47 further API calls successfully and those functions will continue to report errors until the 48

cause of the error is corrected or the user terminates the application. The user can make the determination of whether or not to attempt to continue after detecting such an error.

Advice to users. For example, users may be unable to correct errors corresponding to some error classes, such as MPI_ERR_INTERN. Such errors may cause subsequent MPI calls to complete in error. (End of advice to users.)

Advice to implementors. A high-quality implementation will, to the greatest possible extent, circumscribe the impact of an error, so that normal processing can continue after an error handler was invoked. The implementation documentation will provide information on the possible effect of each class of errors and available recovery actions. (End of advice to implementors.)

An MPI error handler is an opaque object, which is accessed by a handle. MPI calls 14are provided to create new error handlers, to associate error handlers with objects, and to 15test which error handler is associated with an object. C has distinct typedefs for user de-16fined error handling callback functions that accept communicator, file, window, and session 17arguments. In Fortran there are four user routines.

An error handler object is created by a call to MPI_XXX_CREATE_ERRHANDLER, 19where XXX is, respectively, COMM, WIN, FILE, or SESSION. 20

An error handler is attached to a communicator, window, file, or session by a call to 21MPI_XXX_SET_ERRHANDLER. The error handler must be either a predefined error han-22 dler, or an error handler that was created by a call to MPI_XXX_CREATE_ERRHANDLER, 23with matching XXX. An error handler can also be attached to a session using the errorhandler 24 argument to MPI_SESSION_INIT. The predefined error handlers MPI_ERRORS_RETURN and 25MPI_ERRORS_ARE_FATAL can be attached to communicators, windows, files, or sessions. 26

The error handler currently associated with a communicator, window, file, or session 27can be retrieved by a call to MPI_XXX_GET_ERRHANDLER. 28

The MPI function MPI_ERRHANDLER_FREE can be used to free an error handler that 29 was created by a call to MPI_XXX_CREATE_ERRHANDLER. 30

MPI_XXX_GET_ERRHANDLER behave as if a new error handler object is created. That 31is, once the error handler is no longer needed, MPI_ERRHANDLER_FREE should be called 32 with the error handler returned from MPI_XXX_GET_ERRHANDLER to mark the error 33 handler for deallocation. This provides behavior similar to that of MPI_COMM_GROUP 34 and MPI_GROUP_FREE. 35

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

41 42

- 45
- 46
- 47
- 48

1

 $\mathbf{2}$

3

4

5

6 7

8

9

10

11

1213

```
8.3. ERROR HANDLING
                                                                                 393
8.3.1 Error Handlers for Communicators
                                                                                         2
MPI_COMM_CREATE_ERRHANDLER(comm_errhandler_fn, errhandler)
  IN
           comm_errhandler_fn
                                       user defined error handling procedure (function)
  OUT
           errhandler
                                       MPI error handler (handle)
C binding
                                                                                        10
int MPI_Comm_create_errhandler(MPI_Comm_errhandler_function
                                                                                        11
               *comm_errhandler_fn, MPI_Errhandler *errhandler)
                                                                                        12
Fortran 2008 binding
                                                                                        13
MPI_Comm_create_errhandler(comm_errhandler_fn, errhandler, ierror)
                                                                                        14
    PROCEDURE(MPI_Comm_errhandler_function), INTENT(IN) ::
                                                                                        15
               comm_errhandler_fn
                                                                                        16
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
                                                                                        17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                        18
                                                                                        19
Fortran binding
                                                                                        20
MPI_COMM_CREATE_ERRHANDLER(COMM_ERRHANDLER_FN, ERRHANDLER, IERROR)
                                                                                        21
    EXTERNAL COMM_ERRHANDLER_FN
                                                                                        22
    INTEGER ERRHANDLER, IERROR
                                                                                        23
    Creates an error handler that can be attached to communicators.
                                                                                        24
    The user routine should be, in C, a function of type MPI_Comm_errhandler_function, which
                                                                                        25
is defined as
                                                                                        26
typedef void MPI_Comm_errhandler_function(MPI_Comm *comm, int *error_code,
                                                                                        27
               ...);
                                                                                        28
                                                                                        29
    The first argument is the communicator in use. The second is the error code to be
                                                                                        30
returned by the MPI routine that raised the error. If the routine would have returned
                                                                                        ^{31}
MPI_ERR_IN_STATUS, it is the error code returned in the status for the request that caused
                                                                                        32
the error handler to be invoked. The remaining arguments are "varargs" arguments whose
                                                                                        33
number and meaning is implementation-dependent. An implementation should clearly doc-
                                                                                        34
ument these arguments. Addresses are used so that the handler may be written in Fortran.
                                                                                        35
With the Fortran mpi_f08 module, the user routine comm_errhandler_fn should be of the
                                                                                        36
form:
                                                                                        37
ABSTRACT INTERFACE
                                                                                        38
  SUBROUTINE MPI_Comm_errhandler_function(comm, error_code)
                                                                                        39
    TYPE(MPI_Comm) :: comm
                                                                                        40
    INTEGER :: error_code
                                                                                        41
With the Fortran mpi module and mpif.h, the user routine COMM_ERRHANDLER_FN
                                                                                        42
should be of the form:
                                                                                        43
SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)
                                                                                        44
    INTEGER COMM, ERROR_CODE
                                                                                        45
                                                                                        46
                                                                                        47
     Rationale.
                  The variable argument list is provided because it provides an ISO-
```

Unofficial Draft for Comment Only

standard hook for providing additional information to the error handler; without this

	394	CHA	APTER 8.	MPI ENVIRONMENTAL MANAGEMENT
1 2	h	nook, ISO C prohibits addition	onal argum	ents. (End of rationale.)
3 4 5 6 7 8	is a c	s associated with the "paren "global" error handler for a	nt" commu all commu	ommunicator inherits the error handler that inicator. In particular, the user can specify nicators by associating this handler with the ediately after initialization. (<i>End of advice to</i>
9 10	MPI_C	OMM_SET_ERRHANDLER(comm, errh	andler)
11 12	INOL			municator (handle)
13	IN	errhandler		error handler for communicator (handle)
14 15 16 17	C bin int MF	0	PI_Comm c	omm, MPI_Errhandler errhandler)
 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 	MPI_Co TY TY IN Fortra MPI_CO IN At a pred MPI_C	efined error handler, or an COMM_CREATE_ERRHANDL COMM_GET_ERRHANDLER(comm) :: comm ENT(IN) : (OUT) :: ERRHANDI IERROR to a comm error han ER. comm, errh com erro	: errhandler ierror ER, IERROR) municator. The error handler must be either dler created by a call to
37 38 39	C bin int MF	2	PI_Comm c	omm, MPI_Errhandler *errhandler)
40 41 42 43 44	<pre>Fortran 2008 binding MPI_Comm_get_errhandler(comm, errhandler, ierror) TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>			
45 46 47 48	MPI_CO	an binding DMM_GET_ERRHANDLER(COMM, VTEGER COMM, ERRHANDLER,		ER, IERROR)

Retrieves the error handler currently associated with a communicator.

For example, a library function may register at its entry point the current error handler for a communicator, set its own private error handler for this communicator, and restore before exiting the previous error handler.

8.3.2	Error Handlers for Wind	ows	6
			7
			8 9
MPI_\	VIN_CREATE_ERRHANDL	_ER(win_errhandler_fn, errhandler)	9 10
IN	win_errhandler_fn	user defined error handling procedure (function)	11
OUT	errhandler	MPI error handler (handle)	12
			13
C bir	ding		14
	0	er(MPI_Win_errhandler_function	15
	<pre>*win_errhandle</pre>	er_fn, MPI_Errhandler *errhandler)	16
Fortr	an 2008 binding		17 18
	_	vin_errhandler_fn, errhandler, ierror)	18 19
		undler_function), INTENT(IN) :: win_errhandler_fn	20
		NTENT(OUT) :: errhandler	21
I	NTEGER, OPTIONAL, INTE	CNT(OUT) :: ierror	22
Fortr	an binding		23
	0	/IN_ERRHANDLER_FN, ERRHANDLER, IERROR)	24
	XTERNAL WIN_ERRHANDLER		25
	NTEGER ERRHANDLER, IER		26
C	neates an annan handlan th	at can be attached to a window chiest. The user routing	27
		at can be attached to a window object. The user routine pe MPI_Win_errhandler_function which is defined as	28 29
		dler_function(MPI_Win *win, int *error_code,	29 30
oyped);		31
т			32
		indow in use, the second is the error code to be returned.	33
	ACT INTERFACE	le, the user routine win_errhandler_fn should be of the form:	34
		dler_function(win, error_code)	35
	YPE(MPI_Win) :: win		36
	NTEGER :: error_code		37
W 7:+1	the Deuteren and and delte en	densify the second section WIN EDDUANDLED EN descha	38
	the form:	d mpif.h, the user routine WIN_ERRHANDLER_FN should	39 40
		UNCTION(WIN, ERROR_CODE)	41
	NTEGER WIN, ERROR_CODE	-	42
	,,		43
			44
			45
			46
			47
			48

 $\mathbf{2}$

 $\mathbf{5}$

```
1
     MPI_WIN_SET_ERRHANDLER(win, errhandler)
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
```

8.3.	ERR	OR HANDLING	3	397
8.3.3	B Er	ror Handlers for Fi	les	1
				3
MPI	FII E	CREATE ERRHAI	NDLER(file_errhandler_fn, errhandler)	4
IN		file_errhandler_fn		5
				6
Οι)	errhandler	MPI error handler (handle)	7
Ch	indin	a.		9
		•	andler(MPI_File_errhandler_function	10
1110			andler_fn, MPI_Errhandler *errhandler)	11
D (12
		2008 binding	er(file_errhandler_fn, errhandler, ierror)	13
			rrhandler_function), INTENT(IN) ::	14
		file_errha		15 16
	TYPE	(MPI_Errhandler)	, INTENT(OUT) :: errhandler	17
	INTE	GER, OPTIONAL, I	NTENT(OUT) :: ierror	18
Fort	ran l	oinding		19
		0	ER(FILE_ERRHANDLER_FN, ERRHANDLER, IERROR)	20
	EXTE	RNAL FILE_ERRHAN	DLER_FN	21
	INTE	GER ERRHANDLER,	IERROR	22 23
	Creat	es an error handler	that can be attached to a file object. The user routine show	
			MPI_File_errhandler_function, which is defined as	25
type	edef v	void MPI_File_er	<pre>rhandler_function(MPI_File *file, int *error_code</pre>	, 26
);		27
	The f	irst argument is th	e file in use, the second is the error code to be returned.	28
With	n the l	Fortran mpi_f08 m	odule, the user routine file_errhandler_fn should be of the for	°m: 29
		INTERFACE		30 31
SU			rhandler_function(file, error_code)	31
		(MPI_File) :: fi		33
		GER :: error_cod		34
		-	e and mpif.h, the user routine FILE_ERRHANDLER_FN show	uld 35
	f the f			36
SUBR		SER FILE_ERRHANDL	ER_FUNCTION(FILE, ERROR_CODE)	37
		JER FILE, ERRUR_	CODE	38 39
				40
MPI	_FILE	_SET_ERRHANDL	ER(file, errhandler)	41
	OUT	file	file (handle)	42
	001			43
IN		errhandler	new error handler for file (handle)	44
Ch	indin	G		45
	indin MPT T	0	ler(MPI_File file, MPI_Errhandler errhandler)	46 47
			· · · · · · · · · · ·	48

```
1
     Fortran 2008 binding
\mathbf{2}
     MPI_File_set_errhandler(file, errhandler, ierror)
3
         TYPE(MPI_File), INTENT(IN) :: file
4
         TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     Fortran binding
7
     MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
8
         INTEGER FILE, ERRHANDLER, IERROR
9
10
         Attaches a new error handler to a file. The error handler must be either a predefined
11
     error handler, or an error handler created by a call to MPI_FILE_CREATE_ERRHANDLER.
12
13
     MPI_FILE_GET_ERRHANDLER(file, errhandler)
14
15
       IN
                file
                                            file (handle)
16
       OUT
                errhandler
                                            error handler currently associated with file (handle)
17
18
     C binding
19
     int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)
20
21
     Fortran 2008 binding
22
     MPI_File_get_errhandler(file, errhandler, ierror)
23
         TYPE(MPI_File), INTENT(IN) :: file
^{24}
         TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
     Fortran binding
27
     MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
28
         INTEGER FILE, ERRHANDLER, IERROR
29
30
         Retrieves the error handler currently associated with a file.
31
32
     8.3.4 Error Handlers for Sessions
33
34
35
     MPI_SESSION_CREATE_ERRHANDLER(session_errhandler_fn, errhandler)
36
37
       IN
                session_errhandler_fn
                                            user defined error handling procedure (function)
38
       OUT
                errhandler
                                            MPI error handler (handle)
39
40
     C binding
41
     int MPI_Session_create_errhandler(MPI_Session_errhandler_function
42
                    *session_errhandler_fn, MPI_Errhandler *errhandler)
43
44
     Fortran 2008 binding
45
     MPI_Session_create_errhandler(session_errhandler_fn, errhandler, ierror)
46
         PROCEDURE(MPI_Session_errhandler_function), INTENT(IN) ::
47
                     session_errhandler_fn
48
         TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1
Fortran binding	2
MPI_SESSION_CREATE_ERRHANDLER(SESSION_ERRHANDLER_FN, ERRHANDLER, IERROR)	3
EXTERNAL SESSION_ERRHANDLER_FN	4
INTEGER ERRHANDLER, IERROR	5
	6
Creates an error handler that can be attached to a session object. In C, the	7
${\tt session_errhandler_fn} \ {\tt argument} \ {\tt should} \ {\tt be} \ {\tt a} \ {\tt function} \ {\tt of} \ {\tt type} \ {\tt MPI_Session_errhandler_function},$	8
which is defined as	9
<pre>typedef void MPI_Session_errhandler_function(MPI_Session *session,</pre>	10
<pre>int *error_code,);</pre>	11
The first argument is the session in use the second is the amon and to be returned	12
The first argument is the session in use, the second is the error code to be returned. With the Fortran mpi_f08 module, the session_errhandler_fn argument should be of the	13 14
form:	15
ABSTRACT INTERFACE	16
SUBROUTINE MPI_Session_errhandler_function(session, error_code)	17
TYPE(MPI_Session) :: session	18
INTEGER :: error_code	19
	20
With the Fortran mpi module and mpif.h, the SESSION_ERRHANDLER_FN argument	20
should be of the form:	22
SUBROUTINE SESSION_ERRHANDLER_FUNCTION(SESSION, ERROR_CODE)	23
INTEGER SESSION, ERROR_CODE	24
	25
	26
MPI_SESSION_SET_ERRHANDLER(session, errhandler)	27
INOUT session (handle)	28
IN errhandler new error handler for session (handle)	29
	30
C binding	31
int MPI_Session_set_errhandler(MPI_Session session,	32
MPI_Errhandler errhandler)	33
	34
Fortran 2008 binding	35
MPI_Session_set_errhandler(session, errhandler, ierror)	36
TYPE(MPI_Session), INTENT(IN) :: session	37
TYPE(MPI_Errhandler), INTENT(IN) :: errhandler	38
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	39
Fortran binding	40
MPI_SESSION_SET_ERRHANDLER(SESSION, ERRHANDLER, IERROR)	41
	42
INTEGER SESSION, ERRHANDLER, IERROR	43
Attaches a new error handler to a session. The error handler must be either a pre-	44
defined error handler, or an error handler created by a call to	45
MPI_SESSION_CREATE_ERRHANDLER.	46
	47
	48

1 MPI_SESSION_GET_ERRHANDLER(session, errhandler) 2 IN session (handle) 3 OUT errhandler error handler currently associated with session 4 (handle) 56 C binding 7 int MPI_Session_get_errhandler(MPI_Session session, 8 9 MPI_Errhandler *errhandler) 10 Fortran 2008 binding 11 MPI_Session_get_errhandler(session, errhandler, ierror) 12TYPE(MPI_Session), INTENT(IN) :: session 13 TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler 14INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1516Fortran binding 17 MPI_SESSION_GET_ERRHANDLER(SESSION, ERRHANDLER, IERROR) 18 INTEGER SESSION, ERRHANDLER, IERROR 19Retrieves the error handler currently associated with a session. 20218.3.5 Freeing Errorhandlers and Retrieving Error Strings 22 2324MPI_ERRHANDLER_FREE(errhandler) 2526INOUT errhandler MPI error handler (handle) 2728C binding 29 int MPI_Errhandler_free(MPI_Errhandler *errhandler) 30 31 Fortran 2008 binding MPI_Errhandler_free(errhandler, ierror) 32 33 TYPE(MPI_Errhandler), INTENT(INOUT) :: errhandler 34INTEGER, OPTIONAL, INTENT(OUT) :: ierror 35 Fortran binding 36 MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR) 37 INTEGER ERRHANDLER, IERROR 38 39 Marks the error handler associated with errhandler for deallocation and sets errhandler 40to MPI_ERRHANDLER_NULL. The error handler will be deallocated after all the objects 41 associated with it (communicator, window, or file) have been deallocated. 4243 44 4546 47 48

MPI_ERF	ROR_STRING(errorcod	le, string, resultlen)	1
IN	errorcode	Error code returned by an MPI routine	2
OUT	string	Text that corresponds to the errorcode	$\frac{3}{4}$
OUT	resultlen	-	4 5
001	resultien	Length (in printable characters) of the result returned in string	6
		Teturned in string	7
C bindi	nœ		8
	•	errorcode, char *string, int *resultlen)	9
	_	siloitoat, and sporting, interiobation,	10
	2008 binding		11
	•	e, string, resultlen, ierror)	12
	EGER, INTENT(IN) :	: errorcode _ERROR_STRING), INTENT(OUT) :: string	13
	EGER, INTENT(OUT)		14 15
		FENT(OUT) :: ierror	16
			17
	binding		18
	EGER ERRORCODE, RE	E, STRING, RESULTLEN, IERROR)	19
	RACTER*(*) STRING	SULLEN, LEMON	20
			21
		associated with an error code or class. The argument string	22
-	0	at least MPI_MAX_ERROR_STRING characters long.	23
		actually written is returned in the output argument, resultlen. s be thread-safe, as defined in Section 10.6. It is one of the	24 25
	•	d before MPI is initialized or after MPI is finalized.	25 26
10 10 10 10	ines that may be cane	a before with is informized of after with is infanzed.	27
Rat	tionale. The form of	this function was chosen to make the Fortran and C bindings	28
\sin	ilar. A version that	returns a pointer to a string has two difficulties. First, the	29
	e e e e e e e e e e e e e e e e e e e	tically allocated and different for each error message (allowing	30
		v successive calls to MPI_ERROR_STRING to point to the	31
	0,	, in Fortran, a function declared as returning CHARACTER*(*)	32
can	not be referenced in,	for example, a PRINT statement. (End of rationale.)	33
		▼	34
8.4 E	rror Codes and Cl	asses	35 36
(T)			27

The error codes returned by MPI are left entirely to the implementation (with the exception of MPI_SUCCESS). This is done to allow an implementation to provide as much information as possible in the error code (for use with MPI_ERROR_STRING).

All MPI function calls shall return MPI_SUCCESS if and only if the specification of that function has been fulfilled at the point of return. For multiple completion functions, if the function returns MPI_ERR_IN_STATUS, the error code in each status object shall be set to MPI_SUCCESS if and only if the specification of the operation represented by the corresponding MPI_Request has been fulfilled at the point of return.

When an operation raises an error, it may not satisfy its specification (for example, a synchronizing operation may not have synchronized) and the content of the output buffers, targeted memory, or output parameters is undefined. However, a valid error code shall

Unofficial Draft for Comment Only

37

38

39

40 41

42

43

44

4546

47

1 always be set when an operation raises an error, whether in the return value, error field in $\mathbf{2}$ the status object, or element in an array of error codes. 3 To make it possible for an application to interpret an error code, the routine 4 MPI_ERROR_CLASS converts any error code into one of a small set of standard error codes, $\mathbf{5}$ called *error classes*. Valid error classes are shown in Table 8.1 and Table 8.2. 6 The error classes are a subset of the error codes: an MPI function may return an error 7class number; and the function MPI_ERROR_STRING can be used to compute the error 8 string associated with an error class. The values defined for MPI error classes are valid MPI 9 error codes. 10 The error codes satisfy, 11 $0 = MPI_SUCCESS < MPI_ERR_... < MPI_ERR_LASTCODE.$ 1213 Rationale. The difference between MPI_ERR_UNKNOWN and MPI_ERR_OTHER is that 14MPI_ERROR_STRING can return useful information about MPI_ERR_OTHER. 1516Note that $MPI_SUCCESS = 0$ is necessary to be consistent with C practice; the sepa-17 ration of error classes and error codes allows us to define the error classes this way. 18 Having a known LASTCODE is often a nice sanity check as well. (*End of rationale.*) 19 2021MPI_ERROR_CLASS(errorcode, errorclass) 22 23IN errorcode Error code returned by an MPI routine 24 Error class associated with errorcode OUT errorclass 2526C binding 27int MPI_Error_class(int errorcode, int *errorclass) 2829Fortran 2008 binding 30 MPI_Error_class(errorcode, errorclass, ierror) 31 INTEGER, INTENT(IN) :: errorcode 32 INTEGER, INTENT(OUT) :: errorclass 33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 34Fortran binding 35 MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR) 36 INTEGER ERRORCODE, ERRORCLASS, IERROR 37 38 The function MPI_ERROR_CLASS maps each standard error code (error class) onto 39 itself. 40 This function must always be thread-safe, as defined in Section 10.6. It is one of the 41 few routines that may be called before MPI is initialized or after MPI is finalized. 4243

8.5 Error Classes, Error Codes, and Error Handlers

⁴⁵ Users may want to write a layered library on top of an existing MPI implementation, and ⁴⁷ this library may have its own set of error codes and classes. An example of such a library ⁴⁸ is an I/O library based on MPI, see Chapter 13. For this purpose, functions are needed to:

Unofficial Draft for Comment Only

MPI_SUCCESS	No error	1
MPI_ERR_BUFFER	Invalid buffer pointer	2
MPI_ERR_COUNT	Invalid count argument	3
MPI_ERR_TYPE	Invalid datatype argument	4
MPI_ERR_TAG	Invalid tag argument	5
MPI_ERR_COMM	Invalid communicator	6
MPI_ERR_RANK	Invalid rank	7
MPI_ERR_REQUEST	Invalid request (handle)	8
MPI_ERR_ROOT	Invalid root	9
MPI_ERR_GROUP	Invalid group	10
MPI_ERR_OP	Invalid operation	11
MPI_ERR_TOPOLOGY	Invalid topology	12
MPI_ERR_DIMS	Invalid dimension argument	13
MPI_ERR_ARG	Invalid argument of some other kind	14
MPI_ERR_UNKNOWN	Unknown error	15
MPI_ERR_TRUNCATE	Message truncated on receive	16
MPI_ERR_OTHER	Known error not in this list	17
MPI_ERR_INTERN	Internal MPI (implementation) error	18
MPI_ERR_IN_STATUS	Error code is in status	19
MPI_ERR_PENDING	Pending request	20
MPI_ERR_KEYVAL	Invalid keyval has been passed	21
MPI_ERR_PROC_ABORTED	Operation failed because a peer process has	22
	aborted	23
MPI_ERR_NO_MEM	MPI_ALLOC_MEM failed because memory	24
	is exhausted	25
MPI_ERR_BASE	Invalid base passed to MPI_FREE_MEM	26
MPI_ERR_INFO_KEY	Key longer than MPI_MAX_INFO_KEY	27
MPI_ERR_INFO_VALUE	Value longer than MPI_MAX_INFO_VAL	28
MPI_ERR_INFO_NOKEY	Invalid key passed to MPI_INFO_DELETE	29
MPI_ERR_SPAWN	Error in spawning processes	30
MPI_ERR_PORT	Invalid port name passed to	31
	MPI_COMM_CONNECT	32
MPI_ERR_SERVICE	Invalid service name passed to	33
	MPI_UNPUBLISH_NAME	34
MPI_ERR_NAME	Invalid service name passed to	35
	MPI_LOOKUP_NAME	36
MPI_ERR_WIN	Invalid win argument	37
MPI_ERR_SIZE	Invalid size argument	38
MPI_ERR_DISP	Invalid disp argument	39
MPI_ERR_INFO	Invalid info argument	40
MPI_ERR_LOCKTYPE	Invalid locktype argument	41
MPI_ERR_ASSERT	Invalid assert argument	42
MPI_ERR_RMA_CONFLICT	Conflicting accesses to window	43
MPI_ERR_RMA_SYNC	Wrong synchronization of RMA calls	44
		45
		46

Table 8.1: Error classes (Part 1)

Unofficial Draft for Comment Only

1	MPI_ERR_RMA_RANGE	Target memory is not part of the win- dow (in the case of a window created
3		with MPI_WIN_CREATE_DYNAMIC, tar-
4		get memory is not attached)
5	MPI_ERR_RMA_ATTACH	Memory cannot be attached (e.g., because
6		of resource exhaustion)
7	MPI_ERR_RMA_SHARED	Memory cannot be shared (e.g., some pro-
8		cess in the group of the specified commu-
9		nicator cannot expose shared memory)
10	MPI_ERR_RMA_FLAVOR	Passed window has the wrong flavor for the
11		called function
12	MPI_ERR_FILE	Invalid file handle
13	MPI_ERR_NOT_SAME	Collective argument not identical on all
14		processes, or collective routines called in
15		a different order by different processes
16	MPI_ERR_AMODE	Error related to the amode passed to
17		MPI_FILE_OPEN
18	MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to
19		MPI_FILE_SET_VIEW
20	MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on
21		a file which supports sequential access only
22	MPI_ERR_NO_SUCH_FILE	File does not exist
23	MPI_ERR_FILE_EXISTS	File exists
24	MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)
25	MPI_ERR_ACCESS	Permission denied
26	MPI_ERR_NO_SPACE	Not enough space
27	MPI_ERR_QUOTA	Quota exceeded
28	MPI_ERR_READ_ONLY	Read-only file or file system
29 30	MPI_ERR_FILE_IN_USE	File operation could not be completed, as
31	MPI_ERR_DUP_DATAREP	the file is currently open by some process Conversion functions could not be regis-
32	MFI_ERK_DOF_DATAKEP	tered because a data representation identi-
33		fier that was already defined was passed to
34		MPI_REGISTER_DATAREP
35	MPI_ERR_CONVERSION	An error occurred in a user supplied data
36		conversion function.
37	MPI_ERR_IO	Other I/O error
38	MPI_ERR_SESSION	Invalid session argument
39	MPI_ERR_LASTCODE	Last error code
40	•	
41		
42	Table 8.2: Erre	or classes (Part 2)
43		
44		
45		
46		
47		
48		

1. add a new error class to the ones an with implementation aready knows.	1
2 aggregiste oppen og deg with this oppen class as that MDL EDDOD CLASS works	2 3
3. associate strings with these error codes, so that MPI_ERROR_STRING works.	4
	5 6
	7
to free error classes or codes: it is not expected that an application will generate them in	8 9
	10
	12
	13
	14 15
	16
int MPI_Add_error_class(int *errorclass)	17
Fortran 2008 binding	18 19
MPI_Add_error_class(errorclass, ierror)	20
INTEGER, INTENT(OUT) :: errorclass	21
	22
Torotal bliding	23 24
	25
The second se	26
	27
<i>Rationale.</i> To avoid conflicts with existing error codes and classes, the value is set by the implementation and not by the user. (<i>End of rationale.</i>)	28 29 30
Advice to implementors. A high-quality implementation will return the value for	31
a new errorclass in the same deterministic way on all processes. (End of advice to	32 33
implementors.)	34
Advice to users. Since a call to MPI_ADD_ERROR_CLASS is local, the same errorclass	35
may not be returned on an processes that make this can. Thus, it is not safe to assume	36
that registering a new error on a set of processes at the same time will yield the same	37 38
errorciass on an of the processes. However, if an implementation returns the new	39
	40
However, even if a deterministic algorithm is used, the value can vary across processes.	41
This can happen, for example, if different but overlapping groups of processes make	42 43
a series of cans. As a result of these issues, getting the same error on multiple	±3
	15
	16

The value of MPI_ERR_LASTCODE is a constant value and is not affected by new userdefined error codes and classes. Instead, a predefined attribute key MPI_LASTUSEDCODE is 48

```
406
                                    CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT
1
      associated with MPI_COMM_WORLD. The attribute value corresponding to this key is the
\mathbf{2}
      current maximum error class including the user-defined ones. This is a local value and may
3
      be different on different processes. The value returned by this key is always greater than or
4
     equal to MPI_ERR_LASTCODE.
5
           Advice to users. The value returned by the key MPI_LASTUSEDCODE will not change
6
           unless the user calls a function to explicitly add an error class/code. In a multi-
7
           threaded environment, the user must take extra care in assuming this value has not
8
           changed. Note that error codes and error classes are not necessarily dense. A user
9
           may not assume that each error class below MPI_LASTUSEDCODE is valid. (End of
10
           advice to users.)
11
12
13
14
      MPI_ADD_ERROR_CODE(errorclass, errorcode)
15
       IN
                 errorclass
                                              error class (integer)
16
17
       OUT
                 errorcode
                                              new error code to be associated with errorclass
18
                                              (integer)
19
20
      C binding
21
      int MPI_Add_error_code(int errorclass, int *errorcode)
22
      Fortran 2008 binding
23
      MPI_Add_error_code(errorclass, errorcode, ierror)
^{24}
          INTEGER, INTENT(IN) :: errorclass
25
          INTEGER, INTENT(OUT) :: errorcode
26
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
      Fortran binding
29
      MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR)
30
          INTEGER ERRORCLASS, ERRORCODE, IERROR
31
          Creates new error code associated with errorclass and returns its value in errorcode.
32
33
                      To avoid conflicts with existing error codes and classes, the value of the
          Rationale.
34
           new error code is set by the implementation and not by the user. (End of rationale.)
35
36
           Advice to implementors.
                                      A high-quality implementation will return the value for
37
           a new errorcode in the same deterministic way on all processes. (End of advice to
38
           implementors.)
39
40
41
42
      MPI_ADD_ERROR_STRING(errorcode, string)
43
       IN
                  errorcode
                                              error code or class (integer)
44
```

C binding int MPI_Add_error_string(int errorcode, const char *string)

IN

45 46 47

48

string

text corresponding to errorcode (string)

Fortran 2008 binding MPI_Add_error_string(errorcode, string, ierror) INTEGER, INTENT(IN) :: errorcode CHARACTER(LEN=*), INTENT(IN) :: string INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR) INTEGER ERRORCODE, IERROR CHARACTER*(*) STRING Associates an error string with an error code or class. The string must be no more than MPI_MAX_ERROR_STRING characters long. The length of the string is as defined in the calling language. The length of the string does not include the null terminator in C. Trailing blanks will be stripped in Fortran. Calling MPI_ADD_ERROR_STRING for an errorcode that already has a string will replace the old string with the new string. It is erroneous to call MPI_ADD_ERROR_STRING for an error code or class with a value < MPI_ERR_LASTCODE. If MPI_ERROR_STRING is called when no string has been set, it will return a empty string (all spaces in Fortran, "" in C). Section 8.3 describes the methods for creating and associating error handlers with communicators, files, windows, and sessions. MPI_COMM_CALL_ERRHANDLER(comm, errorcode) IN communicator with error handler (handle) comm IN errorcode error code (integer)

C binding

int MPI_Comm_call_errhandler(MPI_Comm com	mm, int	errorcode)	
Fortran 2008 binding			

```
MPI_Comm_call_errhandler(comm, errorcode, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: errorcode
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_COMM_CALL_ERRHANDLER(COMM, ERRORCODE, IERROR)
INTEGER COMM, ERRORCODE, IERROR
```

This function invokes the error handler assigned to the communicator with the error code supplied. This function returns MPI_SUCCESS in C and the same value in IERROR if the error handler was successfully called (assuming the process is not aborted and the error handler returns).

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

32

33

34

35

36

37

38 39

40

41

42

```
1
     MPI_WIN_CALL_ERRHANDLER(win, errorcode)
2
       IN
                                             window with error handler (handle)
                 win
3
       IN
                 errorcode
                                             error code (integer)
4
5
6
     C binding
7
     int MPI_Win_call_errhandler(MPI_Win win, int errorcode)
8
     Fortran 2008 binding
9
     MPI_Win_call_errhandler(win, errorcode, ierror)
10
          TYPE(MPI_Win), INTENT(IN) :: win
11
          INTEGER, INTENT(IN) :: errorcode
12
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     Fortran binding
15
     MPI_WIN_CALL_ERRHANDLER(WIN, ERRORCODE, IERROR)
16
          INTEGER WIN, ERRORCODE, IERROR
17
          This function invokes the error handler assigned to the window with the error code
18
     supplied. This function returns MPI_SUCCESS in C and the same value in IERROR if the
19
     error handler was successfully called (assuming the process is not aborted and the error
20
     handler returns).
21
22
                              In contrast to communicators, the error handler
           Advice to users.
23
           MPI_ERRORS_ARE_FATAL is associated with a window when it is created. (End of
24
           advice to users.)
25
26
27
     MPI_FILE_CALL_ERRHANDLER(fh, errorcode)
28
29
                                             file with error handler (handle)
       IN
                 fh
30
       IN
                 errorcode
                                             error code (integer)
^{31}
32
     C binding
33
     int MPI_File_call_errhandler(MPI_File fh, int errorcode)
34
35
     Fortran 2008 binding
36
     MPI_File_call_errhandler(fh, errorcode, ierror)
37
          TYPE(MPI_File), INTENT(IN) :: fh
38
          INTEGER, INTENT(IN) :: errorcode
39
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     Fortran binding
42
     MPI_FILE_CALL_ERRHANDLER(FH, ERRORCODE, IERROR)
          INTEGER FH, ERRORCODE, IERROR
43
44
          This function invokes the error handler assigned to the file with the error code supplied.
45
     This function returns MPI_SUCCESS in C and the same value in IERROR if the error handler
46
     was successfully called (assuming the process is not aborted and the error handler returns).
47
```

408

1 Advice to users. The default error handler for files is MPI_ERRORS_RETURN. (End of 2 advice to users.) 3 4 MPI_SESSION_CALL_ERRHANDLER(session, errorcode) IN session session with error handler (handle) IN errorcode error code (integer) 10 C binding 11 int MPI_Session_call_errhandler(MPI_Session session, int errorcode) 12Fortran 2008 binding 13 MPI_Session_call_errhandler(session, errorcode, ierror) 14TYPE(MPI_Session), INTENT(IN) :: session 15INTEGER, INTENT(IN) :: errorcode 16INTEGER, OPTIONAL, INTENT(OUT) :: ierror 17

Fortran binding

MPI_SESSION_CALL_ERRHANDLER(SESSION, ERRORCODE, IERROR) INTEGER SESSION, ERRORCODE, IERROR

This function invokes the error handler assigned to the session with the error code supplied. This function returns MPI_SUCCESS in C and the same value in IERROR if the error handler was successfully called (assuming the process is not aborted and the error handler returns).

Advice to users. Users are warned that handlers should not be called recursively with MPI_COMM_CALL_ERRHANDLER, MPI_FILE_CALL_ERRHANDLER, MPI_WIN_CALL_ERRHANDLER, or MPI_SESSION_CALL_ERRHANDLER. Doing this can create a situation where an infinite recursion is created. This can occur if MPI_COMM_CALL_ERRHANDLER, MPI_FILE_CALL_ERRHANDLER, MPI_WIN_CALL_ERRHANDLER, or MPI_SESSION_CALL_ERRHANDLER, MPI_WIN_CALL_ERRHANDLER, or MPI_SESSION_CALL_ERRHANDLER is called inside an error handler.

Error codes and classes are associated with a process. As a result, they may be used in any error handler. Error handlers should be prepared to deal with any error code they are given. Furthermore, it is good practice to only call an error handler with the appropriate error codes. For example, file errors would normally be sent to the file error handler. (*End of advice to users.*)

8.6 Timers and Synchronization

MPI defines a timer. A timer is specified even though it is not "message-passing," because timing parallel programs is important in "performance debugging" and because existing timers (both in POSIX 1003.1-1988 and 1003.4D 14.1 and in Fortran 90) are either inconvenient or do not provide adequate access to high resolution timers. See also Section 2.6.4.

MPI_WTIME()

Unofficial Draft for Comment Only

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
     C binding
\mathbf{2}
     double MPI_Wtime(void)
3
     Fortran 2008 binding
4
     DOUBLE PRECISION MPI_Wtime()
5
6
     Fortran binding
\overline{7}
     DOUBLE PRECISION MPI_WTIME()
8
          MPI_WTIME returns a floating-point number of seconds, representing elapsed wall-
9
     clock time since some time in the past.
10
          The "time in the past" is guaranteed not to change during the life of the process.
11
     The user is responsible for converting large numbers of seconds to other units if they are
12
     preferred.
13
          This function is portable (it returns seconds, not "ticks"), it allows high-resolution,
14
     and carries no unnecessary baggage. One would use it like this:
15
16
     {
17
          double starttime, endtime;
18
          starttime = MPI_Wtime();
19
          ... stuff to be timed
                                      . . .
20
          endtime
                      = MPI_Wtime();
21
          printf("That took %f seconds\n", endtime-starttime);
22
     }
23
^{24}
          The times returned are local to the node that called them. There is no requirement
25
     that different nodes return "the same time." (But see also the discussion of
26
     MPI_WTIME_IS_GLOBAL in Section 8.1.2).
27
28
29
     MPI_WTICK()
30
     C binding
^{31}
     double MPI_Wtick(void)
32
33
     Fortran 2008 binding
34
     DOUBLE PRECISION MPI_Wtick()
35
36
     Fortran binding
37
     DOUBLE PRECISION MPI_WTICK()
38
          MPI_WTICK returns the resolution of MPI_WTIME in seconds. That is, it returns,
39
     as a double precision value, the number of seconds between successive clock ticks. For
40
     example, if the clock is implemented by the hardware as a counter that is incremented
41
     every millisecond, the value returned by MPI_WTICK should be (10^{-3}).
42
43
44
45
46
47
48
```

Chapter 9

The Info Object

Many of the routines in MPI take an argument info. info is an opaque object with a handle of type MPI_Info in C and Fortran with the mpi_f08 module, and INTEGER in Fortran with the mpi module or the include file mpif.h. It stores an unordered set of (key,value) pairs (both key and value are strings). A key can have only one value. MPI reserves several keys and requires that if an implementation uses a reserved key, it must provide the specified functionality. An implementation is not required to support these keys and may support any others not reserved by MPI.

Some info hints allow the MPI library to restrict its support for certain operations in order to improve performance or resource utilization. If an application provides such an info hint, it must be compatible with any changes in the behavior of the MPI library that are allowed by the info hint.

An implementation must support info objects as caches for arbitrary (key,value) pairs, regardless of whether it recognizes the key. Each function that takes hints in the form of an MPI_Info must be prepared to ignore any key it does not recognize. This description of info objects does not attempt to define how a particular function should react if it recognizes a key but not the associated value. MPI_INFO_GET_NKEYS, MPI_INFO_GET_NTHKEY, MPI_INFO_GET_VALUELEN, 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 argument to a nonblocking routine, it is parsed before that routine returns, so that it may be 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 7integer 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 47Fortran binding 48

	SET(INFO, KEY, VALUE, IE ER INFO, IERROR	RROR)	12
CHARA	CTER*(*) KEY, VALUE		3
	NEO SET adda tha (key yelve) noin to info and accomides the value if a value for	4
) pair to info, and overrides the value if a value for	5
	· · ·	d value are null-terminated strings in C. In Fortran,	6
0	•	value are stripped. If either key or value are larger MPI_ERR_INFO_KEY or MPI_ERR_INFO_VALUE are	7
raised, resp	,	MFI_ERR_INFO_RET OF MFI_ERR_INFO_VALUE are	8
Taiseu, Test	pectively.		9
			10
MPI_INFO	_DELETE(info, key)		11
INOUT	info	info object (handle)	12 13
IN	key	key (string)	14
			15
C binding	y		16
-	nfo_delete(MPI_Info info	. const char *kev)	17
		,	18
	008 binding		19
	delete(info, key, ierror)		20
	<pre>MPI_Info), INTENT(IN) :: CTER(LEN=*), INTENT(IN)</pre>		21
	ER, OPTIONAL, INTENT(OUT)		22
	ER, OFIIONAL, INIENI(001)) IEIIUI	23
Fortran binding			24
	DELETE(INFO, KEY, IERROR))	25 26
	ER INFO, IERROR		20 27
CHARA	CTER*(*) KEY		21
MPI_I	NFO_DELETE deletes a (key,	value) pair from info. If key is not defined in info,	29
	ses an error of class MPI_ERR		30
			31
			32
	_GET(info, key, valuelen, value		33
IN	info	info object (handle)	34
IN	key	key (string)	35
IN	valuelen	length of value arg (integer)	36 37
OUT	value	value (string)	38
OUT	flag	true if key defined, false if not (boolean)	39
			40
C binding	y		41
		onst char *key, int valuelen, char *value,	42
	int *flag)		43
-	-		44
	008 binding		45
	MPI_Info_get(info, key, valuelen, value, flag, ierror)		
	<pre>MPI_Info), INTENT(IN) :: CTEP(LEN-*) INTENT(IN)</pre>		47
CHARA	CTER(LEN=*), INTENT(IN)	rey	48

```
1
          INTEGER, INTENT(IN) :: valuelen
\mathbf{2}
          CHARACTER(LEN=valuelen), INTENT(OUT) :: value
3
          LOGICAL, INTENT(OUT) :: flag
4
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     Fortran binding
6
     MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)
7
          INTEGER INFO, VALUELEN, IERROR
8
          CHARACTER*(*) KEY, VALUE
9
          LOGICAL FLAG
10
11
          This function retrieves the value associated with key in a previous call to
12
     MPI_INFO_SET. If such a key exists, it sets flag to true and returns the value in value,
13
     otherwise it sets flag to false and leaves value unchanged. valuelen is the number of characters
14
     available in value. If it is less than the actual size of the value, the value is truncated. In
15
     C, valuelen should be one less than the amount of allocated space to allow for the null
16
     terminator.
17
          If key is larger than MPI_MAX_INFO_KEY, the call is erroneous.
18
19
     MPI_INFO_GET_VALUELEN(info, key, valuelen, flag)
20
21
       IN
                 info
                                              info object (handle)
22
       IN
                                              key (string)
                 key
23
       OUT
                 valuelen
                                              length of value arg (integer)
^{24}
25
       OUT
                                              true if key defined, false if not (boolean)
                 flag
26
27
     C binding
28
     int MPI_Info_get_valuelen(MPI_Info info, const char *key, int *valuelen,
29
                     int *flag)
30
     Fortran 2008 binding
31
     MPI_Info_get_valuelen(info, key, valuelen, flag, ierror)
32
          TYPE(MPI_Info), INTENT(IN) :: info
33
34
         CHARACTER(LEN=*), INTENT(IN) :: key
          INTEGER, INTENT(OUT) :: valuelen
35
          LOGICAL, INTENT(OUT) :: flag
36
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
     Fortran binding
39
     MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
40
          INTEGER INFO, VALUELEN, IERROR
41
          CHARACTER*(*) KEY
42
          LOGICAL FLAG
43
          Retrieves the length of the value associated with key. If key is defined, valuelen is set to
44
     the length of its associated value and flag is set to true. If key is not defined, valuelen is not
45
     touched and flag is set to false. The length returned in C does not include the end-of-string
46
47
     character.
          If key is larger than MPI_MAX_INFO_KEY, the call is erroneous.
48
```

MPI INF	O GET STRING(info, key, buflen, value, flag)	1
- IN	info	info object (handle)	2
IN	key	key (string)	3
INOUT	buflen	length of buffer (integer)	4 5
			6
OUT	value	value (string)	7
OUT	flag	true if key defined, false if not (boolean)	8
C bindi	no		9 10
	_Info_get_strin	g(MPI_Info info, const char *key, int *buflen,	11 12
		.ue, int *flag)	12
	2008 binding		14
		fo, key, buflen, value, flag, ierror) TENT(IN) :: info	15
		INTENT(IN) :: key	16
	EGER, INTENT(IN		17
CHAI	RACTER(LEN=*),	INTENT(OUT) :: value	18 19
	ICAL, INTENT(OU	<u> </u>	20
INTI	EGER, OPTIONAL,	INTENT(OUT) :: ierror	21
Fortran	binding		22
		FO, KEY, BUFLEN, VALUE, FLAG, IERROR)	23
	EGER INFO, BUFL		24 25
	RACTER*(*) KEY, ICAL FLAG	VALOE	26
			27
		es the value associated with key in a previous call to a key exists, it sets flag to true and returns the value in value,	28
		se and leaves value unchanged. buflen on input is the size of the	29
		tput of buflen it is the size of the buffer needed to store the value	30 31
-		into the function is less than the actual size needed to store the	32
value stri	ing (including nul	l terminator in C), the value is truncated. On return, the value	33
		e required buffer size to hold the value string. If buflen is set to	34
,		C, buflen includes the required space for the null terminator. In	35
greater the		null terminated string in all cases where the buflen input value is	36
0		IPI_MAX_INFO_KEY, the call is erroneous.	37
ii ke	y is larger than it		38
		he MPI_INFO_GET_STRING function can be used to obtain the	39 40
	-	buffer for a value string by setting the buffen to 0. The returned	41
		sed to allocate memory before calling MPI_INFO_GET_STRING	42
aga	in to obtain the v	value string. (End of advice to users.)	43
			44
			45
			46

```
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
int MPI	<pre>inio_create_env(int argc,</pre>	char argv[], MPI_Info *info)	45
Fortran 2	2008 binding		46
MPI_Info	_create_env(info, ierror)		47
			48

1	
1	TYPE(MPI_Info), INTENT(OUT) :: info
2	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3	Fortran binding
4	MPI_INFO_CREATE_ENV(INFO, IERROR)
5	INTEGER INFO, IERROR
6	INTEGER INFO, TERROR
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 10.2.1; 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 10.6. 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.
21	In some circumstances (e.g., when passing 0 to argc and NULL to argv in C or in Fortran
22	where such arguments do not exist), the info object may not be populated or may be
23	populated incompletely because this procedure is local and the implementation may
24	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	
44	
45 46	
40 47	
47	
10	

Chapter 10

Process Initialization, Creation, and Management

10.1 Introduction

MPI is primarily concerned with communication rather than process or resource management. However, it is necessary to address these issues to some degree in order to define a useful framework for communication. This chapter presents a set of MPI interfaces that allows for 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 10.2) communicator. In the Sessions Model (Section 10.3), 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 10.7), 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

420 CHAPTER 10. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

¹ propriate for the broad spectrum of existing and potential resource and process controllers. ² Resource control can encompass a wide range of abilities, including adding and deleting ³ nodes from a virtual parallel machine, reserving and scheduling resources, managing com-⁴ pute partitions of an MPP, and returning information about available resources. MPI as-⁵ sumes that resource control is provided externally — probably by computer vendors, in the ⁶ case of tightly coupled systems, or by a third party software package when the environment ⁷ is a cluster of workstations.

⁸ The reasons for including process management in MPI are both technical and practical. ⁹ Important classes of message-passing applications require process control. These include ¹⁰ task farms, serial applications with parallel modules, and problems that require a run-time ¹¹ assessment of the number and type of processes that should be started. On the practical ¹² side, users of workstation clusters who are migrating from PVM to MPI may be accustomed ¹³ to using PVM's capabilities for process and resource management. The lack of these features ¹⁴ would be a practical stumbling block to migration.

15 16

17

18 19

20

21

22 23

24

25 26

27

28

29

30

 31

35 36

37 38 The following goals are central to the design of MPI process management:

- The MPI process model must apply to the vast majority of current parallel environments. These include everything from tightly integrated MPPs to heterogeneous networks of workstations.
 - MPI must not take over operating system responsibilities. It should instead provide a clean interface between an application and system software.
 - MPI must guarantee communication determinism in the presense of dynamic processes, i.e., dynamic process management must not introduce unavoidable race conditions.
 - MPI must not contain features that compromise performance.
- The *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.
 - 10.2 The
 - 2 The World Model
 - 10.2.1 Starting MPI Processes
- When using the World Model, MPI is initialized by calling either MPI_INIT or MPI_INIT_THREAD.
- 42 43

44

```
MPI_INIT()
```

 45 C binding

```
46 int MPI_Init(int *argc, char ***argv)
47
```

48 Fortran 2008 binding

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

Fortran binding

MPI_INIT(IERROR) INTEGER IERROR

In the World Model, an MPI program must contain exactly one call to an MPI initialization routine: MPI_INIT or 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 10.4. 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 43 an error before the error handling capabilities are fully operational, in which cases 44 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 46 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 48

Unofficial Draft for Comment Only

 31

case, before MPI is initialized, the MPI_ERRORS_ABORT error handler may abort what
 would have become multiple MPI processes.

When a failure occurs during MPI initialization, the implementation may decide to return MPI_SUCCESS from the MPI initialization function instead of raising an error. It is recommended that an implementation masks an initialization error only when it expects that later MPI calls will result in well-specified behavior (i.e., barring additional failures, either the outcome of any call will be correct, or the call will raise an appropriate error). For example, it may be difficult for an implementation to avoid unspecified behavior when the group of MPI_COMM_WORLD does not contain the same set of MPI processes at all members of the communicator, or if the communicator returned from MPI_COMM_GET_PARENT was not initialized correctly. (*End of advice to implementors.*)

After MPI is initialized, the application can access information about the execution environment by querying the predefined info object MPI_INFO_ENV. The following keys are predefined for this object, corresponding to the arguments of MPI_COMM_SPAWN or of mpiexec:

- ¹⁹ command Name of program executed.
- ²⁰ ₂₁ argv Space separated arguments to command.
- ²² maxprocs Maximum number of MPI processes to start.
- ²⁴ mpi_initial_errhandler Name of the initial errhandler.
- $_{26}^{25}$ soft Allowed values for number of processors.
- ²⁷ host Hostname.

3

4

5

6

7

8

9

10

11

12

13

23

28

33

36

37

38

39 40

41

42

43

- ²⁹ arch Architecture name.
- $_{31}^{30}$ wdir Working directory of the MPI process.
- ³² file Value is the name of a file in which additional information is specified.
- thread_level Requested level of thread support, if requested before the program started execution.

Note that all values are strings. Thus, the maximum number of processes is represented by a string such as "1024" and the requested level is represented by a string such as "MPI_THREAD_SINGLE".

- Advice to users. If one of the argv arguments contains a space, there is no way to tell from the value of the argv info key whether a space is part of the argument or is separating different arguments. (*End of advice to users.*)
- ⁴⁴ The info object MPI_INFO_ENV need not contain a (key,value) pair for each of these ⁴⁵ predefined keys; the set of (key,value) pairs provided is implementation-dependent. Imple-⁴⁶ mentations may provide additional, implementation specific, (key,value) pairs.

⁴⁷ In cases where the MPI processes were started with MPI_COMM_SPAWN_MULTIPLE ⁴⁸ or, equivalently, with a startup mechanism that supports multiple process specifications,

then the values stored in the info object MPI_INFO_ENV at a process are those values that affect the local MPI process.

Example 10.1 If MPI is started with a call to

mpiexec -n 5 -arch sun ocean : -n 10 -arch rs6000 atmos

Then the first 5 processes will have in their MPI_INFO_ENV object the pairs (command, ocean), (maxprocs, 5), and (arch, sun). The next 10 processes will have in MPI_INFO_ENV (command, atmos), (maxprocs, 10), and (arch, rs6000)

Advice to users. The values passed in MPI_INFO_ENV are the values of the arguments passed to the mechanism that started the MPI execution — not the actual value provided. Thus, the value associated with maxprocs is the number of MPI processes requested; it can be larger than the actual number of processes obtained, if the soft option was used. (*End of advice to users.*)

Advice to implementors. High-quality implementations will provide a (key,value) pair for each parameter that can be passed to the command that starts an MPI program. (End of advice to implementors.)

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

Advice to users. In C, the passing of argc and argv is optional, as with MPI_INIT as discussed in Section 10.2.1. In C, null pointers may be passed in their place. (*End of advice to users.*)

This call initializes MPI in the same way that a call to MPI_INIT would. In addition, it initializes the thread environment. The argument required is used to specify the desired level of thread support. The possible values are listed in increasing order of thread support.

Unofficial Draft for Comment Only

 $\frac{4}{5}$

1 **MPI_THREAD_SINGLE** Only one thread will execute. $\mathbf{2}$ **MPI_THREAD_FUNNELED** The process may be multi-threaded, but the application must 3 ensure that only the main thread makes MPI calls (for the definition of main thread, 4 see MPI_IS_THREAD_MAIN on page 425). 56 **MPI_THREAD_SERIALIZED** The process may be multi-threaded, and multiple threads may $\overline{7}$ make MPI calls, but only one at a time: MPI calls are not made concurrently from 8 two distinct threads (all MPI calls are "serialized"). 9 10**MPI_THREAD_MULTIPLE** Multiple threads may call MPI, with no restrictions. 11These values are monotonic; i.e., MPI_THREAD_SINGLE < MPI_THREAD_FUNNELED < 12MPI_THREAD_SERIALIZED < MPI_THREAD_MULTIPLE. 13 Different processes in MPI_COMM_WORLD may require different levels of thread sup-14port. 15The call returns in provided information about the actual level of thread support that 16will be provided by MPI. It can be one of the four values listed above. 17The level(s) of thread support that can be provided by MPI_INIT_THREAD will depend 18 on the implementation, and may depend on information provided by the user before the 19program started to execute (e.g., with arguments to mpiexec). If possible, the call will 20return provided = required. Failing this, the call will return the least supported level such 21that provided > required (thus providing a stronger level of support than required by the 22user). Finally, if the user requirement cannot be satisfied, then the call will return in 23provided the highest supported level. 24 A thread compliant MPI implementation will be able to return provided 25= MPI_THREAD_MULTIPLE. Such an implementation may always return provided 26= MPI_THREAD_MULTIPLE, irrespective of the value of required. 27An MPI library that is not thread compliant must always return provided =28MPI_THREAD_SINGLE, even if MPI_INIT_THREAD is called on a multithreaded process. 29The library should also return correct values for the MPI calls that can be executed before 30 initialization, even if multiple threads have been spawned. 31 32 Rationale. Such code is erroneous, but if the MPI initialization is performed by a 33 library, the error cannot be detected until MPI_INIT_THREAD is called. The require-34 ments in the previous paragraph ensure that the error can be properly detected. (End 35of rationale.) 36 37 A call to MPI_INIT has the same effect as a call to MPI_INIT_THREAD with a required 38= MPI_THREAD_SINGLE. 39 Vendors may provide (implementation dependent) means to specify the level(s) of 40thread support available when the MPI program is started, e.g., with arguments to mpiexec. 41 This will affect the outcome of calls to MPI_INIT and MPI_INIT_THREAD. Suppose, for 42example, that an MPI program has been started so that only MPI_THREAD_MULTIPLE is 43available. Then MPI_INIT_THREAD will return provided = MPI_THREAD_MULTIPLE, irre- 44 spective of the value of required; a call to MPI_INIT will also initialize the MPI thread support 45level to MPI_THREAD_MULTIPLE. Suppose, instead, that an MPI program has been started 46so that all four levels of thread support are available. Then, a call to MPI_INIT_THREAD 47will return provided = required; alternatively, a call to MPI_INIT will initialize the MPI 48thread support level to MPI_THREAD_SINGLE.

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

The design accommodates a static specification of the thread support level, for environments that require static binding of libraries, and for compatibility for current multi-threaded MPI codes. (*End of rationale.*)

Advice to implementors. If provided is not MPI_THREAD_SINGLE then the MPI library should not invoke C or Fortran library calls that are not thread safe, e.g., in an environment where malloc is not thread safe, then malloc should not be used by the MPI library.

Some implementors may want to use different MPI libraries for different levels of thread support. They can do so using dynamic linking and selecting which library will be linked when MPI_INIT_THREAD is invoked. If this is not possible, then optimizations for lower levels of thread support will occur only when the level of thread support required is specified at link time.

Note that required need not be the same value on all processes of MPI_COMM_WORLD. (*End of advice to implementors.*)

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

MPI_QUERY_THREAD(provided)
OUT provided provided level of thread support (integer)
C binding
int MPI_Query_thread(int *provided)
int MF1_Query_thread(int *provided)
Fortran 2008 binding
MPI_Query_thread(provided, ierror)
INTEGER, INTENT(OUT) :: provided
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
MPI_QUERY_THREAD(PROVIDED, IERROR)
INTEGER PROVIDED, IERROR
The call returns in provided the current level of thread support, which will be the value

returned in provided by MPI_INIT_THREAD, if MPI was initialized by a call to MPI_INIT_THREAD(). This function is only applicable when using the World Model to initialize MPI. In the case of applications using both the World Model and the Sessions Model, this function only returns the thread support level returned in provided by MPI_INIT_THREAD.

Unofficial Draft for Comment Only

 $\overline{7}$

 24

```
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
                         MPI libraries are required to provide these calls even if they do not
24
           Rationale.
           support threads, so that portable code that contains invocations to these functions
25
           can link correctly. MPI_INIT continues to be supported so as to provide compatibility
26
           with current MPI codes. (End of rationale.)
27
28
                                It is possible to spawn threads before MPI is initialized, but
           Advice to users.
29
           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
           a multi-threaded process.
33
34
          In the World Model, the level of thread support provided is a global property of the
           MPI process that can be specified only once, when MPI is initialized on that process (or
35
           before). Portable third party libraries have to be written so as to accommodate any
36
           provided level of thread support. Otherwise, their usage will be restricted to specific
37
           level(s) of thread support. If such a library can run only with specific level(s) of thread
38
           support, e.g., only with MPI_THREAD_MULTIPLE, then MPI_QUERY_THREAD can be
39
           used to check whether the user initialized MPI to the correct level of thread support
40
           and, if not, raise an exception. (End of advice to users.)
41
42
43
      10.2.2 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 10.3 and 10.8.

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 10.10.4.

The following examples illustrate these rules.

Example 10.2 The following code is correct

Process 0	Process 1
<pre>MPI_Init(); MPI_Send(dest=1);</pre>	<pre>MPI_Init(); MPI_Recv(src=0);</pre>
<pre>MPI_Finalize();</pre>	<pre>MPI_Finalize();</pre>

Example 10.3 Without a matching receive, the program is erroneous

Process 0	Process 1
<pre>MPI_Init();</pre>	<pre>MPI_Init();</pre>
<pre>MPI_Send (dest=1);</pre>	
<pre>MPI_Finalize();</pre>	<pre>MPI_Finalize();</pre>

Unofficial Draft for Comment Only

Example 10.4 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.

4

```
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 10.5 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 10.6
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.)
```

. . .

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 10.2.4), errors that are not associated with a communicator, window, or file raise the initial error handler (set during the launch operation, see 10.8.4).

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 10.7 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 10.7 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);
```

10.2.3 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
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 10.6.
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 10.6.
42
43
     10.2.4 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.)

10.3 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 10.2.1, 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 10.1, 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 36 cation requirements and available resources, which in turn can be used to create an MPI 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 multi-threaded applications the Sessions Model provides fine-grain control of the thread support level for MPI objects. It is possible to specify different thread support levels when creating different *MPI Session handles*. Thus different components of an application can use different thread support levels.

The Sessions Model introduces a concept of isolation. MPI objects derived from different MPI Session handles, or derived from MPI_COMM_WORLD and MPI_COMM_SELF, shall not be intermixed with each other in a single MPI function call. Additionally,

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

1920

21

22

23 24

25

26

27

2829

30

 31

32

38 39

40 41

42

4344

4546

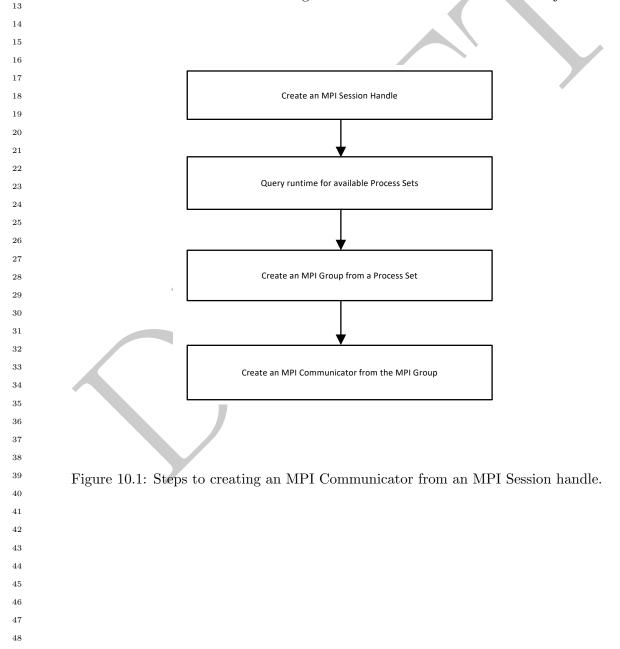
47

two communicator handles compared using MPI_COMM_COMPARE shall be derived from the same *MPI Session Handle*,
for MPI_COMM_CREATE_GROUP, the input communicator and group handles shall be derived from the same *MPI Session handle*,

 $\mathbf{2}$

• all *request handles* supplied to a multi-request completion function, for example MPI_TESTALL, shall represent non-blocking or persistent operations associated with MPI objects derived from the same *MPI Session handle*.

This restriction does not apply to generalized requests (Section 12.2) 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.



MPI_SESSION_INIT(info, errhandler, session)

IN	info	info object to specify thread support level and MPI implementation specific resources (handle)
IN	errhandler	error handler to invoke in the event that an error is encountered during this function call (handle)
OUT	session	new session (handle)

C binding

```
int MPI_Session_init(MPI_Info info, MPI_Errhandler errhandler,
             MPI_Session *session)
```

Fortran 2008 binding

MPI_Session_init(info, errhandler, session, ierror)	
TYPE(MPI_Info), INTENT(IN) :: info	
TYPE(MPI_Errhandler), INTENT(IN) :: errhandler	
TYPE(MPI_Session), INTENT(OUT) :: session	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	

Fortran binding

MPI_	_SESSION_	INIT(INFO,	ERRHANI	DLER,	SESS	SION,	IERR	OR)
	INTEGER	INFO,	ERRH	ANDLER,	SESS1	CON,	IERRO)R	

The info argument is used to request MPI functionality requirements and possible MPI implementation specific capabilities. The following info key is predefined:

mpi_thread_support_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 33 level actually provided by the MPI implementation can be determined via a subse-34 quent call to MPI_SESSION_GET_INFO to return the info object associated with the Session. The default thread support level is MPI implementation dependent. 36

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 8.3) 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.)

2

5 6

13

14

15

23 24 25

26

2728

29

30

31

32

35

37

38

39

40

41

42

43 44

45

46

1 Advice to implementors. Owing to the restrictions of the "MPI_THREAD_SINGLE" $\mathbf{2}$ 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 10.10.4. 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 4210.3.2 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 mpi://SELF. Additional 47process set names may be defined, for example, mpix://UNIVERSE and hwloc://L3Cache may

Unofficial Draft for Comment Only

be defined by the MPI implementation. The mpi:// namespace is reserved for exclusive use



 $\mathbf{2}$

job://12942 mpi://WORLD location://rack/17 location://rack/23 app://atmos app://ocean mpi://SELF mpi://SELF mpi://SELF mpi://SELF mpi://SELF MPI process 1 MPI process 2 MPI process 3 MPI process 4 MPI process 0

Figure 10.2: 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.

by the MPI standard. Figure 10.2 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.

10.3.3 Runtime Query Functions

MPI_SESSION_GET_NUM_PSETS(session, info, npset_names)								
IN	session	session (handle)						
IN	info	info object (handle)						
OUT	npset_names	number of available process sets (non-negative						

integer)

C binding

Fortran 2008 binding MPI_Session_get_num_psets(session, info, npset_names, ierror)

1 2 3 4	TYPE(INTEG	MPI_Session), INTENT(IN) MPI_Info), INTENT(IN) :: ER, INTENT(OUT) :: npset ER, OPTIONAL, INTENT(OUT	info _names					
5 6 7 8		8	, INFO, NPSET_NAMES, IERROR) NAMES, IERROR					
9 10 11 12 13 14 15 16 17 18 19 20 21	which the the number process set the index of particular set become in the com- error. For is a local of communica	calling MPI process is a mem r of available process sets dur is become available. However of a particular process set na index, or to delete a process is invalid, for example, when imunication system, subseque example, creating an MPI_Gro operation, but creating an MP ation should raise an error.	runtime for the number of available process sets in ber. An MPI implementation is allowed to increase ring the execution of an MPI application when new r, MPI implementations are not allowed to change me, or to change the name of the process set at a set name once it has been added. When a process some processes become unreachable due to failures ent usage of the process set name should raise an up from such a process set might succeed because it PI_Comm from that group and attempting collective					
22 23 24 25 26 27 28 29	Advice to implementors. It is anticipated that an MPI implementation may be re- lying on an external runtime system to provide process sets. Such runtime systems may have the ability to dynamically create process sets during the course of appli- cation execution. Requiring the number of process sets returned by MPI_SESSION_GET_NUM_PSETS to be constant over the course of application exe- cution would prevent an application from taking advantage of such capabilities. (<i>End</i> <i>of advice to implementors.</i>)							
30 31	MPI_SESS	ION_GET_NTH_PSET(sessior	n, info, n, pset_len, pset_name)					
32	IN	session	session (handle)					
33	IN	info	info object (handle)					
34	IN	n	index of the desired process set name (integer)					
35 36	INOUT	 pset_len	length of the pset_name argument (integer)					
37								
38	OUT	pset_name	name of the nth process set (string)					
39 40 41 42	C binding int MPI_S	-	Session session, MPI_Info info, int n, *pset_name)					
42 43	Fortran 2	008 binding						
44		0	info, n, pset_len, pset_name, ierror)					
45		MPI_Session), INTENT(IN)						
46		<pre>MPI_Info), INTENT(IN) ::</pre>	info					
47 48		ER, INTENT(IN) :: n ER, INTENT(INOUT) :: pse	t lon					
		, INTENT(INCOT) pse	0_1 <u>0</u> 1					

CHARACTER(LEN=*), INTENT(OUT) :: pset_name INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

```
MPI_SESSION_GET_NTH_PSET(SESSION, INFO, N, PSET_LEN, PSET_NAME, IERROR)
INTEGER SESSION, INFO, N, PSET_LEN, IERROR
CHARACTER*(*) PSET_NAME
```

This function returns the name of the nth process set in the supplied pset_name buffer. 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, then the string value returned in pset_name is truncated. If pset_len is set to 0, pset_name is not changed. On return, the value of pset_len will be set to the required buffer size to hold the process set name. In C, pset_len includes the required space for the null terminator. In C, this function returns a null terminated string in all cases where the pset_len input value is greater than 0.

If two MPI processes get the same process set name, then the intersection of the two process sets shall either be the empty set or identical to the union of the two process sets.

After a successful call to MPI_SESSION_GET_NTH_PSET, subsequent calls to routines that query information about the same process set name and same session handle must return the same information. An MPI implementation is not allowed to alter any of the returned process set names.

Process set names have an implementation-defined maximum length of MPI_MAX_PSET_NAME_LEN.

Advice to users. MPI_MAX_PSET_NAME_LEN might be very large, so it might not be wise to declare a string of that size. Users are encouraged to use MPI_SESSION_GET_NTH_PSET both for obtaining the length of a pset_name and the process set name. (*End of advice to users.*)

MPI_SESSION_GET_INFO(session, info_used)							
WF1_3E33ION_GE1_INFO(session, Into_useu)							
IN session	session (handle)	33					
OUT info_used	see explanation below (handle)	34					
_		35					
C binding		36					
5	Session session, MPI_Info *info_used)	37					
int in i_bession_get_into(in i_t	Jession Session, in 1_1110 #1110_used/	38					
Fortran 2008 binding		39					
MPI_Session_get_info(session,	info_used, ierror)	40					
TYPE(MPI_Session), INTENT	(IN) :: session	41					
TYPE(MPI_Info), INTENT(OUT	[) :: info_used	42					
INTEGER, OPTIONAL, INTENT	(OUT) :: ierror	43					
Fortron hinding		44					
Fortran binding		45					
MPI_SESSION_GET_INFO(SESSION,	-	46					
INTEGER SESSION, INFO_USE	J, IERRUR	47					
		48					

1

2

5

6

7 8

9

10

11

12

13

14

15

16

17

18

19

20

21

22 23

 24

25

26

27

438 CHAPTER 10. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

1			returns a new info object containing the hints of the MPI									
2	Session associated with session. The current setting of all hints related to this MPI Session is returned in info_used. An MPI implementation is required to return all hints that are											
3 4			· ·									
5	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											
6	the implementation. If no such hints exist, a handle to a newly created info object is											
7	-		alue pair. The user is responsible for freeing info_used via									
8	MPI_INF		and pair. The user is responsible for meeting into_used via									
9		•										
10												
11	MPI_SES	SION_GET_PSET_INF	O(session, pset_name, info)									
12	IN	session	session (handle)									
13 14	IN	pset_name	name of process set (string)									
15 16	OUT	info	info object containing information about the given process set (handle)									
17												
18	C bindi	•										
19	int MPL_		nfo(MPI_Session session, const char *pset_name,									
20 21		MPI_Info *inf	6)									
21	Fortran	2008 binding										
23			session, pset_name, info, ierror)									
24		E(MPI_Session), INTH										
25		RACTER(LEN=*), INTEN	-									
26		E(MPI_Info), INTENT										
27	INTE	EGER, OPTIONAL, INTE	CNT(UUT) :: lerror									
28	Fortran	binding										
29	MPI_SESS	SION_GET_PSET_INFO(S	SESSION, PSET_NAME, INFO, IERROR)									
30	INTE	EGER SESSION, INFO,	IERROR									
31	CHAF	RACTER*(*) PSET_NAME	5									
32	This	function is used to au	ery properties of a specific process set. The returned <i>info</i>									
33			ting MPI info object query functions. One key/value pair									
34		-	alue of the mpi_size key specifies the number of MPI processes									
35 36			esponsible for freeing the returned MPI_Info object.									
37	_											
38	10.3.4	Sessions Model Examp	les									
39	This sect	ion presents several ex	amples of how to use MPI Sessions to create MPI Groups									
40		Communicators.										
41		Communications.										
42	Example		le illustrating creation of an MPI communicator using the									
43	Sessions 2	Model.										
44												
45												
46												
47 48												
-10												

```
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;
   int ret = 0;
                                                                                      12
   const char pset_name[] = "mpi://WORLD";
                                                                                      13
                                                                                      14
   const char mt_key[] = "mpi_thread_support_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
   MPI_Info_get(tinfo, mt_key, sizeof(out_value),
                                                                                      40
                 out_value, &flag);
                                                                                      41
   if (flag != 1) {
                                                                                      42
      printf("Could not find key %s\n", mt_key);
      ret = -1;
                                                                                      43
                                                                                      44
      goto fn_exit;
   }
                                                                                      45
                                                                                      46
                                                                                      47
   if (strcmp(out_value, mt_value)) {
                                                                                      48
      printf("Did not get thread multiple support, got %s\n",
```

```
1
                   out_value);
\mathbf{2}
            ret = -1;
3
            goto fn_exit;
4
        }
5
6
        /*
7
         * create a group from the WORLD process set
8
         */
9
        rc = MPI_Group_from_session_pset(lib_shandle,
10
                                             pset_name,
11
                                             &wgroup);
12
        if (rc != MPI_SUCCESS) {
13
            ret = -1;
14
            goto fn_exit;
15
        }
16
17
        /*
18
         * get a communicator
19
         */
20
        rc = MPI_Comm_create_from_group(wgroup,
21
                                             "org.mpi-forum.mpi-v4_0.example-ex10_8",
22
                                            MPI_INFO_NULL,
23
                                            MPI_ERRORS_RETURN,
24
                                            &lib_comm);
25
        if (rc != MPI_SUCCESS) {
26
            ret = -1;
27
            goto fn_exit;
        }
28
29
30
        /*
31
         * free group, library doesn't need it.
32
         */
33
34
     fn_exit:
35
        MPI_Group_free(&wgroup);
36
37
        if (sinfo != MPI_INFO_NULL) {
38
            MPI_Info_free(&sinfo);
39
        }
40
41
        if (tinfo != MPI_INFO_NULL) {
42
            MPI_Info_free(&tinfo);
        }
43
44
45
        if (ret != 0) {
46
            MPI_Session_finalize(&lib_shandle);
47
        }
48
```

```
return ret;
```

}

Example 10.8 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 10.9 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;
   char *pset_name = NULL;
                                                                                  21
   char *info_val = NULL;
   MPI_Session shandle = MPI_SESSION_NULL;
                                                                                   22
                                                                                   23
   MPI_Info sinfo = MPI_INFO_NULL;
   MPI_Group pgroup = MPI_GROUP_NULL;
   if (argc < 2) {
      fprintf(stderr, "A process set name fragment is required\n");
                                                                                   27
                                                                                   28
      return -1;
                                                                                   29
   }
                                                                                   30
   rc = MPI_Session_init(MPI_INFO_NULL, MPI_ERRORS_RETURN, &shandle);
                                                                                   32
   if (rc != MPI_SUCCESS) {
                                                                                   33
      fprintf(stderr, "Could not initialize session, bailing out\n");
                                                                                  34
      return -1;
   }
                                                                                  35
                                                                                   36
                                                                                  37
   MPI_Session_get_num_psets(shandle, MPI_INFO_NULL, &n_psets);
                                                                                   38
   for (i=0, pset_name=NULL; i<n_psets; i++) {</pre>
       psetlen = 0;
       MPI_Session_get_nth_pset(shandle, MPI_INFO_NULL, i,
                                                                                  42
                                 &psetlen, NULL);
       pset_name = (char *)malloc(sizeof(char) * psetlen);
       MPI_Session_get_nth_pset(shandle, MPI_INFO_NULL, i,
                                 &psetlen, pset_name);
       if (strstr(pset_name, argv[1]) != NULL) break;
       free(pset_name);
```

1

 $\mathbf{2}$

3

4

5

6 7

8

9 10

11

1213

1415

16

17

18

19

20

 24

2526

31

39

40

41

43

4445

46

```
1
             pset_name = NULL;
\mathbf{2}
        }
3
4
         /*
5
          * get instance of an info object for this Session
6
          */
7
8
        MPI_Session_get_pset_info(shandle, pset_name, &sinfo);
9
        MPI_Info_get_valuelen(sinfo, "mpi_size", &valuelen, &flag);
10
         info_val = (char *)malloc(valuelen+1);
11
        MPI_Info_get(sinfo, "mpi_size", valuelen, info_val, &flag);
12
        free(info_val);
13
14
         /*
15
          * create a group from the process set
16
          */
17
18
        rc = MPI_Group_from_session_pset(shandle, pset_name,
19
                                              &pgroup);
20
        ret = (rc == MPI_SUCCESS) ? 0 : -1;
21
22
        free(pset_name);
23
        MPI_Group_free(&pgroup);
24
        MPI_Info_free(&sinfo);
25
        MPI_Session_finalize(&shandle);
26
27
        fprintf(stderr, "Test completed ret = %d\n", ret);
28
        return ret;
29
30
     }
^{31}
         Example 10.9 illustrates several aspects of the Sessions Model. First, the default error
32
     handler can be specified when instantiating a Session instance. Second, there must be at
33
     least two process sets associated with a Session. Third, the example illustrates use of the
34
     Sessions info object and the one required key: mpi_size.
35
36
     Example 10.10 A Fortran 2008 example illustrating how to obtain information about
37
     available process sets, create an MPI Group from a process set, and subsequently create an
38
     MPI Communicator.
39
40
     PROGRAM MAIN
41
          USE mpi_f08
42
          IMPLICIT NONE
43
          INTEGER :: pset_len, ierror, n_psets
44
          CHARACTER(LEN=:), ALLOCATABLE :: pset_name
45
          TYPE(MPI_Session) :: shandle
46
          TYPE(MPI_Group) :: pgroup
47
          TYPE(MPI_Comm) :: pcomm
```

```
48
```

```
1
    CALL MPI_Session_init(MPI_INFO_NULL, MPI_ERRORS_RETURN, &
                                                                                      \mathbf{2}
                           shandle, ierror)
                                                                                      3
    IF (ierror .NE. MPI_SUCCESS) THEN
       WRITE(*,*) "MPI_Session_init failed"
                                                                                      4
       ERROR STOP
                                                                                      5
    END IF
                                                                                      6
    CALL MPI_Session_get_num_psets(shandle, MPI_INFO_NULL, n_psets)
    IF (n_psets .LT. 2) THEN
                                                                                      9
                                                                                      10
       WRITE(*,*) "MPI_Session_get_num_psets didn't return at least 2 psets"
                                                                                      11
       ERROR STOP
    END IF
                                                                                      12
                                                                                      13
                                                                                      14
!
                                                                                      15
!
    Just get the second pset's length and name
                                                                                      16
!
    Note that index values are zero-based, even in Fortran
                                                                                      17
ŗ
                                                                                      18
                                                                                      19
    pset_len = 0
    CALL MPI_Session_get_nth_pset(shandle, MPI_INFO_NULL, 1,
                                                                      &
                                                                                      20
                                                                                      21
                                    pset_len, pset_name)
    ALLOCATE(CHARACTER(LEN=pset_len)::pset_name)
                                                                                      22
                                                                                      23
    CALL MPI_Session_get_nth_pset(shandle, MPI_INFO_NULL, 1,
                                                                      &
                                                                                      ^{24}
                                     pset_len, pset_name)
                                                                                      25
                                                                                      26
!
!
    create a group from the pset
                                                                                      27
                                                                                      28
!
                                                                                      29
    CALL MPI_Group_from_session_pset(shandle, pset_name, pgroup)
                                                                                      30
ŗ
                                                                                      31
!
    free the buffer used for the pset name
!
                                                                                      32
                                                                                      33
    DEALLOCATE(pset_name)
                                                                                      34
                                                                                      35
1
    create a MPI communicator from the group
                                                                                      36
!
                                                                                      37
!
    CALL MPI_Comm_create_from_group(pgroup, "session_example",
                                                                                      38
                                                                      &
                                                                                      39
                                               MPI_INFO_NULL,
                                                                       &
                                               MPI_ERRORS_RETURN,
                                                                                      40
                                                                       &
                                                                                      41
                                               pcomm)
                                                                                      42
    CALL MPI_Barrier(pcomm, ierror)
                                                                                      43
                                                                                      44
    IF (ierror .NE. MPI_SUCCESS) THEN
        WRITE(*,*) "Barrier call on communicator failed"
                                                                                      45
                                                                                      46
        ERROR STOP
                                                                                      47
    END IF
                                                                                      48
```

444 CHAPTER 10. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

1 2 3	CALL MPI_Comm_free(pcomm) CALL MPI_Group_free(pgroup) CALL MPI_Session_finalize(shandle, ierror)
4 5 6	END PROGRAM MAIN
7 8 9 10 11	Note in this example that the call to MPI_SESSION_FINALIZE may block in order to ensure that the calling MPI process has completed its involvement in the preceding MPI_BARRIER operation. If MPI_COMM_DISCONNECT had been used instead of MPI_COMM_FREE, the example would have blocked in MPI_COMM_DISCONNECT rather than MPI_SESSION_FINALIZE.
13 14	10.4 Common Elements of Both Process Models
15 16	10.4.1 MPI Functionality that is Always Available
17 18 19 20 21 22 23 24	Some MPI functions may be invoked at any time, including prior to calling MPI_INIT or MPI_SESSION_INIT, and following MPI finalization, independent of whether the World Model, Sessions Model, or both are used. These functions can be called concurrently by multiple threads within an MPI Process. Table 10.1 lists the applicable MPI functions. In addition to the functions listed in Table 10.1, any function with the prefix MPI_T_ (within the constraints for functions with this prefix listed in Section 14.3.4) may also be called prior to MPI initialization and after MPI finalization.
25 26 27 28	10.4.2 Aborting MPI Processes
29	MPI_ABORT(comm, errorcode)
30 31	IN comm communicator of tasks to abort
32	IN errorcode error code to return to invoking environment
33 34 35	C binding int MPI_Abort(MPI_Comm comm, int errorcode)
36 37 38 39 40	<pre>Fortran 2008 binding MPI_Abort(comm, errorcode, ierror) TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, INTENT(IN) :: errorcode INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>
41 42 43 44	Fortran binding MPI_ABORT(COMM, ERRORCODE, IERROR) INTEGER COMM, ERRORCODE, IERROR
45 46 47	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

⁴⁸ from the main program.

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 10.1: List of MPI Functions that can be called at any time within an MPI program, including prior to MPI initialization and following MPI finalization

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 im-plementation should be able to provide error handling for communicators, windows, and files involving both aborted and non-aborted processes. As an example, if the user changes the error handler for MPI_COMM_WORLD to MPI_ERRORS_RETURN or a custom error handler, when a subset of MPI_COMM_WORLD is aborted, the remaining processes in MPI_COMM_WORLD should be able to continue communicating with each other and receive 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.)

Unofficial Draft for Comment Only

 31

Advice to users. Whether the errorcode is returned from the executable or from the MPI process startup mechanism (e.g., mpiexec), is an aspect of quality of the MPI library but not mandatory. (End of advice to users.)

Advice to implementors. Where possible, a high-quality implementation will try to return the errorcode from the MPI process startup mechanism (e.g. mpiexec or singleton init). (End of advice to implementors.)

10.5Portable MPI Process Startup

1

2

3

4 5

6

7

8 9 10

11 12

13

14

15

18

32 33

34

37

39

40

41

42

43

44 45 A number of implementations of MPI provide a startup command for MPI programs that is of the form

mpirun <mpirun arguments> <program> <program arguments>

16Separating the command to start the program from the program itself provides flexibility, 17particularly 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. 19

Having a standard startup mechanism also extends the portability of MPI programs one 20step further, to the command lines and scripts that manage them. For example, a validation 21suite script that runs hundreds of programs can be a portable script if it is written using such 22 a standard startup mechanism. In order that the "standard" command not be confused 23with existing practice, which is not standard and not portable among implementations, 24instead of mpirun MPI specifies mpiexec. 25

While a standardized startup mechanism improves the usability of MPI, the range of 26environments is so diverse (e.g., there may not even be a command line interface) that MPI 27cannot mandate such a mechanism. Instead, MPI specifies an mpiexec startup command 28and recommends but does not require it, as advice to implementors. However, if an im-29 plementation does provide a command called mpiexec, it must be of the form described 30 below. 31

It is suggested that

mpiexec -n <numprocs> <program>

be at least one way to start <program> with an initial set of <numprocs> processes, which 35 will be accessible as the process set named "mpi://world" in the Sessions Model and/or 36 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. 38

Advice to implementors. Implementors, if they do provide a special startup command for MPI programs, are advised to give it the following form. The syntax is chosen in order that mpiexec be able to be viewed as a command-line version of MPI_COMM_SPAWN (See Section 10.8.4).

Analogous to MPI_COMM_SPAWN, we have

46	mpiexec -n	<maxp:< th=""><th>rocs></th></maxp:<>	rocs>
47	-soft	<	>
48	-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 10.8.4 for the meanings of these arguments. For the case corresponding to MPI_COMM_SPAWN_MULTIPLE there are two possible formats:

>

>

> >

>

Form A:

<pre>mpiexec { <ab< pre=""></ab<></pre>	pove arguments>	}	:	{	• • •	}	: {		.}	:		{	• • •	}
---	-----------------	---	---	---	-------	---	-----	--	----	---	--	---	-------	---

As with MPI_COMM_SPAWN, all the arguments are optional. (Even the $-n \times$ argument is optional; the default is implementation dependent. It might be 1, it might be taken from an environment variable, or it might be specified at compile time.) The names and meanings of the arguments are taken from the keys in the info argument to MPI_COMM_SPAWN. There may be other, implementation-dependent arguments as well.

Note that Form A, though convenient to type, prevents colons from being program arguments. Therefore an alternate, file-based form is allowed:

Form B:

mpiexec -configfile <filename>

where the lines of <filename> are of the form separated by the colons in Form A. Lines beginning with '#' are comments, and lines may be continued by terminating the partial line with '\'.

Example 10.11 Start 16 instances of myprog on the current or default machine:

mpiexec -n 16 myprog

Example 10.12 Start 10 processes on the machine called ferrari:

mpiexec -n 10 -host ferrari myprog

Example 10.13 Start three copies of the same program with different command-line arguments:

mpiexec myprog infile1 : myprog infile2 : myprog infile3

 $\mathbf{2}$

 $45 \\ 46$

1 **Example 10.14** Start the ocean program on five Suns and the atmos program on 2 10 RS/6000's: 3 4 mpiexec -n 5 -arch sun ocean : -n 10 -arch rs6000 atmos 56 It is assumed that the implementation in this case has a method for choosing hosts of 7 the appropriate type. Their ranks are in the order specified. 8 9 **Example 10.15** Start the ocean program on five Suns and the atmos program on 10 10 RS/6000's (Form B): 11 12mpiexec -configfile myfile 13 14where myfile contains 1516-n 5 -arch sun ocean 17 -n 10 -arch rs6000 atmos 18 19 (End of advice to implementors.) 2021

10.6 MPI and Threads

22

37

38

23This section specifies the interaction between MPI calls and threads. Although thread com-24pliance is not required, the standard specifies how threads are to work if they are provided. 25The section lists minimal requirements for thread compliant MPI implementations and 26defines functions that can be used for initializing the thread environment. MPI may be im-27plemented in environments where threads are not supported or perform poorly. Therefore, 28MPI implementations are not required to be thread compliant as defined in this section. 29 Regardless of whether or not the MPI implementation is thread compliant, a subset of MPI 30 functions must always be thread safe. A complete list of such MPI functions is given in Ta- 31 ble 10.1. When a thread is executing one of these routines, if another concurrently running 32 thread also makes an MPI call, the outcome will be as if the calls executed in some order. 33

This section generally assumes a thread package similar to POSIX threads [41], but the syntax and semantics of thread calls are not specified here — these are beyond the scope of this document.

10.6.1 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.

Rationale. This model corresponds to the POSIX model of interprocess communi cation: the fact that a process is multi-threaded, rather than single-threaded, does
 not affect the external interface of this process. MPI implementations in which MPI
 'processes' are POSIX threads inside a single POSIX process are not thread-compliant
 by this definition (indeed, their "processes" are single-threaded). (End of rationale.)

Advice to users. It is the user's responsibility to prevent races when threads within the same application post conflicting communication calls. The user can make sure that two threads in the same process will not issue conflicting communication calls by using distinct communicators at each thread. (End of advice to users.)

The two main requirements for a thread-compliant implementation are listed below.

- 1. All MPI calls are *thread-safe*, i.e., two concurrently running threads may make MPI calls and the outcome will be as if the calls executed in some order, even if their execution is interleaved.
- 2. Blocking MPI calls will block the calling thread only, allowing another thread to execute, if available. The calling thread will be blocked until the event on which it is waiting occurs. Once the blocked communication is enabled and can proceed, then the call will complete and the thread will be marked runnable, within a finite time. A blocked thread will not prevent progress of other runnable threads on the same process, and will not prevent them from executing MPI calls.

Example 10.16 Process 0 consists of two threads. The first thread executes a blocking 19send call MPI_Send(buff1, count, type, 0, 0, comm), whereas the second thread executes 20a blocking receive call MPI_Recv(buff2, count, type, 0, 0, comm, &status), i.e., the first 21thread sends a message that is received by the second thread. This communication should 22always succeed. According to the first requirement, the execution will correspond to some 23interleaving of the two calls. According to the second requirement, a call can only block the calling thread and cannot prevent progress of the other thread. If the send call went ahead of the receive call, then the sending thread may block, but this will not prevent the receiving thread from executing. Thus, the receive call will occur. Once both calls occur, the communication is enabled and both calls will complete. On the other hand, a single-threaded process that posts a send, followed by a matching receive, may deadlock. The progress requirement for multithreaded implementations is stronger, as a blocked call 30 cannot prevent progress in other threads.

Advice to implementors. MPI calls can be made thread-safe by executing only one at a time, e.g., by protecting MPI code with one process-global lock. However, blocked operations cannot hold the lock, as this would prevent progress of other threads in the process. The lock is held only for the duration of an atomic, locally-completing suboperation such as posting a send or completing a send, and is released in between. Finer locks can provide more concurrency, at the expense of higher locking overheads. Concurrency can also be achieved by having some of the MPI protocol executed by separate server threads. (End of advice to implementors.)

10.6.2 Clarifications

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

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

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

 24

25

26

27

28

29

 31 32

33

34

35

36

37

38

39

40 41

4243

44

45

46

47

1 Threads and the Sessions Model The Sessions Model provides a finer-grain approach to $\mathbf{2}$ controlling the interaction between MPI calls and threads. When using this model, the 3 desired level of thread support is specified at Session initialization time. See Section 10.3. 4 Thus it is possible for communicators and other MPI objects derived from one Session $\mathbf{5}$ to provide a different level of thread support than those created from another Session 6 for which a different level of thread support was requested. Depending on the level of 7thread support requested at Session initialization time, different threads in a MPI process 8 can make concurrent calls to MPI when using MPI objects derived from different session 9 handles. Note that the requested and provided level of thread support when creating a 10 Session may influence the granted level of thread support in a subsequent invocation of 11MPI_SESSION_INIT. Likewise, if the application at some point calls

¹² MPI_INIT_THREAD, the requested and granted level of thread support may influence the ¹³ granted level of thread support for subsequent calls to MPI_SESSION_INIT. Similarly, if the ¹⁴ application calls MPI_INIT_THREAD after a call to MPI_SESSION_INIT, the level of thread ¹⁵ support returned from MPI_INIT_THREAD may be similarly influenced by the requested ¹⁶ 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.

22

29

30

31

32

33

34

35

²³ Multiple threads completing the same request. A program in which two threads block, wait ²⁴ ing on the same request, is erroneous. Similarly, the same request cannot appear in the
 ²⁵ array of requests of two concurrent MPI_{WAIT|TEST}{ANY|SOME|ALL} calls. In MPI, a
 ²⁶ request can only be completed once. Any combination of wait or test that violates this rule
 ²⁷ 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.*)
- 36 37

³⁸ Probe A receive call that uses source and tag values returned by a preceding call to ³⁹ MPI_PROBE or MPI_IPROBE will receive the message matched by the probe call only ⁴⁰ if there was no other matching receive after the probe and before that receive. In a multi-⁴¹ threaded environment, it is up to the user to enforce this condition using suitable mutual ⁴² exclusion logic. This can be enforced by making sure that each communicator is used by ⁴³ only one thread on each process. Alternatively, MPI_MPROBE or MPI_IMPROBE can be ⁴⁴ used.

45

⁴⁶ Collective calls Matching of collective calls on a communicator, window, or file handle is
 ⁴⁷ done according to the order in which the calls are issued at each process. If concurrent

threads issue such calls on the same communicator, window or file handle, it is up to the user to make sure the calls are correctly ordered, using interthread synchronization.

Advice to users. With three concurrent threads in each MPI process of a communicator comm, it is allowed that thread A in each MPI process calls a collective operation on comm, thread B calls a file operation on an existing filehandle that was formerly opened on comm, and thread C invokes one-sided operations on an existing window handle that was also formerly created on comm. (*End of advice to users.*)

Rationale. As specified in MPI_FILE_OPEN and MPI_WIN_CREATE, a file handle and a window handle inherit only the group of processes of the underlying communicator, but not the communicator itself. Accesses to communicators, window handles and file handles cannot affect one another. (*End of rationale.*)

Advice to implementors. If the implementation of file or window operations internally uses MPI communication then a duplicated communicator may be cached on the file or window object. (*End of advice to implementors.*)

Exception handlers An exception handler does not necessarily execute in the context of the thread that made the exception-raising MPI call; the exception handler may be executed by a thread that is distinct from the thread that will return the error code.

Rationale. The MPI implementation may be multithreaded, so that part of the communication protocol may execute on a thread that is distinct from the thread that made the MPI call. The design allows the exception handler to be executed on the thread where the exception occurred. (*End of rationale.*)

Interaction with signals and cancellations The outcome is undefined if a thread that executes an MPI call is cancelled (by another thread), or if a thread catches a signal while executing an MPI call. However, a thread of an MPI process may terminate, and may catch signals or be cancelled by another thread when not executing MPI calls.

Rationale. Few C library functions are signal safe, and many have cancellation points — points at which the thread executing them may be cancelled. The above restriction simplifies implementation (no need for the MPI library to be "async-cancel-safe" or "async-signal-safe"). (End of rationale.)

Advice to users. Users can catch signals in separate, non-MPI threads (e.g., by masking signals on MPI calling threads, and unmasking them in one or more non-MPI threads). A good programming practice is to have a distinct thread blocked in a call to sigwait for each user expected signal that may occur. Users must not catch signals used by the MPI implementation; as each MPI implementation is required to document the signals used internally, users can avoid these signals. (*End of advice to users.*)

Advice to implementors. The MPI library should not invoke library calls that are not thread safe, if multiple threads execute. (*End of advice to implementors.*)

$47 \\ 48$

Unofficial Draft for Comment Only

 $\mathbf{5}$

 $\overline{7}$

 24

 $44 \\ 45$

¹ 10.7 The Dynamic Process Model

The dynamic process model allows for the creation and cooperative termination of processes after an MPI application has started. It provides a mechanism to establish communication between the newly created processes and the existing MPI application. It also provides a mechanism to establish communication between two existing MPI applications, even when one did not "start" the other.

8 9

13

14

15

10.7.1 Starting Processes

 $^{10}_{11}$ MPI applications may start new processes through an interface to an external process man- $^{12}_{12}$ ager.

MPI_COMM_SPAWN starts MPI processes and establishes communication with them, returning an intercommunicator. MPI_COMM_SPAWN_MULTIPLE starts several different binaries (or the same binary with different arguments), placing them in the same

MPI_COMM_WORLD and returning an intercommunicator.

¹⁶ MPI uses the group abstraction to represent processes. A process is identified by a ¹⁷ (group, rank) pair.

¹⁹ ₂₀ 10.7.2 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. Examples of such environments are:

2526

27

28 29

30

31

32

33

34

35

36

37 38

39

40

41

- MPP managed by a batch queueing system. Batch queueing systems generally allocate resources before an application begins, enforce limits on resource use (CPU time, memory use, etc.), and do not allow a change in resource allocation after a job begins. Moreover, many MPPs have special limitations or extensions, such as a limit on the number of processes that may run on one processor, or the ability to gang-schedule processes of a parallel application.
 - Network of workstations with PVM. PVM (Parallel Virtual Machine) allows a user to create a "virtual machine" out of a network of workstations. An application may extend the virtual machine or manage processes (create, kill, redirect output, etc.) through the PVM library. Requests to manage the machine or processes may be intercepted and handled by an external resource manager.
 - Network of workstations managed by a load balancing system. A load balancing system may choose the location of spawned processes based on dynamic quantities, such as load average. It may transparently migrate processes from one machine to another when a resource becomes unavailable.
- Large SMP with Unix. Applications are run directly by the user. They are scheduled at a low level by the operating system. Processes may have special scheduling characteristics (gang-scheduling, processor affinity, deadline scheduling, processor locking, etc.) and be subject to OS resource limits (number of processes, amount of memory, etc.).

MPI assumes, implicitly, the existence of an environment in which an application runs. It does not provide "operating system" services, such as a general ability to query what processes are running, to kill arbitrary processes, to find out properties of the runtime environment (how many processors, how much memory, etc.).

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

At some low level, obviously, MPI must be able to interact with the runtime system, but the interaction is not visible at the application level and the details of the interaction are not specified by the MPI standard.

In many cases, it is impossible to keep environment-specific information out of the MPI interface without seriously compromising MPI functionality. To permit applications to take advantage of environment-specific functionality, many MPI routines take an info argument that allows an application to specify environment-specific information. There is a tradeoff between functionality and portability: applications that make use of environment-specific info are not portable.

MPI does not require the existence of an underlying "virtual machine" model, in which there is a consistent global view of an MPI application and an implicit "operating system" managing resources and processes. For instance, processes spawned by one task may not be visible to another; additional hosts added to the runtime environment by one process may not be visible in another process; tasks spawned by different processes may not be automatically distributed over available resources.

Interaction between MPI and the runtime environment is limited to the following areas:

- A process may start new processes with MPI_COMM_SPAWN and MPI_COMM_SPAWN_MULTIPLE.
- When a process spawns a child process, it may optionally use an info argument to tell the runtime environment where or how to start the process. This extra information may be opaque to MPI.
- An attribute MPI_UNIVERSE_SIZE (See Section 10.10.1) 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.

10.8 Process Manager Interface

10.8.1 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.

 $\mathbf{2}$

 $\mathbf{5}$

```
1
      10.8.2 Starting Processes and Establishing Communication
\mathbf{2}
      The following routine starts a number of MPI processes and establishes communication with
3
      them, returning an intercommunicator.
4
5
           Advice to users.
                               It is possible in MPI to start a static SPMD or MPMD appli-
6
           cation by first starting one process and having that process start its siblings with
7
           MPI_COMM_SPAWN. This practice is discouraged primarily for reasons of perfor-
8
           mance. If possible, it is preferable to start all processes at once, as a single MPI
9
           application. (End of advice to users.)
10
11
12
      MPI_COMM_SPAWN(command, argv, maxprocs, info, root, comm, intercomm,
13
14
                     array_of_errcodes)
15
        IN
                  command
                                               name of program to be spawned (string, significant
16
                                               only at root)
17
                                               arguments to command (array of strings, significant
        IN
                  argv
18
                                               only at root)
19
20
                                               maximum number of processes to start (integer,
        IN
                  maxprocs
21
                                               significant only at root)
22
        IN
                  info
                                               a set of key-value pairs telling the runtime system
23
                                               where and how to start the processes (handle,
^{24}
                                               significant only at root)
25
        IN
                                               rank of process in which previous arguments are
                  root
26
                                               examined (integer)
27
28
        IN
                                               intracommunicator containing group of spawning
                  comm
29
                                               processes (handle)
30
        OUT
                  intercomm
                                               intercommunicator between original group and the
^{31}
                                               newly spawned group (handle)
32
        OUT
                  array_of_errcodes
                                               one code per process (array of integer)
33
34
      C binding
35
      int MPI_Comm_spawn(const char *command, char *argv[], int maxprocs,
36
                     MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm,
37
                     int array_of_errcodes[])
38
39
      Fortran 2008 binding
40
      MPI_Comm_spawn(command, argv, maxprocs, info, root, comm, intercomm,
41
                     array_of_errcodes, ierror)
42
          CHARACTER(LEN=*), INTENT(IN) :: command, argv(*)
43
          INTEGER, INTENT(IN) :: maxprocs, root
44
          TYPE(MPI_Info), INTENT(IN) :: info
45
          TYPE(MPI_Comm), INTENT(IN) :: comm
46
          TYPE(MPI_Comm), INTENT(OUT) :: intercomm
47
          INTEGER :: array_of_errcodes(*)
48
```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES, IERROR) CHARACTER*(*) COMMAND, ARGV(*) INTEGER MAXPROCS, INFO, ROOT, COMM, INTERCOMM, ARRAY OF ERRCODES(*), IERROR

MPI_COMM_SPAWN tries to start maxprocs identical copies of the MPI program specified by command, establishing communication with them and returning an intercommunicator. The spawned processes are referred to as children. The children have their own MPI_COMM_WORLD, which is separate from that of the parents. MPI_COMM_SPAWN is collective over comm, and also may not return until MPI_INIT has been called in the children. Similarly, MPI_INIT in the children may not return until all parents have called MPI_COMM_SPAWN. In this sense, MPI_COMM_SPAWN in the parents and MPI_INIT in the children form a collective operation over the union of parent and child processes. The intercommunicator returned by MPI_COMM_SPAWN contains the parent processes in the local group and the child processes in the remote group. The ordering of processes in the local and remote groups is the same as the ordering of the group of the comm in the parents and of MPI_COMM_WORLD of the children, respectively. This intercommunicator can be obtained in the children through the function MPI_COMM_GET_PARENT.

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 intercommunicator can be used immediately). (End of advice to users.)

The command argument The command argument is a string containing the name of a program to be spawned. The string is null-terminated in C. In Fortran, leading and trailing spaces are stripped. MPI does not specify how to find the executable or how the working directory is determined. These rules are implementation-dependent and should be appropriate for the runtime environment.

Advice to implementors. The implementation should use a natural rule for finding 35executables and determining working directories. For instance, a homogeneous sys-36 tem with a global file system might look first in the working directory of the spawning 37 process, or might search the directories in a PATH environment variable as do Unix shells. An implementation on top of PVM would use PVM's rules for finding executables (usually in \$HOME/pvm3/bin/\$PVM_ARCH). An MPI implementation running under POE on an IBM SP would use POE's method of finding executables. An implementation should document its rules for finding executables and determining working directories, and a high-quality implementation should give the user some control over these rules. (End of advice to implementors.)

If the program named in command does not call MPI_INIT, but instead forks a process 46that calls MPI_INIT, the results are undefined. Implementations may allow this case to 47work but are not required to. 48

Unofficial Draft for Comment Only

1 2

3

5

6

7

8 9

10

11

1213

1415

16

17

18

19

20

21

22

23

 24

25

26

2728 29

30

 31

32

33

34

38

39

40

41

42

43

456 CHAPTER 10. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

Advice to users. MPI does not say what happens if the program you start is a shell script and that shell script starts a program that calls MPI_INIT. Though some implementations may allow you to do this, they may also have restrictions, such as requiring that arguments supplied to the shell script be supplied to the program, or requiring that certain parts of the environment not be changed. (*End of advice to users.*)

⁸ The argv argument argv is an array of strings containing arguments that are passed to ⁹ the program. The first element of argv is the first argument passed to command, not, as ¹⁰ is conventional in some contexts, the command itself. The argument list is terminated by ¹¹ NULL in C and an empty string in Fortran. In Fortran, leading and trailing spaces are ¹² always stripped, so that a string consisting of all spaces is considered an empty string. The ¹³ constant MPI_ARGV_NULL may be used in C and Fortran to indicate an empty argument ¹⁴ list. In C this constant is the same as NULL.

¹⁶ Example 10.17 Examples of argv in C and Fortran

¹⁷ To run the program "ocean" with arguments "-gridfile" and "ocean1.grd" in C:

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

or, if not everything is known at compile time:

23		
24	char *command;	
25	char **argv;	
26	<pre>command = "ocean";</pre>	
27	argv=(char **)malloc(3 *	<pre>sizeof(char *));</pre>
28	<pre>argv[0] = "-gridfile";</pre>	
28	<pre>argv[1] = "ocean1.grd";</pre>	
	argv[2] = NULL;	
30	MPI_Comm_spawn(command,	argv,);
31		

³² In Fortran: ³³

1

 $\mathbf{2}$

3

4

5

6

7

15

18

19

20

21

22

34

35

36

37

38

39

```
CHARACTER*25 command, argv(3)

command = 'ocean'

argv(1) = '-gridfile'

argv(2) = 'ocean1.grd'

argv(3) = ''
```

call MPI_COMM_SPAWN(command, argv, ...)

40Arguments are supplied to the program if this is allowed by the operating system. In 41 C, the MPI_COMM_SPAWN argument argv differs from the argv argument of main in two 42respects. First, it is shifted by one element. Specifically, argv[0] of main is provided by the 43implementation and conventionally contains the name of the program (given by command). 44 argv[1] of main corresponds to argv[0] in MPI_COMM_SPAWN, argv[2] of main to argv[1] 45of MPI_COMM_SPAWN, etc. Passing an argv of MPI_ARGV_NULL to MPI_COMM_SPAWN 46results in main receiving argc of 1 and an argv whose element 0 is (conventionally) the 47name of the program. Second, argv of MPI_COMM_SPAWN must be null-terminated, so 48that its length can be determined.

If a Fortran implementation supplies routines that allow a program to obtain its arguments, the arguments may be available through that mechanism. In C, if the operating system does not support arguments appearing in argv of main(), the MPI implementation may add the arguments to the argv that is passed to MPI_INIT.

The maxprocs argument MPI tries to spawn maxprocs processes. If it is unable to spawn maxprocs processes, it raises an error of class MPI_ERR_SPAWN.

An implementation may allow the info argument to change the default behavior, such that if the implementation is unable to spawn all maxprocs processes, it may spawn a smaller number of processes instead of raising an error. In principle, the info argument may specify an arbitrary set $\{m_i : 0 \le m_i \le \text{maxprocs}\}$ of allowed values for the number of processes spawned. The set $\{m_i\}$ does not necessarily include the value maxprocs. If an implementation is able to spawn one of these allowed numbers of processes,

MPI_COMM_SPAWN returns successfully and the number of spawned processes, *m*, is given by the size of the remote group of intercomm. If *m* is less than maxproc, reasons why the other processes were not spawned are given in array_of_errcodes as described below. If it is not possible to spawn one of the allowed numbers of processes, MPI_COMM_SPAWN raises an error of class MPI_ERR_SPAWN.

A spawn call with the default behavior is called *hard*. A spawn call for which fewer than maxprocs processes may be returned is called soft. See Section 10.8.4 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 9.

For the SPAWN calls, info provides additional (and possibly implementation-dependent) instructions to MPI and the runtime system on how to start processes. An application may pass MPI_INFO_NULL in C or Fortran. Portable programs not requiring detailed control over process locations should use MPI_INFO_NULL.

MPI does not specify the content of the info argument, except to reserve a number of special key values (see Section 10.8.4). 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

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 processes is ignored.

 $\mathbf{2}$ 3 The array_of_errcodes argument The array_of_errcodes is an array of length maxprocs in 4 which MPI reports the status of each process that MPI was requested to start. If all maxprocs $\mathbf{5}$ processes were spawned, $\operatorname{array_of}$ errcodes is filled in with the value MPI_SUCCESS. If only m 6 $(0 \le m \le max procs)$ processes are spawned. m of the entries will contain MPI_SUCCESS and 7the rest will contain an implementation-specific error code indicating the reason MPI could 8 not start the process. MPI does not specify which entries correspond to failed processes. 9 An implementation may, for instance, fill in error codes in one-to-one correspondence with 10 a detailed specification in the info argument. These error codes all belong to the error class 11MPI_ERR_SPAWN if there was no error in the argument list. In C or Fortran, an application 12may pass MPI_ERRCODES_IGNORE if it is not interested in the error codes. 13

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.)

¹⁹ MPI_COMM_GET_PARENT(parent)

OUT parent

the parent communicator (handle)

```
23 C binding
```

14

15

16 17 18

20

21 22

27

28

29

32

38

39

40 41

42

43

44

45

46

```
int MPI_Comm_get_parent(MPI_Comm *parent)
```

```
<sup>25</sup> Fortran 2008 binding
```

```
MPI_Comm_get_parent(parent, ierror)
    TYPE(MPI_Comm), INTENT(OUT) :: parent
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
30 Fortran binding
```

```
31 MPI_COMM_GET_PARENT(PARENT, IERROR)
```

INTEGER PARENT, IERROR

³³ If a process was started with MPI_COMM_SPAWN or MPI_COMM_SPAWN_MULTIPLE, ³⁴ MPI_COMM_GET_PARENT returns the "parent" intercommunicator of the current process. ³⁵ This parent intercommunicator is created implicitly inside of MPI_INIT and is the same in-³⁶ tercommunicator returned by SPAWN in the parents.

If the process was not spawned, MPI_COMM_GET_PARENT returns MPI_COMM_NULL. After the parent communicator is freed or disconnected, MPI_COMM_GET_PARENT returns MPI_COMM_NULL.

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

Rationale. The desire of the Forum was to create a constant MPI_COMM_PARENT similar to MPI_COMM_WORLD. Unfortunately such a constant cannot be used (syntactically) as an argument to MPI_COMM_DISCONNECT, which is explicitly allowed. (*End of rationale.*)

10.8.3 Starting Multiple Executables and Establishing Communication

While MPI_COMM_SPAWN is sufficient for most cases, it does not allow the spawning of multiple binaries, or of the same binary with multiple sets of arguments. The following routine spawns multiple binaries or the same binary with multiple sets of arguments, establishing communication with them and placing them in the same MPI_COMM_WORLD.

MPI_COMM_SPAWN_MULTIPLE(count, array_of_commands, array_of_argv, array_of_maxprocs, array_of_info, root, comm, intercomm,

array_of_errcodes)

			16
IN	count	number of commands (positive integer, significant	17
		only at root)	18
IN	array_of_commands	programs to be executed (array of strings, significant	
		only at root)	20
IN	array_of_argv	arguments for commands (array of array of strings, significant only at root)	21
			22
			23
IN	array_of_maxprocs	maximum number of processes to start for each	
		command (array of integers, significant only at root)	25
IN	array_of_info	info objects telling the runtime system where and	26
		how to start processes (array of handles, significant	27
		only at root)	28
IN	root	rank of process in which previous arguments are	29
IIN	1001	examined (integer)	30
		examined (mileger)	31
IN	comm	intracommunicator containing group of spawning	
		processes (handle)	33
OUT	intercomm	intercommunicator between original group and the	
		newly spawned group (handle)	35
OUT	array_of_errcodes	one error code per process (array of integers)	36
001			37
			38
C bindin	0		39
int MPL_C		ount, char *array_of_commands[],	40
	• •	[], const int array_of_maxprocs[],	41
	•	<pre>r_of_info[], int root, MPI_Comm comm,</pre>	42
	MPI_Comm *intercomm,	<pre>int array_of_errcodes[])</pre>	43

Fortran 2008 binding MPI_Comm_spawn_multiple(count, array_of_commands, array_of_argv,

```
array_of_maxprocs, array_of_info, root, comm, intercomm,
array_of_errcodes, ierror)
INTEGER, INTENT(IN) :: count, array_of_maxprocs(*), root
```

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5 6

7

8

9

10

11 12 13

14

15

44

 $45 \\ 46$

47

```
1
          CHARACTER(LEN=*), INTENT(IN) :: array_of_commands(*),
2
                     array_of_argv(count, *)
3
          TYPE(MPI_Info), INTENT(IN) :: array_of_info(*)
4
          TYPE(MPI_Comm), INTENT(IN) :: comm
5
          TYPE(MPI_Comm), INTENT(OUT) :: intercomm
6
          INTEGER :: array_of_errcodes(*)
7
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
     Fortran binding
9
     MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
10
                    ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,
11
                    ARRAY_OF_ERRCODES, IERROR)
12
          INTEGER COUNT, ARRAY_OF_MAXPROCS(*), ARRAY_OF_INFO(*), ROOT, COMM,
13
                     INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR
14
          CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)
15
16
          MPI_COMM_SPAWN_MULTIPLE is identical to MPI_COMM_SPAWN except that there
17
     are multiple executable specifications. The first argument, count, gives the number of
18
     specifications. Each of the next four arguments are simply arrays of the corresponding
19
     arguments in MPI_COMM_SPAWN. For the Fortran version of array_of_argv, the element
20
     array_of_argv(i,j) is the j-th argument to command number i.
21
           Rationale.
                        This may seem backwards to Fortran programmers who are familiar
22
           with Fortran's column-major ordering. However, it is necessary to do it this way to
23
           allow MPI_COMM_SPAWN to sort out arguments. Note that the leading dimension
24
           of array_of_argv must be the same as count. Also note that Fortran rules for sequence
25
           association allow a different value in the first dimension; in this case, the sequence of
26
           array elements is interpreted by MPI_COMM_SPAWN_MULTIPLE as if the sequence is
27
           stored in an array defined with the first dimension set to count. This Fortran feature
28
           allows an implementor to define MPI_ARGVS_NULL (see below) with fixed dimensions,
29
           e.g., (1,1), or only with one dimension, e.g., (1). (End of rationale.)
30
31
           Advice to users. The argument count is interpreted by MPI only at the root, as is
32
           array_of_argv. Since the leading dimension of array_of_argv is count, a non-positive
33
           value of count at a non-root node could theoretically cause a runtime bounds check
34
           error, even though array_of_argv should be ignored by the subroutine. If this happens,
35
           you should explicitly supply a reasonable value of count on the non-root nodes. (End
36
           of advice to users.)
37
38
         In any language, an application may use the constant MPI_ARGVS_NULL (which is likely
39
     to be (char ***)0 in C) to specify that no arguments should be passed to any commands.
40
     The effect of setting individual elements of array_of_argv to MPI_ARGV_NULL is not defined.
41
     To specify arguments for some commands but not others, the commands without arguments
42
     should have a corresponding argv whose first element is null ((char *)0 in C and empty
43
     string in Fortran). In Fortran at non-root processes, the count argument must be set to
44
     a value that is consistent with the provided array_of_argv although the content of these
45
     arguments has no meaning for this operation.
46
          All of the spawned processes have the same MPI_COMM_WORLD. Their ranks in
```

⁴⁷ MPI_COMM_WORLD correspond directly to the order in which the commands are specified

in MPI_COMM_SPAWN_MULTIPLE. Assume that m_1 processes are generated by the first command, m_2 by the second, etc. The processes corresponding to the first command have ranks $0, 1, \ldots, m_1 - 1$. The processes in the second command have ranks $m_1, m_1 + 1, \ldots, m_1 + m_2 - 1$. The processes in the third have ranks $m_1 + m_2, m_1 + m_2 + 1, \ldots, m_1 + m_2 + m_3 - 1$, etc.

Advice to users. Calling MPI_COMM_SPAWN multiple times would create many sets of children with different MPI_COMM_WORLDs whereas MPI_COMM_SPAWN_MULTIPLE creates children with a single MPI_COMM_WORLD, so the two methods are not completely equivalent. There are also two performancerelated reasons why, if you need to spawn multiple executables, you may want to use MPI_COMM_SPAWN_MULTIPLE instead of calling MPI_COMM_SPAWN several times. First, spawning several things at once may be faster than spawning them sequentially. Second, in some implementations, communication between processes spawned at the same time may be faster than communication between processes spawned separately. (End of advice to users.)

The array_of_errcodes argument is a 1-dimensional array of size $\sum_{i=1}^{count} n_i$, where n_i is the *i*-th element of array_of_maxprocs. Command number *i* corresponds to the n_i contiguous slots in this array from element $\sum_{j=1}^{i-1} n_j$ to $\left[\sum_{j=1}^{i} n_j\right] - 1$. Error codes are treated as for MPI_COMM_SPAWN.

Example 10.18 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, ...);
```

Here is how you do it in Fortran:

```
CHARACTER*25 commands(2), array_of_argv(2, 3)
                                                                              36
commands(1) = 'ocean'
                                                                              37
array_of_argv(1, 1) = '-gridfile'
                                                                              38
array_of_argv(1, 2) = 'ocean1.grd'
                                                                              39
array_of_argv(1, 3) = ', '
                                                                              40
                                                                              41
commands(2) = 'atmos'
                                                                              42
array_of_argv(2, 1) = 'atmos.grd'
                                                                              43
array_of_argv(2, 2) = ', '
                                                                              44
                                                                              45
call MPI_COMM_SPAWN_MULTIPLE(2, commands, array_of_argv, ...)
                                                                              46
                                                                              47
```

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

1	10.8.4 Reserved Keys
2 3 4	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.
5 6	host Value is a hostname. The format of the hostname is determined by the implementation.
7 8 9	arch Value is an architecture name. Valid architecture names and what they mean are determined by the implementation.
9 10 11 12	wdir Value is the name of a directory on a machine on which the spawned process(es) execute(s). This directory is made the working directory of the executing process(es). The format of the directory name is determined by the implementation.
13 14 15	path Value is a directory or set of directories where the implementation should look for the executable. The format of path is determined by the implementation.
16 17	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.
18 19 20 21 22 23 24	<pre>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 predefined 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.</pre>
25 26 27 28 29 30 31 32 33	 soft Value specifies a set of numbers which are allowed values for the number of processes that MPI_COMM_SPAWN (et al.) may create. The format of the value is a comma-separated list of Fortran-90 triplets each of which specifies a set of integers and which together specify the set formed by the union of these sets. Negative values in this set and values greater than maxprocs are ignored. MPI will spawn the largest number of processes it can, consistent with some number in the set. The order in which triplets are given is not significant. By Fortran-90 triplets, we mean:
34	1. a means a
35 36	2. a:b means $a, a + 1, a + 2,, b$
37 38 39	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.
40 41	Examples:
42	1. a:b gives a range between a and b
43	2. 0:N gives full "soft" functionality
44 45	3. 1,2,4,8,16,32,64,128,256,512,1024,2048,4096 allows a power-of-two number of processes.
46 47	4. 2:10000:2 allows an even number of processes.
	r

10.8.5 Spawn Example

```
Example 10.19 Manager-worker Example Using MPI_COMM_SPAWN
```

```
/* manager */
                                                                                   5
#include "mpi.h"
                                                                                   6
int main(int argc, char *argv[])
{
   int world_size, universe_size, *universe_sizep, flag;
                                                                                   9
                                                                                   10
  MPI_Comm everyone;
                                /* intercommunicator */
                                                                                   11
   char worker_program[100];
                                                                                   12
                                                                                   13
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);
                                                                                   14
                                                                                   15
                                                                                   16
  if (world_size != 1)
                            error("Top heavy with management");
                                                                                   17
                                                                                   18
  MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,
                                                                                   19
                      &universe_sizep, &flag);
   if (!flag) {
                                                                                   20
                                                                                   21
        printf("This MPI does not support UNIVERSE_SIZE. How many\n\
                                                                                   22
processes total?");
                                                                                   23
        scanf("%d", &universe_size);
                                                                                   24
   } else universe_size = *universe_sizep;
                                                                                   25
   if (universe_size == 1) error("No room to start workers");
                                                                                   26
   /*
                                                                                   27
    * Now spawn the workers. Note that there is a run-time determination
                                                                                   28
                                                                                   29
    * of what type of worker to spawn, and presumably this calculation must
    * be done at run time and cannot be calculated before starting
                                                                                   30
                                                                                   31
    * the program. If everything is known when the application is
                                                                                   32
    * first started, it is generally better to start them all at once
                                                                                   33
    * in a single MPI_COMM_WORLD.
                                                                                   34
   */
                                                                                   35
                                                                                   36
   choose_worker_program(worker_program);
                                                                                   37
  MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, universe_size-1,
                                                                                   38
             MPI_INFO_NULL, 0, MPI_COMM_SELF, &everyone,
                                                                                   39
             MPI_ERRCODES_IGNORE);
                                                                                   40
   /*
                                                                                   41
    * Parallel code here. The communicator "everyone" can be used
                                                                                   42
    * to communicate with the spawned processes, which have ranks 0,...
    * MPI_UNIVERSE_SIZE-1 in the remote group of the intercommunicator
                                                                                   43
                                                                                   44
    * "everyone".
                                                                                   45
    */
                                                                                   46
                                                                                   47
  MPI_Finalize();
                                                                                   48
  return 0;
```

```
1
     }
\mathbf{2}
     /* worker */
3
4
     #include "mpi.h"
5
     int main(int argc, char *argv[])
6
     {
7
         int size;
8
         MPI_Comm parent;
9
         MPI_Init(&argc, &argv);
10
         MPI_Comm_get_parent(&parent);
11
         if (parent == MPI_COMM_NULL) error("No parent!");
12
         MPI_Comm_remote_size(parent, &size);
13
         if (size != 1) error("Something's wrong with the parent");
14
15
         /*
16
          * Parallel code here.
17
          * The manager is represented as the process with rank 0 in (the remote
18
          * group of) the parent communicator. If the workers need to communicate
19
          * among themselves, they can use MPI_COMM_WORLD.
20
          */
21
22
         MPI_Finalize();
23
         return 0;
24
     }
25
26
27
28
29
             Establishing Communication
     10.9
30
     This section provides functions that establish communication between two sets of MPI
31
32
     processes that do not share a communicator.
33
          Some situations in which these functions are useful are:
34
        1. Two parts of an application that are started independently need to communicate.
35
36
        2. A visualization tool wants to attach to a running process.
37
        3. A server wants to accept connections from multiple clients. Both clients and server
38
           may be parallel programs.
39
40
     In each of these situations, MPI must establish communication channels where none existed
41
     before, and there is no parent/child relationship. The routines described in this section
42
     establish communication between the two sets of processes by creating an MPI intercom-
43
     municator, where the two groups of the intercommunicator are the original sets of processes.
44
          Establishing contact between two groups of processes that do not share an existing
45
     communicator is a collective but asymmetric process. One group of processes indicates its
46
     willingness to accept connections from other groups of processes. We will call this group
47
     the (parallel) server, even if this is not a client/server type of application. The other group
48
     connects to the server; we will call it the client.
```

Advice to users. While the names *client* and *server* are used throughout this section, MPI does not guarantee the traditional robustness of client/server systems. The functionality described in this section is intended to allow two cooperating parts of the same application to communicate with one another. For instance, a client that gets a segmentation fault and dies, or one that does not participate in a collective operation may cause a server to crash or hang. (*End of advice to users.*)

10.9.1 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
     10.9.2 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
           Advice to users. The system copies the port name into port_name. The application
42
           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.

43

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
C bindi	no		22
	_Close_port(const cha	ar *port name)	23 24
	-	ar poro_nemo;	24 25
	2008 binding		25 26
	<pre>se_port(port_name, i</pre>		20
	RACTER(LEN=*), INTEN	-	27
INT	EGER, OPTIONAL, INTE	NT(OUT) :: ierror	28 29
Fortran	binding		29 30
	SE_PORT(PORT_NAME, II	ERBOR)	31
	RACTER*(*) PORT_NAME		32
	EGER IERROR		33
			34
			35
MPI_CO	MM_ACCEPT(port_name	e, info, root, comm, newcomm)	36
IN			37
IIN	port_name	port name (string, significant only at root)	38
IN	info	implementation-dependent information (handle,	39
		significant only at root)	40
IN	root	rank in comm of root node (integer)	41
IN	comm	intracommunicator over which call is collective	42
	comm	(handle)	43
			44
OUT	newcomm	intercommunicator with client as remote group	45
		(handle)	46
			47
C bindi	ing		48

----8

 $\mathbf{2}$

 $\overline{7}$

```
1
     int MPI_Comm_accept(const char *port_name, MPI_Info info, int root,
\mathbf{2}
                    MPI_Comm comm, MPI_Comm *newcomm)
3
     Fortran 2008 binding
4
     MPI_Comm_accept(port_name, info, root, comm, newcomm, ierror)
5
          CHARACTER(LEN=*), INTENT(IN) :: port_name
6
          TYPE(MPI_Info), INTENT(IN) :: info
7
          INTEGER, INTENT(IN) :: root
8
          TYPE(MPI_Comm), INTENT(IN) :: comm
9
          TYPE(MPI_Comm), INTENT(OUT) :: newcomm
10
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
     Fortran binding
13
     MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
14
          CHARACTER*(*) PORT_NAME
15
          INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
16
          MPI_COMM_ACCEPT establishes communication with a client. It is collective over the
17
     calling communicator. It returns an intercommunicator that allows communication with the
18
     client.
19
          The port_name must have been established through a call to MPI_OPEN_PORT.
20
         info can be used to provide directives that may influence the behavior of the ACCEPT
21
     call.
22
23
     10.9.3 Client Routines
24
25
     There is only one routine on the client side.
26
27
     MPI_COMM_CONNECT(port_name, info, root, comm, newcomm)
28
29
                                             network address (string, significant only at root)
       IN
                 port_name
30
       IN
                 info
                                             implementation-dependent information (handle,
^{31}
                                             significant only at root)
32
33
       IN
                 root
                                             rank in comm of root node (integer)
34
       IN
                                             intracommunicator over which call is collective
                 comm
35
                                             (handle)
36
       OUT
                                             intercommunicator with server as remote group
                 newcomm
37
                                             (handle)
38
39
     C binding
40
     int MPI_Comm_connect(const char *port_name, MPI_Info info, int root,
41
                    MPI_Comm comm, MPI_Comm *newcomm)
42
43
     Fortran 2008 binding
44
     MPI_Comm_connect(port_name, info, root, comm, newcomm, ierror)
45
          CHARACTER(LEN=*), INTENT(IN) :: port_name
46
          TYPE(MPI_Info), INTENT(IN) :: info
47
          INTEGER, INTENT(IN) :: root
48
```

```
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)
```

PI_COMM_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM CHARACTER*(*) PORT_NAME INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR

This routine establishes communication with a server specified by port_name. It is collective over the calling communicator and returns an intercommunicator in which the remote group participated in an MPI_COMM_ACCEPT.

If the named port does not exist (or has been closed), MPI_COMM_CONNECT raises an error of class MPI_ERR_PORT.

If the port exists, but does not have a pending MPI_COMM_ACCEPT, the connection attempt will eventually time out after an implementation-defined time, or succeed when the server calls MPI_COMM_ACCEPT. In the case of a time out, MPI_COMM_CONNECT raises an error of class MPI_ERR_PORT.

Advice to implementors. The time out period may be arbitrarily short or long. However, a high-quality implementation will try to queue connection attempts so that a server can handle simultaneous requests from several clients. A high-quality implementation may also provide a mechanism, through the info arguments to MPI_OPEN_PORT, MPI_COMM_ACCEPT, and/or MPI_COMM_CONNECT, for the user to control timeout and queuing behavior. (*End of advice to implementors.*)

MPI provides no guarantee of fairness in servicing connection attempts. That is, connection attempts are not necessarily satisfied in the order they were initiated and competition from other connection attempts may prevent a particular connection attempt from being satisfied.

port_name is the address of the server. It must be the same as the name returned by MPI_OPEN_PORT on the server. Some freedom is allowed here. If there are equivalent forms of port_name, an implementation may accept them as well. For instance, if port_name is (hostname:port), an implementation may accept (ip_address:port) as well.

10.9.4 Name Publishing

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

 31

470 CHAPTER 10. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

1 MPI_PUBLISH_NAME(service_name, info, port_name) 2 IN service_name a service name to associate with the port (string) 3 IN info implementation-specific information (handle) 4 5IN port_name a port name (string) 6 $\overline{7}$ C binding 8 int MPI_Publish_name(const char *service_name, MPI_Info info, 9 const char *port_name) 10 Fortran 2008 binding 11 MPI_Publish_name(service_name, info, port_name, ierror) 12CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name 13 TYPE(MPI_Info), INTENT(IN) :: info 14INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1516Fortran binding 17MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR) 18 CHARACTER*(*) SERVICE_NAME, PORT_NAME 19INTEGER INFO, IERROR 20This routine publishes the pair (port_name, service_name) so that an application may 21retrieve a system-supplied port_name using a well-known service_name. 22The implementation must define the *scope* of a published service name, that is, the 23domain over which the service name is unique, and conversely, the domain over which the 24 (port name, service name) pair may be retrieved. For instance, a service name may be 25unique to a job (where job is defined by a distributed operating system or batch scheduler), 26unique to a machine, or unique to a Kerberos realm. The scope may depend on the info 27argument to MPI_PUBLISH_NAME. 28MPI permits publishing more than one service_name for a single port_name. On the 29 other hand, if service_name has already been published within the scope determined by info, 30 the behavior of MPI_PUBLISH_NAME is undefined. An MPI implementation may, through 31 a mechanism in the info argument to MPI_PUBLISH_NAME, provide a way to allow multiple 32 servers with the same service in the same scope. In this case, an implementation-defined 33 policy will determine which of several port names is returned by MPI_LOOKUP_NAME. 34Note that while service_name has a limited scope, determined by the implementation, 35 port_name always has global scope within the communication universe used by the imple-36 mentation (i.e., it is globally unique). 37 port_name should be the name of a port established by MPI_OPEN_PORT and not yet 38 released by MPI_CLOSE_PORT. If it is not, the result is undefined. 39 40 In some cases, an MPI implementation may use a name Advice to implementors. 41 service that a user can also access directly. In this case, a name published by MPI 42could easily conflict with a name published by a user. In order to avoid such conflicts, 43 MPI implementations should mangle service names so that they are unlikely to conflict 44 with user code that makes use of the same service. Such name mangling will of course 45be completely transparent to the user. 46 47 The following situation is problematic but unavoidable, if we want to allow implemen-48 tations to use nameservers. Suppose there are multiple instances of "ocean" running

oce	ans can coexist. If an in	be of a service name is confined to a job, then multiple aplementation provides site-wide scope, however, multiple s all calls to MPI_PUBLISH_NAME after the first may fail.	1 2 3			
Th	ere is no universal soluti	on to this.	4			
То	handle these situations.	a high-quality implementation should make it possible to	5			
		h names are published. (End of advice to implementors.)	6			
			7			
			8 9			
MPI UN	PUBLISH_NAME(service	name info port name)	9 10			
IN IN	· ·		11			
	service_name	a service name (string)	12			
IN	info	implementation-specific information (handle)	13			
IN	port_name	a port name (string)	14			
a 1 · 1 ·			15 16			
C bindi	•	t char *service_name, MPI_Info info,	17			
THO LUT	const char *po		18			
-	_		19			
	2008 binding	nome info port nome intropy	20			
		name, info, port_name, ierror) Γ(IN) :: service_name, port_name	21			
	E(MPI_Info), INTENT(I		22 23			
	EGER, OPTIONAL, INTER		23 24			
Fortran binding MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)						
CHARACTER*(*) SERVICE_NAME, PORT_NAME, TEAMONT						
	EGER INFO, IERROR		28			
			29			
		ervice name that has been previously published. Attempt- us not been published or has already been unpublished is	30			
0	•	error class MPI_ERR_SERVICE.	31 32			
	· · · ·	e unpublished before the corresponding port is closed and	33			
		s. The behavior of MPI_UNPUBLISH_NAME is implemen-	34			
		tries to unpublish a name that it did not publish.	35			
		ed with MPI_PUBLISH_NAME to tell the implementation	36			
		ementation may require that info passed to	37			
	PUBLISH_NAME contain	n information to tell the implementation how to unpublish	38			
a name.			39			
			40 41			
MPI_LO	DKUP_NAME(service_na	me, info, port_name)	42			
IN	service_name	a service name (string)	43			
IN	info	implementation-specific information (handle)	44			
OUT	port_name	a port name (string)	45			
	L	· · · · · · · · · · · · · · · · · · ·	46			
C bindi	ng		47 48			
	-		-10			

1 2	<pre>int MPI_Lookup_name(const char *service_name, MPI_Info info,</pre>
3 4 5 6 7 8 9	<pre>Fortran 2008 binding MPI_Lookup_name(service_name, info, port_name, ierror) CHARACTER(LEN=*), INTENT(IN) :: service_name TYPE(MPI_Info), INTENT(IN) :: info CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>
10 11 12 13	Fortran binding MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR) CHARACTER*(*) SERVICE_NAME, PORT_NAME INTEGER INFO, IERROR
14 15 16 17 18 19 20 21 22	This function retrieves a port_name published by MPI_PUBLISH_NAME with service_name. If service_name has not been published, it raises an error in the error class MPI_ERR_NAME. The application must supply a port_name buffer large enough to hold the largest possible port name (see discussion above under MPI_OPEN_PORT). If an implementation allows multiple entries with the same service_name within the same scope, a particular port_name is chosen in a way determined by the implementation. If the info argument was used with MPI_PUBLISH_NAME to tell the implementation how to publish names, a similar info argument may be required for MPI_LOOKUP_NAME.
23 24 25 26	10.9.5 Reserved Key Values The following key values are reserved. An implementation is not required to interpret these key values, but if it does interpret the key value, it must provide the functionality described.
27 28 29 30	<pre>ip_port Value contains IP port number at which to establish a port. (Reserved for MPI_OPEN_PORT only).</pre>
31 32 33 34	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).
35 36	10.9.6 Client/Server Examples
37 38 39 40 41	Example 10.20 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:
42 43 44 45	
46 47 48	

```
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 10.21 Ocean/Atmosphere — Relies on Name Publishing
In this example, the "ocean" application is the "server" side of a coupled ocean-atmosphere
climate model. It assumes that the MPI implementation publishes names.
```

```
25
    MPI_Open_port(MPI_INFO_NULL, port_name);
                                                                                     26
    MPI_Publish_name("ocean", MPI_INFO_NULL, port_name);
                                                                                     27
                                                                                     28
    MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
                                                                                     29
    /* do something with intercomm */
                                                                                     30
    MPI_Unpublish_name("ocean", MPI_INFO_NULL, port_name);
                                                                                     31
                                                                                     32
                                                                                     33
On the client side:
                                                                                     34
                                                                                     35
    MPI_Lookup_name("ocean", MPI_INFO_NULL, port_name);
                                                                                     36
    MPI_Comm_connect(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF,
                                                                                    37
                       &intercomm);
                                                                                     38
```

```
Example 10.22 Simple Client-Server Example
This is a simple example; the server accepts only a single connection at a time and serves
```

that connection until the client requests to be disconnected. The server is a single process.Here is the server. It accepts a single connection and then processes data until it receives a message with tag 1. A message with tag 0 tells the server to exit.

Unofficial Draft for Comment Only

1 2

6

9 10

11

12

13 14

15

16

17

18

19 20

21

22

23 24

39

40

41

42

43

```
1
     #include "mpi.h"
\mathbf{2}
     int main(int argc, char *argv[])
3
     {
4
         MPI_Comm client;
5
         MPI_Status status;
6
         char port_name[MPI_MAX_PORT_NAME];
7
         double buf[MAX_DATA];
8
         int
                 size, again;
9
10
         MPI_Init(&argc, &argv);
11
         MPI_Comm_size(MPI_COMM_WORLD, &size);
12
         if (size != 1) error(FATAL, "Server too big");
13
         MPI_Open_port(MPI_INFO_NULL, port_name);
14
         printf("server available at %s\n", port_name);
15
         while (1) {
16
              MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
17
                                &client);
18
              again = 1;
19
              while (again) {
20
                  MPI_Recv(buf, MAX_DATA, MPI_DOUBLE,
21
                            MPI_ANY_SOURCE, MPI_ANY_TAG, client, &status);
22
                  switch (status.MPI_TAG) {
23
                       case 0: MPI_Comm_free(&client);
24
                                MPI_Close_port(port_name);
25
                                MPI_Finalize();
26
                                return 0;
27
                       case 1: MPI_Comm_disconnect(&client);
28
                                again = 0;
29
                                break;
30
                       case 2: /* do something */
31
                       . . .
32
                       default:
33
                                /* Unexpected message type */
34
                                MPI_Abort(MPI_COMM_WORLD, 1);
35
                       }
36
37
              }
38
     }
39
     Here is the client.
40
41
     #include "mpi.h"
42
     int main(int argc, char **argv)
43
     {
44
         MPI_Comm server;
45
         double buf[MAX_DATA];
46
         char port_name[MPI_MAX_PORT_NAME];
47
48
```

```
MPI_Init(&argc, &argv);
strcpy(port_name, argv[1]);/* assume server's name is cmd-line arg */
MPI_Comm_connect(port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
                 &server):
while (!done) {
    tag = 2; /* Action to perform */
    MPI_Send(buf, n, MPI_DOUBLE, 0, tag, server);
    /* etc */
    }
MPI_Send(buf, 0, MPI_DOUBLE, 0, 1, server);
MPI_Comm_disconnect(&server);
MPI_Finalize();
return 0;
```

10.10 Other Functionality

10.10.1 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 28the application to obtain this information in a portable manner. This attribute indicates 29the total number of processes that are expected. In Fortran, the attribute is the integer 30 value. In C, the attribute is a pointer to the integer value. An application typically subtracts 31the size of MPI_COMM_WORLD from MPI_UNIVERSE_SIZE to find out how many processes it should spawn. MPI_UNIVERSE_SIZE is initialized in MPI_INIT and is not changed by MPI. If 33 defined, it has the same value on all processes of MPI_COMM_WORLD. MPI_UNIVERSE_SIZE 34 is determined by the application startup mechanism in a way not specified by MPI. (The 35 size of MPI_COMM_WORLD is another example of such a parameter.) 36

Possibilities for how MPI_UNIVERSE_SIZE might be set include

- A -universe_size argument to a program that starts MPI processes.
- Automatic interaction with a batch scheduler to figure out how many processors have been allocated to an application.
- An environment variable set by the user.
- Extra information passed to MPI_COMM_SPAWN through the info argument.

An implementation must document how MPI_UNIVERSE_SIZE is set. An implementation may not support the ability to set MPI_UNIVERSE_SIZE, in which case the attribute

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

4

5 6

9 10

11

12

13 14

15

1617 18

19 20

21

22

23

24

25

26

27

32

37 38

39 40

41

42

43 44

4546

47

476 CHAPTER 10. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

¹ 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.

⁶ MPI_UNIVERSE_SIZE is assumed to have been specified when an application was started, ⁷ and is in essence a portable mechanism to allow the user to pass to the application (through ⁸ the MPI process startup mechanism, such as mpiexec) a piece of critical runtime informa-⁹ tion. Note that no interaction with the runtime environment is required. If the runtime ¹⁰ environment changes size while an application is running, MPI_UNIVERSE_SIZE is not up-¹¹ dated, and the application must find out about the change through direct communication ¹² with the runtime system.

13 14

15

10.10.2 Singleton MPI Initialization

¹⁶ A high-quality implementation will allow any process (including those not started with a ¹⁷ "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.

22 23

24

25

26

27

28

39 40

41

42

43

44 45

46 47 48 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.

- ²⁹ Considering the World Model, when an application enters MPI_INIT, clearly it must ³⁰ be able to determine if these special steps were taken. If a process enters MPI_INIT and ³¹ determines that no special steps were taken (i.e., it has not been given the information ³² to form an MPI_COMM_WORLD with other processes) it succeeds and forms a singleton ³³ MPI program, that is, one in which MPI_COMM_WORLD has size 1.
- ³⁴ In some implementations, MPI may not be able to function without an "MPI environ-³⁵ ment." For example, MPI may require that daemons be running or MPI may not be ³⁶ able to work at all on the front-end of an MPP. In this case, an MPI implementation ³⁷ may either
 - 1. Create the environment (e.g., start a daemon) or
 - 2. Raise an error if it cannot create the environment and the environment has not been started independently.

A high-quality implementation will try to create a singleton MPI process and not raise an error.

(End of advice to implementors.)

10.10.3 MPI_APPNUM 1 $\mathbf{2}$ There is a predefined attribute MPI_APPNUM of MPI_COMM_WORLD. In Fortran, the at-3 tribute is an integer value. In C, the attribute is a pointer to an integer value. If a process 4 was spawned with MPI_COMM_SPAWN_MULTIPLE, MPI_APPNUM is the command number 5that generated the current process. Numbering starts from zero. If a process was spawned 6 with MPI_COMM_SPAWN, it will have MPI_APPNUM equal to zero. 7 Additionally, if the process was not started by a spawn call, but by an implementation-8 specific startup mechanism that can handle multiple process specifications, MPI_APPNUM 9 should be set to the number of the corresponding process specification. In particular, if it 10 is started with 11 12mpiexec spec0 [: spec1 : spec2 : ...] 13 MPI_APPNUM should be set to the number of the corresponding specification. 14If an application was not spawned with MPI_COMM_SPAWN or 15MPI_COMM_SPAWN_MULTIPLE, and MPI_APPNUM does not make sense in the context of 16the implementation-specific startup mechanism, MPL_APPNUM is not set. 17 MPI implementations may optionally provide a mechanism to override the value of 18 MPI_APPNUM through the info argument. MPI reserves the following key for all SPAWN 19 calls. 2021appnum Value contains an integer that overrides the default value for MPI_APPNUM in the 22 child. 23 24 *Rationale.* When a single application is started, it is able to figure out how many pro-25cesses there are by looking at the size of MPI_COMM_WORLD. An application consisting 26of multiple SPMD sub-applications has no way to find out how many sub-applications 27there are and to which sub-application the process belongs. While there are ways to 28figure it out in special cases, there is no general mechanism. MPI_APPNUM provides 29such a general mechanism. (End of rationale.) 30 3110.10.4 **Releasing Connections** 32 33 Before a client and server connect, they are independent MPI applications. An error in one 34does 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 3536 server to be able to disconnect, so that an error in one will not affect the other. Similarly, 37 it might be desirable for a parent and child to disconnect, so that errors in the child do not 38 affect the parent, or vice-versa. 39 • Two processes are **connected** if there is a communication path (direct or indirect) 40 between them. More precisely: 41 421. Two processes are connected if 43 (a) they both belong to the same communicator (inter- or intra-, including 44MPI_COMM_WORLD) or 45(b) they have previously belonged to a communicator that was freed with 46MPI_COMM_FREE instead of MPI_COMM_DISCONNECT or 47

(c) they both belong to the group of the same window or filehandle.

Unofficial Draft for Comment Only

478 CHAPTER 10. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

1	2. If A is connected to B and B to C, then A is connected to C.
$\frac{2}{3}$	• Two processes are disconnected (also independent) if they are not connected.
4 5 6 7	• By the above definitions, connectivity is a transitive property, and divides the universe of MPI processes into disconnected (independent) sets (equivalence classes) of processes.
8 9 10	• Processes which are connected, but do not share the same MPI_COMM_WORLD, may become disconnected (independent) if the communication path between them is broken by using MPI_COMM_DISCONNECT.
11 12	The following additional rules apply to MPI routines in other chapters:
13	• MPI_FINALIZE is collective over a set of connected processes.
14 15 16 17 18	• MPI_ABORT does not abort independent processes. It may abort all processes in the caller's MPI_COMM_WORLD (ignoring its comm argument). Additionally, it may abort connected processes as well, though it makes a "best attempt" to abort only the processes in comm.
19 20 21	• If a process terminates without calling MPI_FINALIZE, independent processes are not affected but the effect on connected processes is not defined.
22 23 24 25 26 27 28 29 30	Advice to implementors. In practice, it may be difficult to distinguish between an MPI process failure and an erroneous program that terminates without calling an MPI finalization function: an implementation that defines semantics for process failure management may have to exhibit the behavior defined for MPI process failures with such erroneous programs. A high quality implementation should exhibit a different behavior for erroneous programs and MPI process failures. (<i>End of advice to implementors.</i>)
31 32	MPI_COMM_DISCONNECT(comm)
33 34 35	INOUT comm communicator (handle)
36 37	C binding int MPI_Comm_disconnect(MPI_Comm *comm)
38 39 40 41	Fortran 2008 binding MPI_Comm_disconnect(comm, ierror) TYPE(MPI_Comm), INTENT(INOUT) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42 43 44 45	Fortran binding MPI_COMM_DISCONNECT(COMM, IERROR) INTEGER COMM, IERROR
46 47 48	This function waits for all pending communication on comm to complete internally, deallocates the communicator object, and sets the handle to MPI_COMM_NULL. It is a collective operation.

It may not be called with the communicator MPI_COMM_WORLD or MPI_COMM_SELF.

MPI_COMM_DISCONNECT may be called only if all communication is complete and matched, so that buffered data can be delivered to its destination. This requirement is the same as for MPI_FINALIZE.

MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes.

Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (*End of advice to users.*)

Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (*End of rationale.*)

10.10.5 Another Way to Establish MPI Communication

MPI_COM	A_JOIN(fd, intercomm)	
IN	fd	socket file descriptor
OUT	intercomm	new intercommunicator (handle)
C binding	omm_join(int fd, MPI_Comm	*intercomm)
Fortran 2	008 binding	
MPI_Comm_	join(fd, intercomm, ierro	r)
	ER, INTENT(IN) :: fd	
TYPE(I	<pre>MPI_Comm), INTENT(OUT) ::</pre>	intercomm
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror
Fortran b	inding	
MPI_COMM_	JOIN(FD, INTERCOMM, IERRO	R)
INTEG	ER FD, INTERCOMM, IERROR	

MPI_COMM_JOIN is intended for MPI implementations that exist in an environment supporting the Berkeley Socket interface [47, 52]. Implementations that exist in an environment not supporting Berkeley Sockets should provide the entry point for MPI_COMM_JOIN and should return MPI_COMM_NULL.

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

Advice to users. An MPI implementation may require a specific communication medium for MPI communication, such as a shared memory segment or a special switch.

Unofficial Draft for Comment Only

 $45 \\ 46$

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.*)

3 4 5

6

7

8

9 10

1

2

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

fd is a file descriptor representing a socket of type SOCK_STREAM (a two-way reliable byte-stream connection). Nonblocking I/O and asynchronous notification via SIGIO must not be enabled for the socket. The socket must be in a connected state. The socket must be quiescent when MPI_COMM_JOIN is called (see below). It is the responsibility of the application to create the socket using standard socket API calls.

MPI_COMM_JOIN must be called by the process at each end of the socket. It does not
 return until both processes have called MPI_COMM_JOIN. The two processes are referred
 to as the local and remote processes.

¹⁹ MPI uses the socket to bootstrap creation of the intercommunicator, and for nothing ²⁰ else. Upon return from MPI_COMM_JOIN, the file descriptor will be open and quiescent ²¹ (see below).

If MPI is unable to create an intercommunicator, but is able to leave the socket in its original state, with no pending communication, it succeeds and sets intercomm to MPI_COMM_NULL.

The socket must be quiescent before MPL_COMM_JOIN is called and after 25MPI_COMM_JOIN returns. More specifically, on entry to MPI_COMM_JOIN, a read on the 26socket will not read any data that was written to the socket before the remote process called 27MPI_COMM_JOIN. On exit from MPI_COMM_JOIN, a read will not read any data that was 28written to the socket before the remote process returned from MPI_COMM_JOIN. It is the 29 responsibility of the application to ensure the first condition, and the responsibility of the 30 MPI implementation to ensure the second. In a multithreaded application, the application 31 must ensure that one thread does not access the socket while another is calling 32 MPI_COMM_JOIN, or call MPI_COMM_JOIN concurrently. 33

Advice to implementors. MPI is free to use any available communication path(s) for MPI messages in the new communicator; the socket is only used for the initial handshaking. (*End of advice to implementors.*)

MPI_COMM_JOIN uses non-MPI communication to do its work. The interaction of non-MPI communication with pending MPI communication is not defined. Therefore, the result of calling MPI_COMM_JOIN on two connected processes (see Section 10.10.4 for the definition of connected) is undefined.

The returned communicator may be used to establish MPI communication with additional processes, through the usual MPI communicator creation mechanisms.

- 44 45
- 46

34

35

36

37 38

39

40

41

42

- 47
- 48

Chapter 11

One-Sided Communications

11.1 Introduction

Remote Memory Access (RMA) extends the communication mechanisms of MPI by allowing one process to specify all communication parameters, both for the sending side and for the receiving side. This mode of communication facilitates the coding of some applications with dynamically changing data access patterns where the data distribution is fixed or slowly changing. In such a case, each process can compute what data it needs to access or to update at other processes. However, the programmer may not be able to easily determine which data in a process may need to be accessed or to be updated by operations executed by a different process, and may not even know which processes may perform such updates. Thus, the transfer parameters are all available only on one side. Regular send/receive communication requires matching operations by sender and receiver. In order to issue the matching operations, an application needs to distribute the transfer parameters. This distribution may require all processes to participate in a time-consuming global computation, or to poll for potential communication requests to receive and upon which to act periodically. The use of RMA communication mechanisms avoids the need for global computations or explicit polling. A generic example of this nature is the execution of an assignment of the form A = B(map), where map is a permutation vector, and A, B, and map are distributed in the same manner.

Message-passing communication achieves two effects: *communication* of data from sender to receiver and *synchronization* of sender with receiver. The RMA design separates these two functions. The following communication calls are provided:

- Remote write: MPI_PUT, MPI_RPUT
- Remote read: MPI_GET, MPI_RGET
- Remote update: MPI_ACCUMULATE, MPI_RACCUMULATE
- Remote read and update: MPI_GET_ACCUMULATE, MPI_RGET_ACCUMULATE, and MPI_FETCH_AND_OP
- Remote atomic swap operations: MPI_COMPARE_AND_SWAP

This chapter refers to an operations set that includes all remote update, remote read and update, and remote atomic swap operations as "accumulate" operations.

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 $\overline{7}$ in detail in Section 11.4. 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.

11.2 Initialization

20

21

²² MPI provides the following window initialization functions: MPI_WIN_CREATE,

²³ MPI_WIN_ALLOCATE, MPI_WIN_ALLOCATE_SHARED, and

²⁴ MPI_WIN_CREATE_DYNAMIC, which are collective on an intracommunicator.

²⁵ MPI_WIN_CREATE allows each process to specify a "window" in its memory that is made ²⁶ accessible to accesses by remote processes. The call returns an opaque object that represents ²⁷ the group of processes that own and access the set of windows, and the attributes of each ²⁸ window, as specified by the initialization call. MPI_WIN_ALLOCATE differs from

²⁵ MPI_WIN_CREATE in that the user does not pass allocated memory;

³¹ MPI_WIN_ALLOCATE returns a pointer to memory allocated by the MPI implementation. ³¹ MPI_WIN_ALLOCATE_SHARED differs from MPI_WIN_ALLOCATE in that the allocated ³² memory can be accessed from all processes in the window's group with direct load/store ³³ instructions. Some restrictions may apply to the specified communicator.

³⁴ MPI_WIN_CREATE_DYNAMIC creates a window that allows the user to dynamically control ³⁵ which memory is exposed by the window.

1

39

40

41

4243

44

45

46

47

48

11.2.1	Window Creation		1				
			2				
			3				
MPI_WI	N_CREATE(base, size)	ze, disp_unit, info, comm, win)	4				
IN	base	initial address of window (choice)	5 6				
IN	size	size of window in bytes (non-negative integer)					
IN	disp_unit	local unit size for displacements, in bytes (positive	8				
		integer)	9				
IN	info	info argument (handle)	10 11				
IN	comm	intra-communicator (handle)	12				
OUT	win	window object (handle)	13				
			14				
C bind	ing		15				
int MPI		*base, MPI_Aint size, int disp_unit, MPI_Info info,	16 17				
	MPI_Comm c	comm, MPI_Win *win)	18				
Fortran 2008 binding							
MPI_Win_create(base, size, disp_unit, info, comm, win, ierror)							
TYF	PE(*), DIMENSION(), ASYNCHRONOUS :: base	21				
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size							
	INTEGER, INTENT(IN) :: disp_unit						
	PE(MPI_Info), INT		24				
	PE(MPI_Comm), INT		25				
	PE(MPI_Win), INTE		26				
1N1	EGER, UPTIONAL,	INTENT(OUT) :: ierror	27 28				
Fortrar	n binding		20				
MPI_WIN	I_CREATE(BASE, SI	ZE, DISP_UNIT, INFO, COMM, WIN, IERROR)	30				
•	<pre>pe> BASE(*)</pre>		31				
	TEGER(KIND=MPI_AD		32				
INT	EGER DISP_UNIT,	INFO, COMM, WIN, IERROR	33				
Thi	s is a collective cal	l executed by all processes in the group of comm. It returns	34				
		be used by these processes to perform RMA operations. Each	35				
		of existing memory that it exposes to RMA accesses by the	36				
processe	s in the group of c	comm. The window consists of size bytes, starting at address	37				
base. In C, base is the starting address of a memory region. In Fortran, one can pass the							

base. In C, base is the starting address of a memory region. In Fortran, one can pass the first element of a memory region or a whole array, which must be 'simply contiguous' (for 'simply contiguous,' see also Section 18.1.12). A process may elect to expose no memory by specifying size = 0.

The displacement unit argument is provided to facilitate address arithmetic in RMA operations: the target displacement argument of an RMA operation is scaled by the factor disp_unit specified by the target process, at window creation.

Rationale. The window size is specified using an address-sized integer, rather than a basic integer type, to allow windows that span more memory than can be described with a basic integer type. (End of rationale.)

	404	CHAITER II. ONE-SIDED COMMUNICATIONS
1 2 3 4 5 6	siz latte corr	vice to users. Common choices for disp_unit are 1 (no scaling), and (in C syntax) eof(type), for a window that consists of an array of elements of type type. The er choice will allow one to use array indices in RMA calls, and have those scaled rectly to byte displacements, even in a heterogeneous environment. (<i>End of advice users.</i>)
7 8		info argument provides optimization hints to the runtime about the expected usage f the window. The following info keys are predefined:
9 10 11 12 13	niza wine	— if set to true, then the implementation may assume that passive target synchro- ation (i.e., MPI_WIN_LOCK, MPI_WIN_LOCK_ALL) will not be used on the given dow. This implies that this window is not used for 3-party communication, and A can be implemented with no (less) asynchronous agent activity at this process.
13 14 15		$te_ordering$ — controls the ordering of accumulate operations at the target. See tion 11.7.2 for details.
16 17 18 19 20 21 22	accu same calls elim	te_ops — if set to same_op, the implementation will assume that all concurrent umulate calls to the same target address will use the same operation. If set to e_op_no_op, then the implementation will assume that all concurrent accumulate s to the same target address will use the same operation or MPI_NO_OP. This can ninate the need to protect access for certain operation types where the hardware guarantee atomicity. The default is same_op_no_op.
23 24 25	ider	e — if set to true, then the implementation may assume that the argument size is indical on all processes, and that all processes have provided this info key with the ne value.
26 27 28 29	disp	p_unit — if set to true, then the implementation may assume that the argument p_unit is identical on all processes, and that all processes have provided this info with the same value.
30 31 32 33	to q It is	vice to users. The info query mechanism described in Section 11.2.7 can be used query the specified info arguments for windows that have been passed to a library. Is recommended that libraries check attached info keys for each passed window. In advice to users.)
34 35 36 37 38 39 40	windows, put and a should po associated	various processes in the group of comm may specify completely different target in location, size, displacement units, and info arguments. As long as all the get, accumulate accesses to a particular process fit their specific target window this use no problem. The same area in memory may appear in multiple windows, each d with a different window object. However, concurrent communications to distinct, ng windows may lead to undefined results.
41 42 43 44 45 46 47 48	proc can ifica proc imp the	<i>ionale.</i> The reason for specifying the memory that may be accessed from another cess in an RMA operation is to permit the programmer to specify what memory be a target of RMA operations and for the implementation to enforce that spectrion. For example, with this definition, a server process can safely allow a client cess to use RMA operations, knowing that (under the assumption that the MPI elementation does enforce the specified limits on the exposed memory) an error in client cannot affect any memory other than what was explicitly exposed. (<i>End of conale.</i>)

CHAPTER 11. ONE-SIDED COMMUNICATIONS

Advice to users. A window can be created in any part of the process memory. However, on some systems, the performance of windows in memory allocated by MPI_ALLOC_MEM (Section 8.2) will be better. Also, on some systems, performance is improved when window boundaries are aligned at "natural" boundaries (word, double-word, cache line, page frame, etc.). (End of advice to users.)

Advice to implementors. In cases where RMA operations use different mechanisms in different memory areas (e.g., load/store in a shared memory segment, and an asynchronous handler in private memory), the MPI_WIN_CREATE call needs to figure out which type of memory is used for the window. To do so, MPI maintains, internally, the list of memory segments allocated by MPI_ALLOC_MEM, or by other, implementation-specific, mechanisms, together with information on the type of memory segment allocated. When a call to MPI_WIN_CREATE occurs, then MPI checks which segment contains each window, and decides, accordingly, which mechanism to use for RMA operations.

Vendors may provide additional, implementation-specific mechanisms to allocate or to specify memory regions that are preferable for use in one-sided communication. In particular, such mechanisms can be used to place static variables into such preferred regions.

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

11.2.2 Window That Allocates Memory

MPI_WIN_ALLOCATE(size, disp_unit, info, comm, baseptr, win)					
IN	size size of window in bytes (non-negative integer)				
IN	disp_unit	local unit size for displacements, in bytes (positive integer)	29 30 31		
IN	info	info argument (handle)	32		
IN	comm	intra-communicator (handle)	33		
OUT	baseptr	initial address of window (choice)	$\frac{34}{35}$		
OUT	win	window object returned by call (handle)	36		
			37		
C binding			38 39		
<pre>int MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info,</pre>					
MPI_Comm comm, void *baseptr, MPI_Win *win)					
Fortran 2	008 binding		41 42		
	0	info, comm, baseptr, win, ierror)	43		
USE, I	INTRINSIC :: ISO_C_BINDIN	G, ONLY : C_PTR	44		
INTEG	ER(KIND=MPI_ADDRESS_KIND)	, INTENT(IN) :: size	45		
	ER, INTENT(IN) :: disp_un		46		
	MPI_Info), INTENT(IN) ::		47		
TYPE(1	MPI_Comm), INTENT(IN) ::	comm	48		

MPL WIN ALLOCATE(size disp unit info comm baseptr win)

1

 $\mathbf{2}$

3

4

5

6

 $\overline{7}$

8

9

10

11

12

13

14

1516

17

18

19

20

21

2223

 24 2526

1	TYPE(C_PTR), INTENT(OUT) :: baseptr
2	TYPE(MPI_Win), INTENT(OUT) :: win
3	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4	Fortran binding
5	MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
6	INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
7	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
8	
9	This is a collective call executed by all processes in the group of comm. On each
10	process, it allocates memory of at least size bytes, returns a pointer to it, and returns a
11	window object that can be used by all processes in comm to perform RMA operations. The
12	returned memory consists of size bytes local to each process, starting at address baseptr
13	and is associated with the window as if the user called MPI_WIN_CREATE on existing
14	memory. The size argument may be different at each process and $size = 0$ is valid; however, a
15	library might allocate and expose more memory in order to create a fast, globally symmetric
16	allocation. The discussion of and rationales for MPI_ALLOC_MEM and MPI_FREE_MEM in
17	Section 8.2 also apply to MPI_WIN_ALLOCATE; in particular, see the rationale in Section 8.2
18	for an explanation of the type used for baseptr .
19	If the Fortran compiler provides $\texttt{TYPE}(\texttt{C}_\texttt{PTR})$, then the following generic interface must
20	be provided in the mpi module and should be provided in mpif.h through overloading,
21 22	i.e., with the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND)
22	BASEPTR, but with a different specific procedure name:
23	INTERFACE MPI_WIN_ALLOCATE
25	SUBROUTINE MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
26	WIN, IERROR)
27	IMPORT :: MPI_ADDRESS_KIND
28	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
29	INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
30	END SUBROUTINE
31	SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
32	WIN, IERROR)
33	USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
34	IMPORT :: MPI_ADDRESS_KIND
35	INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
36	INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
37	TYPE(C_PTR) :: BASEPTR
38	END SUBROUTINE
39	END INTERFACE
40	The base procedure name of this overloaded function is MPI_WIN_ALLOCATE_CPTR.
41	The implied specific procedure names are described in Section 18.1.5.
42	The implied specific procedure numes are described in Section 10.1.0.
43	Rationale. By allocating (potentially aligned) memory instead of allowing the user
44	to pass in an arbitrary buffer, this call can improve the performance for systems with
45	remote direct memory access. This also permits the collective allocation of memory
46	and supports what is sometimes called the "symmetric allocation" model that can be

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.*)

Unofficial Draft for Comment Only

47

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 8.2 apply to all processes with non-zero size argument.

11.2.3 Window That Allocates Shared Memory

MPI_WIN_ALLOCATE_SHARED(size, disp_unit, info, comm, baseptr, win) IN size of local window in bytes (non-negative integer) size IN disp_unit local unit size for displacements, in bytes (positive integer) IN info info argument (handle) IN intra-communicator (handle) comm OUT address of local allocated window segment (choice) baseptr OUT win window object returned by the call (handle)

C binding

int	MPI_Win_	_allocate_	shared	(MPI_A	lint	size,	int	disp.	_unit,	MPI_Info	info,
		MPI_Comr	n comm,	void	*ba	septr,	MPI.	_Win	*win)		

Fortran 2008 binding

MPI_Win_allocate_shared(size, disp_unit, info, comm, baseptr, win, ierror)
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
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
Fortran binding

MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR) INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR

This is a collective call executed by all processes in the group of comm. On each 40 41 process, it allocates memory of at least size bytes that is shared among all processes in 42comm, 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 4344target of load/store accesses by remote processes; the base pointers for other processes can be queried using the function MPI_WIN_SHARED_QUERY. The call also returns a 4546window object that can be used by all processes in comm to perform RMA operations. 47The size argument may be different at each process and size = 0 is valid. It is the user's 48 responsibility to ensure that the communicator comm represents a group of processes that

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5 6

11

12

13 14

15

16

17

18

19

20 21

22 23 24

37

38

1 can create a shared memory segment that can be accessed by all processes in the group. $\mathbf{2}$ The discussions of rationales for MPI_ALLOC_MEM and MPI_FREE_MEM in Section 8.2 3 also apply to MPI_WIN_ALLOCATE_SHARED; in particular, see the rationale in Section 8.2 4 for an explanation of the type used for **baseptr**. The allocated memory is contiguous across $\mathbf{5}$ process ranks unless the info key alloc_shared_noncontig is specified. Contiguous across process 6 ranks means that the first address in the memory segment of process i is consecutive with $\overline{7}$ the last address in the memory segment of process i-1. This may enable the user to 8 calculate remote address offsets with local information only.

⁹ If the Fortran compiler provides TYPE(C_PTR), then the following generic interface must
 ¹⁰ be provided in the mpi module and should be provided in mpif.h through overloading,
 ¹¹ i.e., with the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND)
 ¹² BASEPTR, but with a different specific procedure name:

14 INTERFACE MPI_WIN_ALLOCATE_SHARED

```
SUBROUTINE MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, &
15
                                              BASEPTR, WIN, IERROR)
16
             IMPORT :: MPI_ADDRESS_KIND
17
             INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
18
             INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
19
         END SUBROUTINE
20
         SUBROUTINE MPI_WIN_ALLOCATE_SHARED_CPTR(SIZE, DISP_UNIT, INFO, COMM, &
21
                                                   BASEPTR, WIN, IERROR)
22
                                 ISO_C_BINDING, ONLY : C_PTR
             USE, INTRINSIC ::
23
             IMPORT :: MPI_ADDRESS_KIND
24
             INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
25
             INTEGER(KIND=MPI_ADDRESS_KIND) ::
                                                  SIZE
26
             TYPE(C_PTR) :: BASEPTR
27
         END SUBROUTINE
28
     END INTERFACE
29
30
```

The base procedure name of this overloaded function is MPI_WIN_ALLOCATE_SHARED_CPTR. The implied specific procedure names are described in Section 18.1.5.

The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE, MPI_WIN_ALLOCATE, and MPI_ALLOC_MEM. The additional info key alloc_shared_noncontig allows the library to optimize the layout of the shared memory segments in memory.

Advice to users. If the info key alloc_shared_noncontig is not set to true, the allocation strategy is to allocate contiguous memory across process ranks. This may limit the performance on some architectures because it does not allow the implementation to modify the data layout (e.g., padding to reduce access latency). (End of advice to users.)

Advice to implementors. If the user sets the info key alloc_shared_noncontig to true,
 the implementation can allocate the memory requested by each process in a location
 that is close to this process. This can be achieved by padding or allocating memory
 in special memory segments. Both techniques may make the address space across
 consecutive ranks noncontiguous. (End of advice to implementors.)

Unofficial Draft for Comment Only

13

34

35

36

37 38

39

40

41

42

For contiguous shared memory allocations, the default alignment requirements outlined for MPI_ALLOC_MEM in Section 8.2 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 11.4) by utilizing the window synchronization functions (see Section 11.5) or explicitly completing outstanding store accesses (e.g., by calling MPI_WIN_FLUSH). MPI does not define semantics for accessing shared memory windows in the *separate memory model*.

21 22

23

MPI_VVIN_SHARED_QUERY(WIN, rank, size, disp_unit, baseptr)							
IN	rank	rank in the group of window win or	26				
		MPI_PROC_NULL (non-negative integer)	27				
OUT	size	size of the window segment (non-negative integer)	28				
OUT	disp_unit	local unit size for displacements, in bytes (positive	29				
		integer)	30				
OUT	hacontr	- ,	31				
001	baseptr	address for load/store access to window segment	32				
		(choice)	33				
			34				
C bindin	g		35				
int MPI_Win_shared_query(MPI_Win win, int rank, MPI_Aint *size,							
	int *disp_unit, void	*baseptr)	37				
Fortran	2008 binding		38				
	e e e e e e e e e e e e e e e e e e e	ize, disp_unit, baseptr, ierror)	39				
	- •		40				
	INTRINSIC :: ISO_C_BINDI		41				
	(MPI_Win), INTENT(IN) :: N	MTU	42				
	GER, INTENT(IN) :: rank		43				
	GER(KIND=MPI_ADDRESS_KIND)	-	44				
	GER, INTENT(OUT) :: disp_u		45				
	(C_PTR), INTENT(OUT) :: ba	-	46				
TNLE	GER, OPTIONAL, INTENT(OUT)) :: lerror	47				
Fortran binding							

MPI_WIN_SHARED_QUERY(win, rank, size, disp_unit, baseptr)

Fortran binding

1

 $\mathbf{2}$

3

4

5 6

7

8

9

10

11

12

13

14 15

16

17

18

19

CHAPTER 11. ONE-SIDED COMMUNICATIONS

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 ad-
dresses for the same physical memory on different processes. The returned memory can be
used for load/store accesses subject to the constraints defined in Section 11.7. This function
can only be called with windows of flavor MPI_WIN_FLAVOR_SHARED. If the passed window
is not of flavor MPI_WIN_FLAVOR_SHARED, the error MPI_ERR_RMA_FLAVOR is raised. When
rank is MPI_PROC_NULL, the pointer, disp_unit, and size returned are the pointer, disp_unit,
and size of the memory segment belonging the lowest rank that specified size > 0 . If all
processes in the group attached to the window specified size $= 0$, then the call returns size
$= 0$ and a baseptr as if MPI_ALLOC_MEM was called with size $= 0$.
If the Fortran compiler provides TYPE(C_PTR), then the following generic interface must
be provided in the mpi module and should be provided in mpif.h through overloading,
i.e., with the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND)
BASEPTR, but with a different specific procedure name:
INTERFACE MPI_WIN_SHARED_QUERY
SUBROUTINE MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, &
BASEPTR, IERROR)
IMPORT :: MPI_ADDRESS_KIND
INTEGER WIN, RANK, DISP_UNIT, IERROR
INTEGER (KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
END SUBROUTINE
SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, &
BASEPTR, IERROR)
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
IMPORT :: MPI_ADDRESS_KIND
INTEGER :: WIN, RANK, DISP_UNIT, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
TYPE(C_PTR) :: BASEPTR
END SUBROUTINE
END INTERFACE
The base procedure name of this overloaded function is
MPI_WIN_SHARED_QUERY_CPTR. The implied specific procedure names are described in
Section 18.1.5.

39 40

41

11.2.4 Window of Dynamically Attached Memory

The MPI-2 RMA model requires the user to identify the local memory that may be a target of RMA calls at the time the window is created. This has advantages for both the programmer (only this memory can be updated by one-sided operations and provides greater safety) and the MPI implementation (special steps may be taken to make onesided access to such memory more efficient). However, consider implementing a modifiable linked list using RMA operations; as new items are added to the list, memory must be allocated. In a C or C++ program, this memory is typically allocated using malloc or

new respectively. In MPI-2 RMA, the programmer must create a window with a predefined amount of memory and then implement routines for allocating memory from within the window's memory. In addition, there is no easy way to handle the situation where the predefined amount of memory turns out to be inadequate. To support this model, the routine MPI_WIN_CREATE_DYNAMIC creates a window that makes it possible to expose memory without remote synchronization. It must be used in combination with the local routines MPI_WIN_ATTACH and MPI_WIN_DETACH.

MPI_WIN_CREATE_DYNAMIC(info, comm, win)

IN	info	info argument (handle)
IN	comm	intra-communicator (handle)
OUT	win	window object returned by the call (handle)

C binding

int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win)

Fortran 2008 binding

MPI_Win_create_dynamic(info, comm, win, ierror)
 TYPE(MPI_Info), INTENT(IN) :: info
 TYPE(MPI_Comm), INTENT(IN) :: comm
 TYPE(MPI_Win), INTENT(OUT) :: win
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

MPI_WIN_CREATE_DYNAMIC(INFO, COMM, WIN, IERROR) INTEGER INFO, COMM, WIN, IERROR

This is a collective call executed by all processes in the group of comm. It returns a window win without memory attached. Existing process memory can be attached as described below. This routine returns a window object that can be used by these processes to perform RMA operations on attached memory. Because this window has special properties, it will sometimes be referred to as a *dynamic* window.

The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE.

In the case of a window created with MPI_WIN_CREATE_DYNAMIC, the target_disp for all RMA functions is the address at the target; i.e., the effective window_base is MPI_BOTTOM and the disp_unit is one. For dynamic windows, the target_disp argument to RMA communication operations is not restricted to non-negative values. Users should use MPI_GET_ADDRESS at the target process to determine the address of a target memory location and communicate this address to the origin process.

Advice to users. Users are cautioned that displacement arithmetic can overflow in variables of type MPI_Aint and result in unexpected values on some platforms. The MPI_AINT_ADD and MPI_AINT_DIFF functions can be used to safely perform address arithmetic with MPI_Aint displacements. (*End of advice to users.*)

Advice to implementors. In environments with heterogeneous data representations, care must be exercised in communicating addresses between processes. For example,

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5 6

7

8 9

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

	492	CI	HAPTER 11.	ONE-SIDED COMMUNICATIONS
1 2 3 4 5 6	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. (<i>End of advice to implementors.</i>)			
7 8 9 10 11 12	been attac MPI_WIN_ MPI_WIN_	ched using the function MPI_V _CREATE_DYNAMIC to creat	WIN_ATTACH e an MPI wir mory may be	the target of an MPI RMA operation.
13 14	MPI_WIN_	_ATTACH(win, base, size)		
15	IN	win	window obje	ct (handle)
16 17	IN	base	initial addres	s of memory to be attached (choice)
18 19	IN	size	size of memo integer)	ry to be attached in bytes (non-negative
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	Fortran 2 MPI_Win_a TYPE(TYPE) INTEG Fortran b MPI_WIN_A INTEG Attack window. T to the win window is MPI_WIN_ of size byte In Fortran must be 's non-overla Ratio one-s	Win_attach(MPI_Win win, vo 2008 binding attach(win, base, size, io (MPI_Win), INTENT(IN) :: vo (*), DIMENSION(), ASYNCH GER(KIND=MPI_ADDRESS_KIND) GER, OPTIONAL, INTENT(OUT) oinding ATTACH(WIN, BASE, SIZE, IN GER WIN, IERROR => BASE(*) GER(KIND=MPI_ADDRESS_KIND) hes a local memory region be The memory region specified r ndow win, that is, attaching of erroneous. The argument wi _CREATE_DYNAMIC. The loc es, starting at address base. In n, one can pass the first elem simply contiguous' (for 'simply opping) memory regions may b onale. Requiring that mem sided access by other processes	error) win HRONOUS :: E), INTENT(IN) :: ierror ERROR)) SIZE ginning at bas nust not conta overlapping m n must be a v cal memory re C, base is the nent of a mem y contiguous, ⁷ be attached to cory be explice s can simplify	base I) :: size se for remote access within the given ain any part that is already attached emory concurrently within the same window that was created with gion attached to the window consists e starting address of a memory region. mory region or a whole array, which ' see Section 18.1.12). Multiple (but the same window. itly attached before it is exposed to implementations and improve perfor-
47 48				r RMA operations without requiring a

collective MPI_WIN_CREATE call is needed for some one-sided programming models. (*End of rationale.*)

Advice to users. Attaching memory to a window may require the use of scarce resources; thus, attaching large regions of memory is not recommended in portable programs. Attaching memory to a window may fail if sufficient resources are not available; this is similar to the behavior of MPI_ALLOC_MEM.

The user is also responsible for ensuring that MPI_WIN_ATTACH at the target has returned before a process attempts to target that memory with an MPI RMA call.

Performing an RMA operation to memory that has not been attached to a window created with MPI_WIN_CREATE_DYNAMIC is erroneous. (*End of advice to users.*)

Advice to implementors. A high-quality implementation will attempt to make as much memory available for attaching as possible. Any limitations should be documented by the implementor. (*End of advice to implementors.*)

Attaching memory is a local operation as defined by MPI, which means that the call is not collective and completes without requiring any MPI routine to be called in any other process. Memory may be detached with the routine MPI_WIN_DETACH. After memory has been detached, it may not be the target of an MPI RMA operation on that window (unless the memory is re-attached with MPI_WIN_ATTACH).

MPI_WI	N_DETACH(win, base)		23
IN	win	window object (handle)	24
IN	base	initial address of memory to be detached (choice)	25
			26
			27

C binding

int MPI_Win_detach(MPI_Win win, const void *base)

Fortran 2008 binding

```
MPI_Win_detach(win, base, ierror)
   TYPE(MPI_Win), INTENT(IN) :: win
   TYPE(*), DIMENSION(..), ASYNCHRONOUS :: base
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

MPI_WIN_DETACH(WIN, BASE, IERROR)
 INTEGER WIN, IERROR
 <type> BASE(*)

Detaches a previously attached memory region beginning at base. The arguments base and win must match the arguments passed to a previous call to MPI_WIN_ATTACH.

Advice to users. Detaching memory may permit the implementation to make more efficient use of special memory or provide memory that may be needed by a subsequent MPI_WIN_ATTACH. Users are encouraged to detach memory that is no longer needed. Memory should be detached before it is freed by the user. (End of advice to users.)

Memory becomes detached when the associated dynamic memory window is freed, see Section 11.2.5.

Unofficial Draft for Comment Only

 $46 \\ 47$

1 2 3	11.2.5 Window Destruction	
4 5 6 7	MPI_WIN_FREE(win) INOUT win w	rindow object (handle)
8 9	C binding int MPI_Win_free(MPI_Win *win)	
10 11 12 13 14	Fortran 2008 binding MPI_Win_free(win, ierror) TYPE(MPI_Win), INTENT(INOUT) :: INTEGER, OPTIONAL, INTENT(OUT) :	
15 16 17	Fortran binding MPI_WIN_FREE(WIN, IERROR) INTEGER WIN, IERROR	
 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 	This is a collective call executed by all pro- MPI_WIN_FREE(win) can be invoked by a pro- in RMA communications on window win: MPI_WIN_FENCE, or called MPI_WIN_WA or called MPI_WIN_COMPLETE to match MPI_WIN_UNLOCK to match a previous ca- with windows created by a call to MPI_WIN_AL the window was created with MPI_WIN_AL memory that was allocated in MPI_WIN_AL MPI_WIN_ALLOCATE_SHARED, MPI_WIN_ allocated in MPI_WIN_ALLOCATE_SHARE Freeing a window that was created with	e.g., the process has called IT to match a previous call to MPI_WIN_POST a previous call to MPI_WIN_START or called all to MPI_WIN_LOCK. The memory associated I_CREATE may be freed after the call returns. If LOCATE, MPI_WIN_FREE will free the window ALLOCATE. If the window was created with N_FREE will free the window memory that was
34 35 36 37 38 39 40	cess can return from free until all pro- that no process will attempt to access it was freed. The only exception to key to true when creating the window	FREE requires a barrier synchronization: no pro- cesses in the group of win call free. This ensures is a remote window (e.g., with lock/unlock) after this rule is when the user sets the no_locks info . In that case, an MPI implementation may free chronization. (<i>End of advice to implementors.</i>)
41 42	11.2.6 Window Attributes	
43 44	The following attributes are cached with a	window when the window is created.
45 46 47 48	MPI_WIN_SIZE w MPI_WIN_DISP_UNIT d	vindow base address. vindow size, in bytes. isplacement unit associated with the window. ow the window was created.

MPI_WIN_MODEL

memory model for window. $\mathbf{2}$ In C, calls to MPI_Win_get_attr(win, MPI_WIN_BASE, &base, &flag), 3 MPI_Win_get_attr(win, MPI_WIN_SIZE, &size, &flag), 4 MPI_Win_get_attr(win, MPI_WIN_DISP_UNIT, &disp_unit, &flag), 5MPI_Win_get_attr(win, MPI_WIN_CREATE_FLAVOR, &create_kind, &flag), and 6 MPI_Win_get_attr(win, MPI_WIN_MODEL, & memory_model, & flag) will return in base a 7 pointer to the start of the window win, and will return in size, disp_unit, create_kind, and 8 memory_model pointers to the size, displacement unit of the window, the kind of routine 9 used to create the window, and the memory model, respectively. A detailed listing of the 10 type of the pointer in the attribute value argument to MPI_WIN_GET_ATTR and 11 MPI_WIN_SET_ATTR is shown in Table 11.1. 1213 Attribute C Type 14MPI_WIN_BASE void * 15MPI_WIN_SIZE MPI_Aint * 16MPI_WIN_DISP_UNIT int * 17 MPI_WIN_CREATE_FLAVOR int * 18 MPI_WIN_MODEL int * 19 2021Table 11.1: C types of attribute value argument to MPI_WIN_GET_ATTR and MPI_WIN_SET_ATTR. 22 23In Fortran, calls to MPI_WIN_GET_ATTR(win, MPI_WIN_BASE, base, flag, ierror), 24 MPI_WIN_GET_ATTR(win, MPI_WIN_SIZE, size, flag, ierror), 25MPI_WIN_GET_ATTR(win, MPI_WIN_DISP_UNIT, disp_unit, flag, ierror), 26MPI_WIN_GET_ATTR(win, MPI_WIN_CREATE_FLAVOR, create_kind, flag, ierror), and 27MPI_WIN_GET_ATTR(win, MPI_WIN_MODEL, memory_model, flag, ierror) will return in 28base, size, disp_unit, create_kind, and memory_model the (integer representation of) the 29base address, the size, the displacement unit of the window win, the kind of routine used to 30 create the window, and the memory model, respectively. 31The values of create_kind are 32 33 MPI_WIN_FLAVOR_CREATE Window was created with MPI_WIN_CREATE. 34 MPI_WIN_FLAVOR_ALLOCATE Window was created with MPI_WIN_ALLOCATE. 35

MPI_WIN_FLAVOR_DYNAMIC Window was created with 36 MPI_WIN_CREATE_DYNAMIC. 37 MPI_WIN_FLAVOR_SHARED Window was created with 38 MPI_WIN_ALLOCATE_SHARED. 39

The values of memory_model are MPI_WIN_SEPARATE and MPI_WIN_UNIFIED. The meaning of these is described in Section 11.4.

In the case of windows created with MPI_WIN_CREATE_DYNAMIC, the base address is MPI_BOTTOM and the size is 0. In C, pointers are returned, and in Fortran, the values are returned, for the respective attributes. (The window attribute access functions are defined in Section 6.7.3.) The value returned for an attribute on a window is constant over the lifetime of the window.

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

Unofficial Draft for Comment Only

40 41

42

4344

4546

47

```
1
     MPI_WIN_GET_GROUP(win, group)
2
       IN
                                              window object (handle)
                 win
3
       OUT
                                              group of processes which share access to the window
                 group
4
                                              (handle)
5
6
7
      C binding
8
      int MPI_Win_get_group(MPI_Win win, MPI_Group *group)
9
     Fortran 2008 binding
10
     MPI_Win_get_group(win, group, ierror)
11
          TYPE(MPI_Win), INTENT(IN) :: win
12
          TYPE(MPI_Group), INTENT(OUT) :: group
13
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
      Fortran binding
16
      MPI_WIN_GET_GROUP(WIN, GROUP, IERROR)
17
          INTEGER WIN, GROUP, IERROR
18
          MPI_WIN_GET_GROUP returns a duplicate of the group of the communicator used to
19
      create the window associated with win. The group is returned in group.
20
21
      11.2.7 Window Info
22
23
      Hints specified via info (see Section 9) allow a user to provide information to direct opti-
^{24}
      mization. Providing hints may enable an implementation to deliver increased performance
25
      or use system resources more efficiently. An implementation is free to ignore all hints;
26
      however, applications must comply with any info hints they provide that are used by the
27
      MPI implementation (i.e., are returned by a call to MPI_WIN_GET_INFO) and that place
28
      a restriction on the behavior of the application. Hints are specified on a per window basis,
29
      in window creation functions and MPI_WIN_SET_INFO, via the opaque info object. When
30
      an info object that specifies a subset of valid hints is passed to MPI_WIN_SET_INFO there
^{31}
      will be no effect on previously set or default hints that the info does not specify.
32
33
           Advice to implementors. It may happen that a program is coded with hints for one
34
           system, and later executes on another system that does not support these hints. In
35
           general, unsupported hints should simply be ignored. Needless to say, no hint can be
36
           mandatory. However, for each hint used by a specific implementation, a default value
37
           must be provided when the user does not specify a value for the hint. (End of advice
38
           to implementors.)
39
40
41
      MPI_WIN_SET_INFO(win, info)
42
43
       INOUT
                 win
                                              window object (handle)
44
       IN
                 info
                                              info argument (handle)
45
46
      C binding
47
      int MPI_Win_set_info(MPI_Win win, MPI_Info info)
48
```

Fortran 2008 binding

MPI_Win_set_info(win, info, ierror)				
TYPE(MPI_Win), INTENT(IN) :: win				
TYPE(MPI_Info), INTENT(IN) :: info				
INTEGER, OPTIONAL, INTENT(OUT) :: ierror				

Fortran binding

MPI_WIN_SET_INFO(WIN, INFO, IERROR) INTEGER WIN, INFO, IERROR

MPI_WIN_SET_INFO updates the hints of the window associated with win using the hints provided in info. This operation has no effect on previously set or defaulted hints that are not specified by info. It also has no effect on previously set or defaulted hints that are specified by info, but are ignored by the MPI implementation in this call to MPI_WIN_SET_INFO. The call is collective on the group of win. The info object may be different on each process, but any info entries that an implementation requires to be the same on all processes must appear with the same value in each process's info object.

Advice to users. Some info items that an implementation can use when it creates a window cannot easily be changed once the window has been created. Thus, an implementation may ignore hints issued in this call that it would have accepted in a creation call. An implementation may also be unable to update certain info hints in a call to MPI_WIN_SET_INFO. MPI_WIN_GET_INFO can be used to determine whether info changes were ignored by the implementation. (*End of advice to users.*)

MPI_WIN_GET_INFO(win, info_used)

			21
IN	win	window object (handle)	28
OUT	info_used	new info object (handle)	29
			30
C bindin	a contraction of the second se		31
	Y		32
int MPI_V	Vin_get_info(MPI_Win win,	MPI_Info *info_used)	33
Fortran 2	2008 binding		34
MPI_Win_g	get_info(win, info_used, :	ierror)	35
TYPE	(MPI_Win), INTENT(IN) :: v	win	36
TYPE	(MPI_Info), INTENT(OUT) :	: info_used	37
INTEC	GER, OPTIONAL, INTENT(OUT)) :: ierror	38
			39
Fortran l	0		40
	GET_INFO(WIN, INFO_USED, I		41
TNLEC	GER WIN, INFO_USED, IERRON	.	42

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

 44

ignored by the implementation; and any additional hints that were set by the implementation. If no such hints exist, a handle to a newly created info object is returned that contains no key/value pair. The user is responsible for freeing info_used via MPI_INFO_FREE.

11.3 Communication Calls

MPI supports the following RMA communication calls: MPI_PUT and MPI_RPUT transfer data from the caller memory (origin) to the target memory; MPI_GET and MPI_RGET transfer data from the target memory to the caller memory; MPI_ACCUMULATE and MPI_RACCUMULATE update locations in the target memory, e.g., by adding to these locations values sent from the caller memory; MPI_GET_ACCUMULATE,

12MPI_RGET_ACCUMULATE, and MPI_FETCH_AND_OP perform atomic read-modify-write 13 and return the data before the accumulate operation; and MPI_COMPARE_AND_SWAP per-14forms a remote atomic compare and swap operation. These operations are *nonblocking*: the 15call initiates the transfer, but the transfer may continue after the call returns. The transfer 16is completed, at the origin or both the origin and the target, when a subsequent synchro-17nization call is issued by the caller on the involved window object. These synchronization 18 calls are described in Section 11.5. Transfers can also be completed with calls to flush rou-19tines; see Section 11.5.4 for details. For the MPI_RPUT, MPI_RGET, MPI_RACCUMULATE, 20and MPI_RGET_ACCUMULATE calls, the transfer can be locally completed by using the 21MPI test or wait operations described in Section 3.7.3. 22

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 26memory locations is undefined; if a location is updated by a put or accumulate operation, 27then the outcome of loads or other RMA operations is undefined until the updating operation 28has completed at the target. There is one exception to this rule; namely, the same location 29can be updated by several concurrent accumulate calls, the outcome being as if these updates 30 occurred in some order. In addition, the outcome of concurrent load/store and RMA updates 31 to the same memory location is undefined. These restrictions are described in more detail 32 in Section 11.7. 33

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

For all RMA calls, the target process may be identical with the origin process; i.e., a process may use an RMA operation to move data in its memory.

39 40 41

42

43

38

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.*)

⁴⁴ MPI_PROC_NULL is a valid target rank in all MPI RMA communication calls. The effect
 ⁴⁵ is the same as for MPI_PROC_NULL in MPI point-to-point communication. After any RMA
 ⁴⁶ operation with rank MPI_PROC_NULL, it is still necessary to finish the RMA epoch with the
 ⁴⁷ synchronization method that started the epoch.

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

 $\frac{4}{5}$

6 7

8

9

10

11.3.1 Put

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

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

IN	origin_addr	initial address of origin buffer (choice)	10
IN	origin_count	number of entries in origin buffer (non-negative integer)	11 12
IN	origin_datatype	datatype of each entry in origin buffer (handle)	$13 \\ 14$
IN	target_rank	rank of target (non-negative integer)	15
IN	target_disp	displacement from start of window to target buffer (non-negative integer)	16 17
IN	target_count	number of entries in target buffer (non-negative integer)	18 19
IN	target_datatype	datatype of each entry in target buffer (handle)	20 21
IN	win	window object used for communication (handle)	22
C binding	Ś		23 24

C hinding

C binding	
int MPI_Put(const void *origin_addr, int origin_count,	25
MPI_Datatype origin_datatype, int target_rank,	26
MPI_Aint target_disp, int target_count,	27
MPI_Datatype target_datatype, MPI_Win win)	28
Fortuge 2008 his diag	29
Fortran 2008 binding	30
MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,	31
<pre>target_disp, target_count, target_datatype, win, ierror)</pre>	32
<pre>TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr</pre>	33
INTEGER, INTENT(IN) :: origin_count, target_rank, target_count	34
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype	35
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp	36
TYPE(MPI_Win), INTENT(IN) :: win	37
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	38
Fortran binding	39
MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,	40
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)	41
<pre><type> ORIGIN_ADDR(*)</type></pre>	42
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,	43
TARGET_DATATYPE, WIN, IERROR	44
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	45
	46
Transfers origin_count successive entries of the type specified by the origin_datatype.	, 47

starting at address origin_addr on the origin node, to the target node specified by the win, 48

Unofficial Draft for Comment Only

1 $\mathbf{2}$

3

4

target_rank pair. The data are written in the target buffer at address $target_addr =$

- window_base+target_disp×disp_unit, where window_base and disp_unit are the base address
 and window displacement unit specified at window initialization, by the target process.
 - The target buffer is specified by the arguments target_count and target_datatype.
- The data transfer is the same as that which would occur if the origin process executed a send operation with arguments origin_addr, origin_count, origin_datatype, target_rank, tag, comm, and the target process executed a receive operation with arguments target_addr, target_count, target_datatype, source, tag, comm, where target_addr is the target buffer address computed as explained above, the values of tag are arbitrary valid matching tag values, and comm is a communicator for the group of win.
- The communication must satisfy the same constraints as for a similar message-passing communication. The target_datatype may not specify overlapping entries in the target buffer. The message sent must fit, without truncation, in the target buffer. Furthermore, the target buffer must fit in the target window or in attached memory in a dynamic window.
- ¹⁵ The target_datatype argument is a handle to a datatype object defined at the origin ¹⁶ process. However, this object is interpreted at the target process: the outcome is as if ¹⁷ the target datatype object was defined at the target process by the same sequence of calls ¹⁸ used to define it at the origin process. The target datatype must contain only relative ¹⁹ displacements, not absolute addresses. The same holds for get and accumulate operations.
 - Advice to users. The target_datatype argument is a handle to a datatype object that is defined at the origin process, even though it defines a data layout in the target process memory. This causes no problems in a homogeneous environment, or in a heterogeneous environment if only portable datatypes are used (portable datatypes are defined in Section 2.4).
 - The performance of a put transfer can be significantly affected, on some systems, by the choice of window location and the shape and location of the origin and target buffer: transfers to a target window in memory allocated by MPI_ALLOC_MEM or MPI_WIN_ALLOCATE may be much faster on shared memory systems; transfers from contiguous buffers will be faster on most, if not all, systems; the alignment of the communication buffers may also impact performance. (*End of advice to users.*)
 - Advice to implementors. A high-quality implementation will attempt to prevent remote accesses to memory outside the window that was exposed by the process. This is important both for debugging purposes and for protection with client-server codes that use RMA. That is, a high-quality implementation will check, if possible, window bounds on each RMA call, and raise an MPI exception at the origin call if an out-of-bound situation occurs. Note that the condition can be checked at the origin. Of course, the added safety achieved by such checks has to be weighed against the added cost of such checks. (*End of advice to implementors.*)
- 42 43
- 44
- 45
- 46
- 47 48

1

4

21

22

23

 24

25 26

27

28

29

30

31

32 33

34

35

36

37

38

39

40

11.3.2 Get MPI_GET(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win) OUT origin_addr initial address of origin buffer (choice) origin_count IN number of entries in origin buffer (non-negative integer) origin_datatype IN datatype of each entry in origin buffer (handle) target_rank IN rank of target (non-negative integer) IN target_disp displacement from window start to the beginning of the target buffer (non-negative integer) number of entries in target buffer (non-negative IN target_count integer) IN target_datatype datatype of each entry in target buffer (handle) IN win window object used for communication (handle) 21C binding 22 int MPI_Get(void *origin_addr, int origin_count, 23MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win) Fortran 2008 binding 27MPI_Get(origin_addr, origin_count, origin_datatype, target_rank, 28target_disp, target_count, target_datatype, win, ierror) 29 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: origin_addr INTEGER, INTENT(IN) :: origin_count, target_rank, target_count TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp TYPE(MPI_Win), INTENT(IN) :: win 34 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 3536 Fortran binding 37 MPI_GET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR) <type> ORIGIN_ADDR(*) INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR 42

INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP

Similar to MPI_PUT, except that the direction of data transfer is reversed. Data 44 are copied from the target memory to the origin. The origin_datatype may not specify 45overlapping entries in the origin buffer. The target buffer must be contained within the 46 target window or within attached memory in a dynamic window, and the copied data must 47fit, without truncation, in the origin buffer. 48

Unofficial Draft for Comment Only

5

6

7

8

9 10

11

12

13

1415

16

17

18

19 20

 24

25

26

30

31

32

33

38

39

40

41

```
1
     11.3.3 Examples for Communication Calls
\mathbf{2}
     These examples show the use of the MPI_GET function. As all MPI RMA communication
3
     functions are nonblocking, they must be completed. In the following, this is accomplished
4
     with the routine MPI_WIN_FENCE, introduced in Section 11.5.
5
6
     Example 11.1 We show how to implement the generic indirect assignment A = B(map),
7
     where A, B, and map have the same distribution, and map is a permutation. To simplify, we
8
     assume a block distribution with equal size blocks.
9
10
     SUBROUTINE MAPVALS(A, B, map, m, comm, p)
11
     USE MPI
12
     INTEGER m, map(m), comm, p
13
     REAL A(m), B(m)
14
15
     INTEGER otype(p), oindex(m),
                                       & ! used to construct origin datatypes
16
           ttype(p), tindex(m),
                                       & ! used to construct target datatypes
17
           count(p), total(p),
                                       &
18
           disp_int, win, ierr
19
     INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint
20
21
     ! This part does the work that depends on the locations of B.
22
     ! Can be reused while this does not change
23
^{24}
     CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
25
     disp_int = realextent
26
     size = m * realextent
27
     CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL,
                                                                   &
28
                           comm, win, ierr)
29
30
     ! This part does the work that depends on the value of map and
31
     ! the locations of the arrays.
32
     ! Can be reused while these do not change
33
34
     ! Compute number of entries to be received from each process
35
36
     DO i=1,p
37
        count(i) = 0
38
     END DO
39
     DO i=1,m
40
        j = map(i)/m+1
41
        count(j) = count(j)+1
42
     END DO
43
^{44}
     total(1) = 0
45
     DO i=2,p
46
        total(i) = total(i-1) + count(i-1)
47
     END DO
48
```

```
1
DO i=1,p
                                                                                       \mathbf{2}
   count(i) = 0
END DO
                                                                                       4
! compute origin and target indices of entries.
                                                                                       5
! entry i at current process is received from location
                                                                                       6
! k at process (j-1), where map(i) = (j-1)*m + (k-1),
! j = 1...p and k = 1...m
                                                                                       9
                                                                                       10
DO i=1,m
                                                                                       11
   j = map(i)/m+1
   k = MOD(map(i), m) + 1
                                                                                       12
   count(j) = count(j)+1
                                                                                       13
   oindex(total(j) + count(j)) = i
                                                                                       14
                                                                                       15
   tindex(total(j) + count(j)) = k
                                                                                       16
END DO
                                                                                       17
                                                                                       18
! create origin and target datatypes for each get operation
                                                                                       19
DO i=1,p
   CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, &
                                                                                       20
                                          oindex(total(i)+1:total(i)+count(i)), &
                                                                                      21
                                          MPI_REAL, otype(i), ierr)
                                                                                      22
                                                                                      23
   CALL MPI_TYPE_COMMIT(otype(i), ierr)
                                                                                      ^{24}
   CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, &
                                                                                      25
                                          tindex(total(i)+1:total(i)+count(i)), &
                                                                                      26
                                         MPI_REAL, ttype(i), ierr)
   CALL MPI_TYPE_COMMIT(ttype(i), ierr)
                                                                                      27
END DO
                                                                                      28
                                                                                      29
                                                                                      30
! this part does the assignment itself
                                                                                       31
CALL MPI_WIN_FENCE(0, win, ierr)
disp_aint = 0
                                                                                      32
                                                                                      33
DO i=1,p
                                                                                      34
   CALL MPI_GET(A, 1, otype(i), i-1, disp_aint, 1, ttype(i), win, ierr)
END DO
                                                                                      35
CALL MPI_WIN_FENCE(0, win, ierr)
                                                                                      36
                                                                                      37
                                                                                       38
CALL MPI_WIN_FREE(win, ierr)
                                                                                       39
DO i=1,p
   CALL MPI_TYPE_FREE(otype(i), ierr)
                                                                                       40
                                                                                       41
   CALL MPI_TYPE_FREE(ttype(i), ierr)
                                                                                      42
END DO
RETURN
                                                                                      43
                                                                                      44
END
                                                                                       45
                                                                                       46
```

Example 11.2 A simpler version can be written that does not require that a datatype be built for the target buffer. But, one then needs a separate get call for each entry, as

Unofficial Draft for Comment Only

47

```
1
     illustrated below. This code is much simpler, but usually much less efficient, for large arrays.
\mathbf{2}
3
     SUBROUTINE MAPVALS(A, B, map, m, comm, p)
4
     USE MPI
     INTEGER m, map(m), comm, p
\mathbf{5}
     REAL A(m), B(m)
6
     INTEGER disp_int, win, ierr
7
     INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint
8
9
10
     CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
11
     disp_int = realextent
     size = m * realextent
12
     CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL,
13
                                                                   &
                            comm, win, ierr)
14
15
16
     CALL MPI_WIN_FENCE(0, win, ierr)
17
     DO i=1,m
18
        j = map(i)/m
        disp_aint = MOD(map(i),m)
19
        CALL MPI_GET(A(i), 1, MPI_REAL, j, disp_aint, 1, MPI_REAL, win, ierr)
20
     END DO
21
     CALL MPI_WIN_FENCE(0, win, ierr)
22
     CALL MPI_WIN_FREE(win, ierr)
23
^{24}
     RETURN
     END
25
26
```

11.3.4 Accumulate Functions

27

28

It is often useful in a put operation to combine the data moved to the target process with the data that resides at that process, rather than replacing it. This will allow, for example, the accumulation of a sum by having all involved processes add their contributions to the sum variable in the memory of one process. The accumulate functions have slightly different semantics with respect to overlapping data accesses than the put and get functions; see Section 11.7 for details.

			2
			3 4
MPI_A	CCUMULATE(origin_addr, or target_count, target	rigin_count, origin_datatype, target_rank, target_disp,	5
		··· ,	6
IN	origin_addr	initial address of buffer (choice)	7
IN	origin_count	number of entries in buffer (non-negative integer)	8
IN	origin_datatype	datatype of each entry (handle)	9 10
IN	target_rank	rank of target (non-negative integer)	11
IN	target_disp	displacement from start of window to beginning of target buffer (non-negative integer)	12 13
IN	target_count	number of entries in target buffer (non-negative integer)	14 15
IN	target_datatype	datatype of each entry in target buffer (handle)	16 17
IN	ор	reduce operation (handle)	18
IN	win	window object (handle)	19
			20
C bind	ling		21 22
int MP	I_Accumulate(const void	<pre>*origin_addr, int origin_count,</pre>	22
		gin_datatype, int target_rank,	24
	-	disp, int target_count,	25
	MPI_Datatype tar	get_datatype, MPI_Op op, MPI_Win win)	26
Fortra	n 2008 binding		27
MPI_Ac		rigin_count, origin_datatype, target_rank,	28
		get_count, target_datatype, op, win, ierror)	29
		NTENT(IN), ASYNCHRONOUS :: origin_addr	30
		igin_count, target_rank, target_count	31
		T(IN) :: origin_datatype, target_datatype	32
		KIND), INTENT(IN) :: target_disp	33 34
	PE(MPI_Op), INTENT(IN) PE(MPI_Win), INTENT(IN)	-	35
	TEGER, OPTIONAL, INTENT		36
TI	ILGER, OF ITOWAL, INTENT		37
	n binding		38
MPI_AC		RIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,	39
		GET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)	40
	<pre>ype> ORIGIN_ADDR(*) TEGED ODIGIN COUNT ODI</pre>		41
ΤN		GIN_DATATYPE, TARGET_RANK, TARGET_COUNT, , OP, WIN, IERROR	42
TN	TEGER(KIND=MPI_ADDRESS_		43
			44
Ac	cumulate the contents of the	origin buffer (as defined by origin_addr, origin_count, and	45

Accumulate the contents of the origin buffer (as defined by origin_addr, origin_count, and origin_datatype) to the buffer specified by arguments target_count and target_datatype, at offset target_disp, in the target window specified by target_rank and win, using the operation

```
1
     op. This is like MPI_PUT except that data is combined into the target area instead of
\mathbf{2}
     overwriting it.
3
          Any of the predefined operations for MPI_REDUCE can be used. User-defined functions
4
     cannot be used. For example, if op is MPI_SUM, each element of the origin buffer is added
\mathbf{5}
     to the corresponding element in the target, replacing the former value in the target.
6
          Each datatype argument must be a predefined datatype or a derived datatype, where
7
     all basic components are of the same predefined datatype. Both datatype arguments must
8
     be constructed from the same predefined datatype. The operation op applies to elements of
9
     that predefined type. The parameter target_datatype must not specify overlapping entries,
10
     and the target buffer must fit in the target window.
11
          A new predefined operation, MPI_REPLACE, is defined. It corresponds to the associative
12
     function f(a,b) = b; i.e., the current value in the target memory is replaced by the value
13
     supplied by the origin.
14
          MPI_REPLACE can be used only in MPI_ACCUMULATE, MPI_RACCUMULATE,
15
     MPI_GET_ACCUMULATE, MPI_FETCH_AND_OP, and MPI_RGET_ACCUMULATE, but not
16
     in collective reduction operations such as MPI_REDUCE.
17
                             MPI_PUT is a special case of MPI_ACCUMULATE, with the op-
           Advice to users.
18
           eration MPI_REPLACE. Note, however, that MPI_PUT and MPI_ACCUMULATE have
19
           different constraints on concurrent updates. (End of advice to users.)
20
21
22
     Example 11.3 We want to compute B(j) = \sum_{map(i)=j} A(i). The arrays A, B, and map
23
     are distributed in the same manner. We write the simple version.
24
25
     SUBROUTINE SUM(A, B, map, m, comm, p)
26
     USE MPI
27
     INTEGER m, map(m), comm, p, win, ierr, disp_int
28
     REAL A(m), B(m)
29
     INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint
30
^{31}
     CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
32
     size = m * realextent
33
     disp_int = realextent
34
     CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL, &
35
                            comm, win, ierr)
36
37
     CALL MPI_WIN_FENCE(0, win, ierr)
38
     DO i=1,m
39
        j = map(i)/m
40
        disp_aint = MOD(map(i),m)
41
        CALL MPI_ACCUMULATE(A(i), 1, MPI_REAL, j, disp_aint, 1, MPI_REAL,
                                                                                      &
42
                               MPI_SUM, win, ierr)
43
     END DO
^{44}
     CALL MPI_WIN_FENCE(0, win, ierr)
45
46
     CALL MPI_WIN_FREE(win, ierr)
47
     RETURN
48
     END
```

 $46 \\ 47$

This code is identical to the code in Example 11.2, except that a call to get has been replaced by a call to accumulate. (Note that, if map is one-to-one, the code computes $B = A(map^{-1})$, which is the reverse assignment to the one computed in that previous example.) In a similar manner, we can replace in Example 11.1, the call to get by a call to accumulate, thus performing the computation with only one communication between any two processes.

Get Accumulate Function

It is often useful to have fetch-and-accumulate semantics such that the remote data is returned to the caller before the sent data is accumulated into the remote data. The get and accumulate steps are executed atomically for each basic element in the datatype (see Section 11.7 for details). The predefined operation MPI_REPLACE provides fetch-and-set behavior.

MPI_GET_ACCUMULATE(origin_addr, origin_count, origin_datatype, result_addr,				
	()	type, target_rank, target_disp, target_count,	17	
	target_datatype, op, win		18	
			19	
IN	origin_addr	initial address of buffer (choice)	20	
IN	origin_count	number of entries in origin buffer (non-negative	21	
		integer)	22	
IN	origin_datatype	datatype of each entry in origin buffer (handle)	23	
OUT	result_addr	initial address of result buffer (choice)	24	
			25	
IN	result_count	number of entries in result buffer (non-negative	26	
		integer)	27	
IN	result_datatype	datatype of each entry in result buffer (handle)	28 29	
IN	target_rank	rank of target (non-negative integer)	30	
IN	target_disp	displacement from start of window to beginning of	31	
		target buffer (non-negative integer)	32	
IN	target count		33	
IN	target_count	number of entries in target buffer (non-negative integer)	34	
		<i>o</i> ,	35	
IN	target_datatype	datatype of each entry in target buffer (handle)	36	
IN	ор	reduce operation (handle)	37	
IN	win	window object (handle)	38	
			39	
C bindin	σ		40	
	0	*origin_addr, int origin_count,	41	
MPI Datatype origin datatype woid *result addr				

<pre>int result_count, MPI_Datatype result_datatype,</pre>
<pre>int target_rank, MPI_Aint target_disp, int target_count,</pre>
MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)

Fortran 2008 binding

Unofficial Draft for Comment Only

```
1
     MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
\mathbf{2}
                    result_count, result_datatype, target_rank, target_disp,
3
                    target_count, target_datatype, op, win, ierror)
4
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
5
         INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
6
                     target_count
7
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
8
                     target_datatype
9
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
10
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
11
         TYPE(MPI_Op), INTENT(IN) :: op
12
         TYPE(MPI_Win), INTENT(IN) :: win
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
     Fortran binding
15
     MPI_GET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_ADDR,
16
                    RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, TARGET_DISP,
17
                    TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
18
          <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
19
         INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
20
                     TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR
21
         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
22
23
         Accumulate origin_count elements of type origin_datatype from the origin buffer (
^{24}
     origin_addr) to the buffer at offset target_disp, in the target window specified by target_rank
25
     and win, using the operation op and return in the result buffer result_addr the content
26
     of the target buffer before the accumulation, specified by target_disp, target_count, and
27
     target_datatype. The data transferred from origin to target must fit, without truncation,
28
     in the target buffer. Likewise, the data copied from target to origin must fit, without
29
     truncation, in the result buffer.
30
         The origin and result buffers (origin_addr and result_addr) must be disjoint. Each
^{31}
     datatype argument must be a predefined datatype or a derived datatype where all basic
32
     components are of the same predefined datatype. All datatype arguments must be con-
33
     structed from the same predefined datatype. The operation op applies to elements of that
34
     predefined type. target_datatype must not specify overlapping entries, and the target buffer
```

³⁵ must fit in the target window or in attached memory in a dynamic window. The operation ³⁶ is executed atomically for each basic datatype; see Section 11.7 for details.

37 Any of the predefined operations for MPI_REDUCE, as well as MPI_NO_OP or 38 MPI_REPLACE can be specified as op. User-defined functions cannot be used. A new 39 predefined operation, MPI_NO_OP, is defined. It corresponds to the associative function 40f(a,b) = a; i.e., the current value in the target memory is returned in the result buffer at 41 the origin and no operation is performed on the target buffer. When MPI_NO_OP is specified 42as the operation, the origin_addr, origin_count, and origin_datatype arguments are ignored. 43MPI_NO_OP can be used only in MPI_GET_ACCUMULATE, MPI_RGET_ACCUMULATE, 44and MPI_FETCH_AND_OP. MPI_NO_OP cannot be used in MPI_ACCUMULATE, 45MPI_RACCUMULATE, or collective reduction operations, such as MPI_REDUCE and others. 46

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. (End of advice to users.)

Fetch and Op Function

The generic functionality of MPI_GET_ACCUMULATE might limit the performance of fetchand-increment or fetch-and-add calls that might be supported by special hardware operations. MPI_FETCH_AND_OP thus allows for a fast implementation of a commonly used subset of the functionality of MPI_GET_ACCUMULATE.

MPI_FETCH_AND_OP(origin_addr, result_addr, datatype, target_rank, target_disp, op, win)

IN	origin_addr	initial address of buffer (choice)
OUT	result_addr	initial address of result buffer (choice)
IN	datatype	datatype of the entry in origin, result, and target buffers (handle)
IN	target_rank	rank of target (non-negative integer)
IN	target_disp	displacement from start of window to beginning of target buffer (non-negative integer)
IN	ор	reduce operation (handle)
IN	win	window object (handle)
C bindi	ng	

C binding

<pre>int MPI_Fetch_and_op(const void *origin_addr, void *result_addr,</pre>	26
MPI_Datatype datatype, int target_rank, MPI_Aint target_disp,	27
MPI_Op op, MPI_Win win)	28
Fortran 2008 binding	29
	30
MPI_Fetch_and_op(origin_addr, result_addr, datatype, target_rank,	31
target_disp, op, win, ierror)	32
<pre>TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr</pre>	33
TYPE(*), DIMENSION(), ASYNCHRONOUS :: result_addr	34
TYPE(MPI_Datatype), INTENT(IN) :: datatype	35
INTEGER, INTENT(IN) :: target_rank	36
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp	
TYPE(MPI_Op), INTENT(IN) :: op	37
TYPE(MPI_Win), INTENT(IN) :: win	38
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	39
	40
Fortran binding	41
MPI_FETCH_AND_OP(ORIGIN_ADDR, RESULT_ADDR, DATATYPE, TARGET_RANK,	42
TARGET_DISP, OP, WIN, IERROR)	43
<type> ORIGIN_ADDR(*), RESULT_ADDR(*)</type>	44
INTEGER DATATYPE, TARGET_RANK, OP, WIN, IERROR	45
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	46
	47
	48

 $\mathbf{2}$

Accumulate one element of type datatype from the origin buffer (origin_addr) to the buffer at offset target_disp, in the target window specified by target_rank and win, using the operation op and return in the result buffer result_addr the content of the target buffer before the accumulation.

⁵ The origin and result buffers (origin_addr and result_addr) must be disjoint. Any of the ⁶ predefined operations for MPI_REDUCE, as well as MPI_NO_OP or MPI_REPLACE, can be ⁷ specified as op; user-defined functions cannot be used. The datatype argument must be a ⁸ predefined datatype. The operation is executed atomically.

¹⁰ Compare and Swap Function

Another useful operation is an atomic compare and swap where the value at the origin is compared to the value at the target, which is atomically replaced by a third value only if the values at origin and target are equal.

```
16
     MPI_COMPARE_AND_SWAP(origin_addr, compare_addr, result_addr, datatype,
17
                     target_rank, target_disp, win)
18
       IN
                 origin_addr
                                              initial address of buffer (choice)
19
20
       IN
                 compare_addr
                                              initial address of compare buffer (choice)
21
                                             initial address of result buffer (choice)
       OUT
                 result_addr
22
                                              datatype of the element in all buffers (handle)
       IN
                 datatype
23
^{24}
       IN
                 target_rank
                                              rank of target (non-negative integer)
25
                 target_disp
                                              displacement from start of window to beginning of
       IN
26
                                              target buffer (non-negative integer)
27
                                              window object (handle)
       IN
                 win
28
29
     C binding
30
     int MPI_Compare_and_swap(const void *origin_addr, const void *compare_addr,
^{31}
                     void *result_addr, MPI_Datatype datatype, int target_rank,
32
                     MPI_Aint target_disp, MPI_Win win)
33
34
     Fortran 2008 binding
35
     MPI_Compare_and_swap(origin_addr, compare_addr, result_addr, datatype,
36
                     target_rank, target_disp, win, ierror)
37
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr,
38
                      compare_addr
39
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
40
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
41
          INTEGER, INTENT(IN) :: target_rank
42
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
43
          TYPE(MPI_Win), INTENT(IN) :: win
44
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
```

```
    Fortran binding
    MPI_COMPARE_AND_SWAP(ORIGIN_ADDR, COMPARE_ADDR, RESULT_ADDR, DATATYPE,
    TARGET_RANK, TARGET_DISP, WIN, IERROR)
```

9

11

<type> ORIGIN_ADDR(*), COMPARE_ADDR(*), RESULT_ADDR(*) INTEGER DATATYPE, TARGET_RANK, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP

This function compares one element of type datatype in the compare buffer compare_addr with the buffer at offset target_disp in the target window specified by target_rank and win and replaces the value at the target with the value in the origin buffer origin_addr if the compare buffer and the target buffer are identical. The original value at the target is returned in the buffer result_addr. The parameter datatype must belong to one of the following categories of predefined datatypes: C integer, Fortran integer, Logical, Multi-language types, or Byte as specified in Section 5.9.2. The origin and result buffers (origin_addr and result_addr) must be disjoint.

11.3.5 Request-based RMA Communication Operations

Request-based RMA communication operations allow the user to associate a request handle with the RMA operations and test or wait for the completion of these requests using the functions described in Section 3.7.3. Request-based RMA operations are only valid within a passive target epoch (see Section 11.5).

Upon returning from a completion call in which an RMA operation completes, all fields of the status object, if any, and the results of status query functions (e.g.,

MPI_GET_COUNT) are undefined with the exception of MPI_ERROR if appropriate (see Section 3.2.5). It is valid to mix different request types (e.g., any combination of RMA requests, collective requests, I/O requests, generalized requests, or point-to-point requests) in functions that enable multiple completions (e.g., MPI_WAITALL). It is erroneous to call MPI_REQUEST_FREE or MPI_CANCEL for a request associated with an RMA operation. RMA requests are not persistent.

The end of the epoch, or explicit bulk synchronization using MPI_WIN_FLUSH, MPI_WIN_FLUSH_ALL, MPI_WIN_FLUSH_LOCAL, or MPI_WIN_FLUSH_LOCAL_ALL, also indicates completion of the RMA operations. However, users must still wait or test on the request handle to allow the MPI implementation to clean up any resources associated with these requests; in such cases the wait operation will complete locally. $\mathbf{2}$

 24

12	MPI_RPU	T(origin_addr, origin_count, or target_count, target_dat	igin_datatype, target_rank, target_disp, atype, win, request)	
$\frac{3}{4}$	IN	origin_addr	initial address of origin buffer (choice)	
5	IN	origin_count	number of entries in origin buffer (non-negative integer)	
7	IN	origin_datatype	datatype of each entry in origin buffer (handle)	
8 9	IN	target_rank	rank of target (non-negative integer)	
9 10 11	IN	target_disp	displacement from start of window to target buffer (non-negative integer)	
12 13	IN	target_count	number of entries in target buffer (non-negative integer)	
14 15	IN	target_datatype	datatype of each entry in target buffer (handle)	
16	IN	win	window object used for communication (handle)	
17	OUT	request	RMA request (handle)	
18				
19 20	C bindin	5	the set of set	
21	int MPI_	Rput(const void *origin_a MPI Datatype origin	dar, int origin_count, _datatype, int target_rank,	
22		MPI_Aint target_disp		
23			_datatype, MPI_Win win,	
24 25		MPI_Request *request	5)	
26	Fortran 2008 binding			
27	<pre>MPI_Rput(origin_addr, origin_count, origin_datatype, target_rank,</pre>			
28			_count, target_datatype, win, request,	
29	<pre>ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr</pre>			
30 31	INTEGER, INTENT(IN) :: origin_count, target_rank, target_count			
32	TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype			
33	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp			
34	TYPE(MPI_Win), INTENT(IN) :: win			
35		(MPI_Request), INTENT(OUT	-	
36	LNIE	GER, OPTIONAL, INTENT(OUT) :: lerror	
37 38	Fortran binding			
39	MPI_RPUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,			
40	TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST,			
41	IERROR) <type> ORIGIN_ADDR(*)</type>			
42	INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,			
43		TARGET_DATATYPE, WI	N, REQUEST, IERROR	
44 45	INTE	GER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	
46	MPI_	RPUT is similar to MPI_PUT	(Section 11.3.1), except that it allocates a commu-	
47	nication r	equest object and associates	it with the request handle (the argument request).	
48				

The completion of an MPI_RPUT operation (i.e., after the corresponding test or wait) indicates that the sender is now free to update the locations in the origin buffer. It does not indicate that the data is available at the target window. If remote completion is required, MPI_WIN_FLUSH, MPI_WIN_FLUSH_ALL, MPI_WIN_UNLOCK, or MPI_WIN_UNLOCK_ALL can be used.

MPI_RGET(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win, request)

OUT	origin_addr	initial address of origin buffer (choice)	10
IN	origin_count	number of entries in origin buffer (non-negative integer)	11 12
IN	origin_datatype	datatype of each entry in origin buffer (handle)	$13 \\ 14$
IN	target_rank	rank of target (non-negative integer)	15
IN	target_disp	displacement from window start to the beginning of the target buffer (non-negative integer)	16 17
IN	target_count	number of entries in target buffer (non-negative integer)	18 19 20
IN	target_datatype	datatype of each entry in target buffer (handle)	21
IN	win	window object used for communication (handle)	22
OUT	request	RMA request (handle)	23 24

C binding

Colliding	20
<pre>int MPI_Rget(void *origin_addr, int origin_count,</pre>	27
MPI_Datatype origin_datatype, int target_rank,	28
MPI_Aint target_disp, int target_count,	29
MPI_Datatype target_datatype, MPI_Win win,	30
MPI_Request *request)	31
Fortran 2008 binding	32
	33
<pre>MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank,</pre>	34
<pre>target_disp, target_count, target_datatype, win, request,</pre>	35
ierror)	36
TYPE(*), DIMENSION(), ASYNCHRONOUS :: origin_addr	37
INTEGER, INTENT(IN) :: origin_count, target_rank, target_count	38
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype	39
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp	40
TYPE(MPI_Win), INTENT(IN) :: win	41
TYPE(MPI_Request), INTENT(OUT) :: request	42
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	43
Fortran binding	44
5	45
MPI_RGET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,	
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST,	46
IERROR)	47
<type> ORIGIN_ADDR(*)</type>	48

1 $\mathbf{2}$

3

4

56 7

8

9

25

	514	CH	HAPTER 11. ONE-SIDED COMMUNICATIONS	
1 2 3	INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP			
4 5 7 8 9 10	MPI_RGET is similar to MPI_GET (Section 11.3.2), except that it allocates a commu- nication 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.			
11 12	MPI_RACC	CUMULATE(origin_addr, origin_ target_count, target_data	_count, origin_datatype, target_rank, target_disp, atype, op, win, request)	
$13 \\ 14$	IN	origin_addr	initial address of buffer (choice)	
15	IN	origin_count	number of entries in buffer (non-negative integer)	
16	IN	origin_datatype	datatype of each entry in origin buffer (handle)	
17 18	IN	target_rank	rank of target (non-negative integer)	
19 20	IN	target_disp	displacement from start of window to beginning of target buffer (non-negative integer)	
21 22	IN	target_count	number of entries in target buffer (non-negative integer)	
23 24	IN	target_datatype	datatype of each entry in target buffer (handle)	
25	IN	ор	reduce operation (handle)	
26	IN	win	window object (handle)	
27 28	OUT	request	RMA request (handle)	
29 30 31 32 33 34 35	C binding int MPI_Raccumulate(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,			
36	Fortran 2	008 binding		
37 38		mulate(origin_addr, origi	in_count, origin_datatype, target_rank,	
39		<pre>target_disp, target_count, target_datatype, op, win, request,</pre>		
40	TYPE(<pre>ierror) *). DIMENSION(). INTENT</pre>	Γ(IN). ASYNCHRONOUS :: origin addr	
41		TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr INTEGER, INTENT(IN) :: origin_count, target_rank, target_count		
42 43		TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype		
44	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp			
45		TYPE(MPI_Op), INTENT(IN) :: op TYPE(MPI_Win), INTENT(IN) :: win		
46 47	TYPE(MPI_Request), INTENT(OUT)) :: request	
47 48	INTEG	ER, OPTIONAL, INTENT(OUT)) :: ierror	

Fortran binding 1 MPI_RACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, 2 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST, 3 IERROR) 4 <type> ORIGIN_ADDR(*) 5 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT, 6 TARGET_DATATYPE, OP, WIN, REQUEST, IERROR 7 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP 8</type>				
MPI_RACCUMULATE is similar to MPI_ACCUMULATE (Section 11.3.4), except that it allocates a communication request object and associates it with the request handle (the argument request) that can be used to wait or test for completion. The completion of an MPI_RACCUMULATE operation indicates that the origin buffer is free to be updated. It does not indicate that the operation has completed at the target window.				
MPI_RGET_ACCUMULATE(origin_addr, origin_count, origin_datatype, result_addr, result_count, result_datatype, target_rank, target_disp, target_count, target_datatype, op, win, request)				
IN	origin_addr	initial address of buffer (choice)	19 20	
IN	origin_count	number of entries in origin buffer (non-negative integer)	21 22	
IN	origin_datatype	datatype of each entry in origin buffer (handle)	23	
OUT	result_addr	initial address of result buffer (choice)	24 25	
IN	result_count	number of entries in result buffer (non-negative integer)	25 26 27	
	weight descent		28	
IN	result_datatype	datatype of entries in result buffer (handle)	29	
IN	target_rank	rank of target (non-negative integer)	30	
IN	target_disp	displacement from start of window to beginning of target buffer (non-negative integer)	31 32	
IN	target_count	number of entries in target buffer (non-negative integer)	33 34	
IN	target_datatype	datatype of each entry in target buffer (handle)	35 36	
IN	ор	reduce operation (handle)	37	
IN	win	window object (handle)	38	
OUT	request	RMA request (handle)	39	
001	request	return request (namale)	40 41	
C binding	<u>,</u>		42	
int MPI_F	get_accumulate(const void	l *origin_addr, int origin_count,	43	
		datatype, void *result_addr,	44	
int result_count, MPI_Datatype result_datatype, 4				
	-	_Aint target_disp, int target_count,	46	
MPI_Datatype target_datatype, MPI_Op op, MPI_Win win, 47				

Unofficial Draft for Comment Only

MPI_Request *request)

1 Fortran 2008 binding $\mathbf{2}$ MPI_Rget_accumulate(origin_addr, origin_count, origin_datatype, 3 result_addr, result_count, result_datatype, target_rank, 4 target_disp, target_count, target_datatype, op, win, request, 5ierror) 6 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr 7 INTEGER, INTENT(IN) :: origin_count, result_count, target_rank, 8 target_count 9 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype, 10 target_datatype 11 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr 12INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp 13 TYPE(MPI_Op), INTENT(IN) :: op 14TYPE(MPI_Win), INTENT(IN) :: win 15TYPE(MPI_Request), INTENT(OUT) :: request 16INTEGER, OPTIONAL, INTENT(OUT) :: ierror 17 Fortran binding 18 MPI_RGET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, 19 RESULT_ADDR, RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, 20TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST, 21IERROR) 22 <type> ORIGIN_ADDR(*), RESULT_ADDR(*) 23INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE, 24TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST, 25IERROR 26INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP 2728MPI_RGET_ACCUMULATE is similar to MPI_GET_ACCUMULATE (Section 11.3.4),

²⁹ except that it allocates a communication request object and associates it with the request
 ³⁰ 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
 ³² the result buffer and the origin buffer is free to be updated. It does not indicate that the
 ³³ operation has been completed at the target window.

11.4 Memory Model

35

36

37 The memory semantics of RMA are best understood by using the concept of *public* and 38 private window copies. We assume that systems have a public memory region that is 39 addressable by all processes (e.g., the shared memory in shared memory machines or the 40 exposed main memory in distributed memory machines). In addition, most machines have 41 fast private buffers (e.g., transparent caches or explicit communication buffers) local to 42each process where copies of data elements from the main memory can be stored for faster 43 access. Such buffers are either coherent, i.e., all updates to main memory are reflected in 44 all private copies consistently, or non-coherent, i.e., conflicting accesses to main memory 45 need to be synchronized and updated in all private copies explicitly. Coherent systems 46 allow direct updates to remote memory without any participation of the remote side. Non-47 coherent systems, however, need to call RMA functions in order to reflect updates to the 48

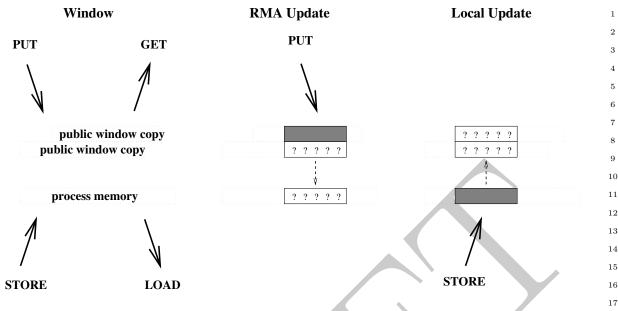


Figure 11.1: Schematic description of the public/private window operations in the MPI_WIN_SEPARATE memory model for two overlapping windows.

public window in their private memory. Thus, in coherent memory, the public and the private window are identical while they remain logically separate in the non-coherent case. MPI thus differentiates between two **memory models** called **RMA unified**, if public and private window are logically identical, and **RMA separate**, otherwise.

In the RMA separate model, there is only one instance of each variable in process memory, but a distinct *public* copy of the variable for each window that contains it. A load accesses the instance in process memory (this includes MPI sends). A local store accesses and updates the instance in process memory (this includes MPI receives), but the update may affect other public copies of the same locations. A get on a window accesses the public copy of that window. A put or accumulate on a window accesses and updates the public copy of that window, but the update may affect the private copy of the same locations in process memory, and public copies of other overlapping windows. This is illustrated in Figure 11.1.

In the RMA unified model, public and private copies are identical and updates via put or accumulate calls are eventually observed by load operations without additional RMA calls. A store access to a window is eventually visible to remote get or accumulate calls without additional RMA calls. These stronger semantics of the RMA unified model allow the user to omit some synchronization calls and potentially improve performance.

Advice to users. If accesses in the RMA unified model are not synchronized (with locks or flushes, see Section 11.5.3), load and store operations might observe changes to the memory while they are in progress. The order in which data is written is not specified unless further synchronization is used. This might lead to inconsistent views on memory and programs that assume that a transfer is complete by only checking parts of the message are erroneous. (*End of advice to users.*)

The memory model for a particular RMA window can be determined by accessing the attribute MPI_WIN_MODEL. If the memory model is the unified model, the value of this attribute is MPI_WIN_UNIFIED; otherwise, the value is MPI_WIN_SEPARATE.

Unofficial Draft for Comment Only

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 2

> 3 4

5

6

7

8

9 10

11

12

13

14

15

16

17

11.5 Synchronization Calls

RMA communications fall in two categories:

• active target communication, where data is moved from the memory of one process to the memory of another, and both are explicitly involved in the communication. This communication pattern is similar to message passing, except that all the data transfer arguments are provided by one process, and the second process only participates in the synchronization.

• **passive target communication**, where data is moved from the memory of one process to the memory of another, and only the origin process is explicitly involved in the transfer. Thus, two origin processes may communicate by accessing the same location in a target window. The process that owns the target window may be distinct from the two communicating processes, in which case it does not participate explicitly in the communication. This communication paradigm is closest to a shared memory model, where shared data can be accessed by all processes, irrespective of location.

RMA communication calls with argument win must occur at a process only within an **access epoch** for win. Such an epoch starts with an RMA synchronization call on win; it proceeds with zero or more RMA communication calls (e.g., MPI_PUT, MPI_GET or MPI_ACCUMULATE) on win; it completes with another synchronization call on win. This allows users to amortize one synchronization with multiple data transfers and provide implementors more flexibility in the implementation of RMA operations.

Distinct access epochs for win at the same process must be disjoint. On the other hand, epochs pertaining to different win arguments may overlap. Local operations or other MPI calls may also occur during an epoch.

In active target communication, a target window can be accessed by RMA operations 27only within an **exposure epoch**. Such an epoch is started and completed by RMA syn-28chronization calls executed by the target process. Distinct exposure epochs at a process on 29 the same window must be disjoint, but such an exposure epoch may overlap with exposure 30 epochs on other windows or with access epochs for the same or other win arguments. There 31 is a one-to-one matching between access epochs at origin processes and exposure epochs 32 on target processes: RMA operations issued by an origin process for a target window will 33 access that target window during the same exposure epoch if and only if they were issued 34 during the same access epoch. 35

In passive target communication the target process does not execute RMA synchronization calls, and there is no concept of an exposure epoch.

38 39

40

41

42

43

44

45

MPI provides three synchronization mechanisms:

1. The MPI_WIN_FENCE collective synchronization call supports a simple synchronization pattern that is often used in parallel computations: namely a loosely-synchronous model, where global computation phases alternate with global communication phases. This mechanism is most useful for loosely synchronous algorithms where the graph of communicating processes changes very frequently, or where each process communicates with many others.

This call is used for active target communication. An access epoch at an origin process or an exposure epoch at a target process are started and completed by calls to MPI_WIN_FENCE. A process can access windows at all processes in the group of win

Unofficial Draft for Comment Only

during such an access epoch, and the local window can be accessed by all processes in the group of win during such an exposure epoch.

2. The four functions MPI_WIN_START, MPI_WIN_COMPLETE, MPI_WIN_POST, and MPI_WIN_WAIT can be used to restrict synchronization to the minimum: only pairs of communicating processes synchronize, and they do so only when a synchronization is needed to order correctly RMA accesses to a window with respect to local accesses to that same window. This mechanism may be more efficient when each process communicates with few (logical) neighbors, and the communication graph is fixed or changes infrequently.

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

3. Finally, shared lock access is provided by the functions MPI_WIN_LOCK, MPI_WIN_LOCK_ALL, MPI_WIN_UNLOCK, and MPI_WIN_UNLOCK_ALL. MPI_WIN_LOCK and MPI_WIN_UNLOCK also provide exclusive lock capability. Lock synchronization is useful for MPI applications that emulate a shared memory model via MPI calls; e.g., in a "billboard" model, where processes can, at random times, access or update different parts of the billboard.

These four calls provide passive target communication. An access epoch is started by a call to MPI_WIN_LOCK or MPI_WIN_LOCK_ALL and terminated by a call to MPI_WIN_UNLOCK or MPI_WIN_UNLOCK_ALL, respectively.

Figure 11.2 illustrates the general synchronization pattern for active target communication. The synchronization between **post** and **start** ensures that the put call of the origin process does not start until the target process exposes the window (with the **post** call); the target process will expose the window only after preceding local accesses to the window have completed. The synchronization between complete and wait ensures that the put call of the origin process completes before the window is unexposed (with the wait call). The target process will execute following local accesses to the target window only after the wait returned.

Figure 11.2 shows operations occurring in the natural temporal order implied by the 35 synchronizations: the post occurs before the matching start, and complete occurs be-36 37 fore the matching wait. However, such strong synchronization is more than needed for correct ordering of window accesses. The semantics of MPI calls allow weak synchronization, as illustrated in Figure 11.3. The access to the target window is delayed until the window is exposed, after the post. However the start may complete earlier; the put and complete may also terminate earlier, if put data is buffered by the implementation. The synchronization calls order correctly window accesses, but do not necessarily synchronize other operations. This weaker synchronization semantic allows for more efficient implementations.

Figure 11.4 illustrates the general synchronization pattern for passive target commu-4546nication. The first origin process communicates data to the second origin process, through 47the memory of the target process; the target process is not explicitly involved in the com-48 munication. The lock and unlock calls ensure that the two RMA accesses do not occur

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6

 $\overline{7}$

8

9

10

11

12

13

14

15

1617

18

19

20

21

22

23

 24

25

26

2728

29

30

 31

32 33

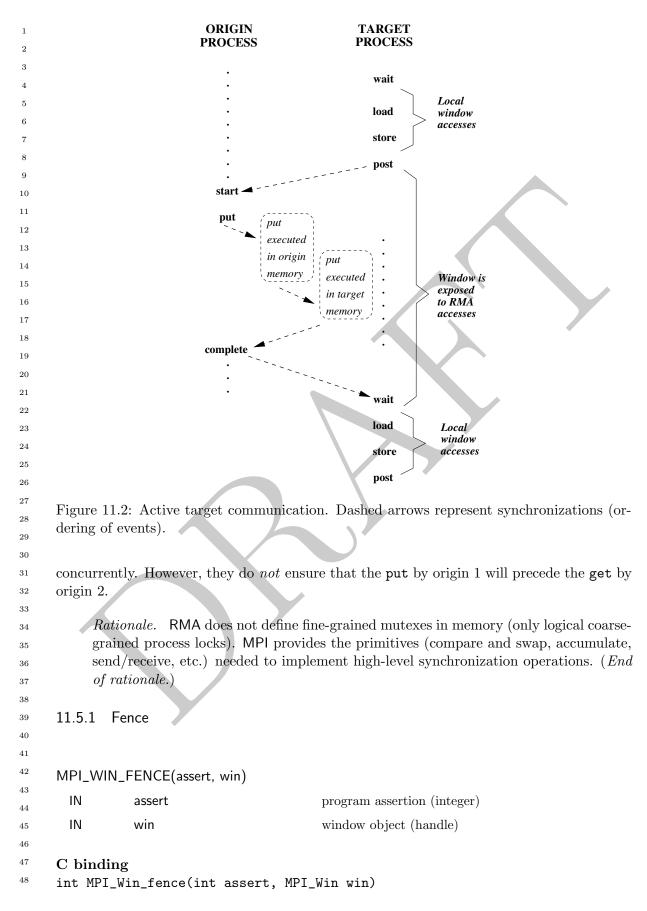
34

38

39

40 41

42



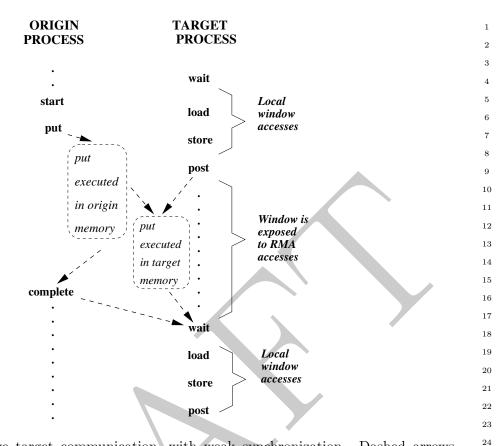


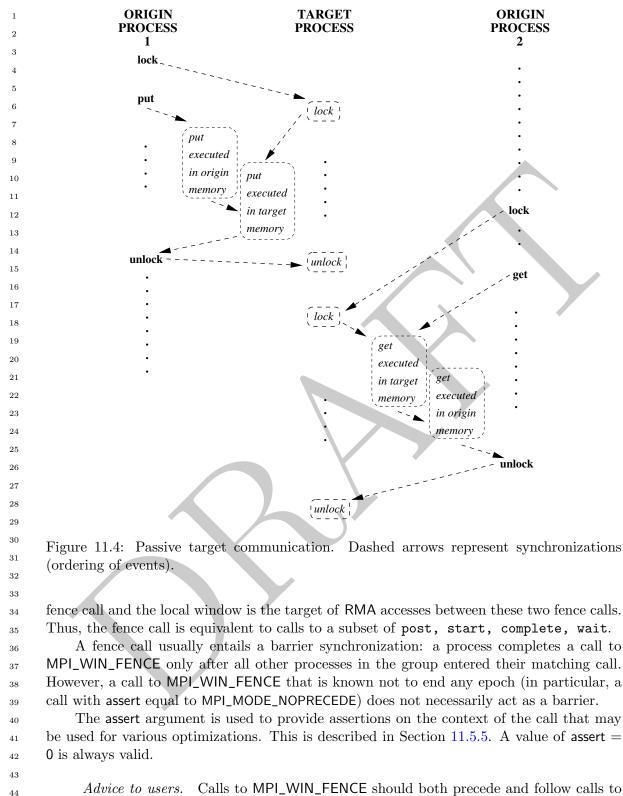
Figure 11.3: Active target communication, with weak synchronization. Dashed arrows represent synchronizations (ordering of events)

```
Fortran 2008 binding
MPI_Win_fence(assert, win, ierror)
    INTEGER, INTENT(IN) :: assert
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
MPI_WIN_FENCE(ASSERT, WIN, IERROR)
    INTEGER ASSERT, WIN, IERROR
```

The MPI call MPI_WIN_FENCE(assert, win) synchronizes RMA calls on win. The call is collective on the group of win. All RMA operations on win originating at a given process and started before the fence call will complete at that process before the fence call returns. They will be completed at their target before the fence call returns at the target. RMA operations on win started by a process after the fence call returns will access their target window only after MPI_WIN_FENCE has been called by the target process.

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

Unofficial Draft for Comment Only



Advice to users. Calls to MPI_WIN_FENCE should both precede and follow calls to RMA communication functions that are synchronized with fence calls. (*End of advice to users.*)

 $\frac{46}{47}$

45

11.5.2 General Active Target Synchronization

MPI_WIN_START(group, assert, win)

IN	group	group of target processes (handle)
IN	assert	program assertion (integer)
IN	win	window object (handle)

C binding

int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)

Fortran 2008 binding

```
MPI_Win_start(group, assert, win, ierror)
   TYPE(MPI_Group), INTENT(IN) :: group
   INTEGER, INTENT(IN) :: assert
   TYPE(MPI_Win), INTENT(IN) :: win
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

MPI_WIN_START(GROUP, ASSERT, WIN, IERROR) INTEGER GROUP, ASSERT, WIN, IERROR

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

The assert argument is used to provide assertions on the context of the call that may be used for various optimizations. This is described in Section 11.5.5. A value of assert =0 is always valid.

MPI_WIN_COMPLETE(win) win

```
IN
```

window object (handle)

```
C binding
```

int MPI_Win_complete(MPI_Win win)

```
Fortran 2008 binding
MPI_Win_complete(win, ierror)
```

TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

```
MPI_WIN_COMPLETE(WIN, IERROR)
    INTEGER WIN, IERROR
```

5 6

9 10

11

12

13

1415

1617

18

19

20

21

22

23 24

25

26

2728

2930

31

32 33

34 35

36 37

38

39

40

41

42

43 44

45

46

1 Completes an RMA access epoch on win started by a call to MPI_WIN_START. All $\mathbf{2}$ RMA communication calls issued on win during this epoch will have completed at the origin 3 when the call returns. 4

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 6 has completed at the origin.

Consider the sequence of calls in the example below.

```
Example 11.4
9
```

```
10
    MPI_Win_start(group, flag, win);
11
```

```
MPI_Put(..., win);
12
```

MPI_Win_complete(win); 13

14The call to MPI_WIN_COMPLETE does not return until the put call has completed 15at the origin; and the target window will be accessed by the put operation only after the 16call to MPI_WIN_START has matched a call to MPI_WIN_POST by the target process. 17This still leaves much choice to implementors. The call to MPI_WIN_START can block 18until the matching call to MPI_WIN_POST occurs at all target processes. One can also 19have implementations where the call to MPI_WIN_START is nonblocking, but the call to 20MPI_PUT blocks until the matching call to MPI_WIN_POST occurs; or implementations 21where the first two calls are nonblocking, but the call to MPI_WIN_COMPLETE blocks 22until the call to MPI_WIN_POST occurred; or even implementations where all three calls 23can complete before any target process has called MPI_WIN_POST — the data put must 24 be buffered, in this last case, so as to allow the put to complete at the origin ahead of its 25completion at the target. However, once the call to MPI_WIN_POST is issued, the sequence 26above must complete, without further dependencies. 27

```
28
29
```

MPI_WIN_POST(group, assert, win)

30	INI		
31	IN	group	group of origin processes (handle)
32	IN	assert	program assertion (integer)
33	IN	win	window object (handle)
34			
35	C bind	ling	
36	int MP	I_Win_post(MPI_Gro	up group, int assert, MPI_Win win)
37	T		
38	Fortra	n 2008 binding	
39	MPI_Wi	n_post(group, asse	rt, win, ierror)
40	TY	PE(MPI_Group), INT	ENT(IN) :: group
41	IN	TEGER, INTENT(IN)	:: assert
42	TY	PE(MPI_Win), INTEN	T(IN) :: win
43	IN	TEGER, OPTIONAL, I	NTENT(OUT) :: ierror
44	Fortra	n binding	
45	MPI_WI	N_POST(GROUP, ASSE	RT, WIN, IERROR)
46	IN	TEGER GROUP, ASSER	T, WIN, IERROR
47			
48			

 $\mathbf{5}$

 $\overline{7}$

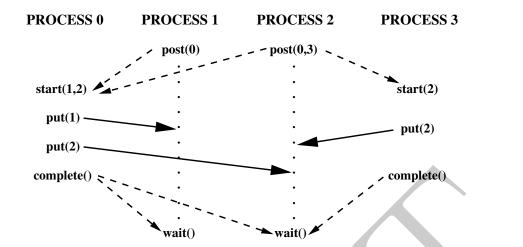


Figure 11.5: Active target communication. Dashed arrows represent synchronizations and solid arrows represent data transfer.

Starts an RMA exposure epoch for the local window associated with win. Only processes in group should access the window with RMA calls on win during this epoch. Each process in group must issue a matching call to MPI_WIN_START. MPI_WIN_POST does not block.

Completes an RMA exposure epoch started by a call to MPI_WIN_POST on win. This call matches calls to MPI_WIN_COMPLETE(win) issued by each of the origin processes that were granted access to the window during this epoch. The call to MPI_WIN_WAIT will block until all matching calls to MPI_WIN_COMPLETE have occurred. This guarantees that all these origin processes have completed their RMA accesses to the local window. When the call returns, all these RMA accesses will have completed at the target window.

Figure 11.5 illustrates the use of these four functions. Process 0 puts data in the windows of processes 1 and 2 and process 3 puts data in the window of process 2. Each start call lists the ranks of the processes whose windows will be accessed; each post call lists the ranks of the processes that access the local window. The figure illustrates a possible timing for the events, assuming strong synchronization; in a weak synchronization, the start, put or complete calls may occur ahead of the matching post calls.

Unofficial Draft for Comment Only

```
1
      MPI_WIN_TEST(win, flag)
2
       IN
                                              window object (handle)
                 win
3
       OUT
                 flag
                                              success flag (logical)
4
5
6
      C binding
\overline{7}
     int MPI_Win_test(MPI_Win win, int *flag)
8
      Fortran 2008 binding
9
      MPI_Win_test(win, flag, ierror)
10
          TYPE(MPI_Win), INTENT(IN) :: win
11
          LOGICAL, INTENT(OUT) :: flag
12
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
      Fortran binding
15
     MPI_WIN_TEST(WIN, FLAG, IERROR)
16
          INTEGER WIN, IERROR
17
          LOGICAL FLAG
18
          This is the nonblocking version of MPI_WIN_WAIT. It returns flag = true if all accesses
19
      to the local window by the group to which it was exposed by the corresponding
20
     MPI_WIN_POST call have been completed as signalled by matching MPI_WIN_COMPLETE
21
      calls, and flag = false otherwise. In the former case MPI_WIN_WAIT would have returned
22
      immediately. The effect of return of MPI_WIN_TEST with flag = true is the same as the
23
      effect of a return of MPI_WIN_WAIT. If flag = false is returned, then the call has no visible
24
      effect.
25
          MPI_WIN_TEST should be invoked only where MPI_WIN_WAIT can be invoked. Once
26
      the call has returned flag = true, it must not be invoked anew, until the window is posted
27
      anew.
28
          Assume that window win is associated with a "hidden" communicator wincomm, used
29
      for communication by the processes of win. The rules for matching of post and start calls
30
      and for matching complete and wait calls can be derived from the rules for matching sends
^{31}
      and receives, by considering the following (partial) model implementation.
32
33
      MPI_WIN_POST(group.0,win) initiates a nonblocking send with tag tag0 to each process
34
           in group, using wincomm. There is no need to wait for the completion of these sends.
35
36
      MPI_WIN_START(group,0,win) initiates a nonblocking receive with tag tag0 from each
37
           process in group, using wincomm. An RMA access to a window in target process i is
38
           delayed until the receive from i is completed.
39
      MPI_WIN_COMPLETE(win) initiates a nonblocking send with tag tag1 to each process
40
           in the group of the preceding start call. No need to wait for the completion of these
41
           sends.
42
43
     MPI_WIN_WAIT(win) initiates a nonblocking receive with tag tag1 from each process in
44
           the group of the preceding post call. Wait for the completion of all receives.
45
46
          No races can occur in a correct program: each of the sends matches a unique receive,
47
     and vice versa.
48
```

Rationale. The design for general active target synchronization requires the user to provide complete information on the communication pattern, at each end of a communication link: each origin specifies a list of targets, and each target specifies a list of origins. This provides maximum flexibility (hence, efficiency) for the implementor: each synchronization can be initiated by either side, since each "knows" the identity of the other. This also provides maximum protection from possible races. On the other hand, the design requires more information than RMA needs: in general, it is sufficient for the origin to know the rank of the target, but not vice versa. Users that want more "anonymous" communication will be required to use the fence or lock mechanisms. (*End of rationale.*)

Advice to users. Assume a communication pattern that is represented by a directed graph $G = \langle V, E \rangle$, where $V = \{0, \ldots, n-1\}$ and $ij \in E$ if origin process i accesses the window at target process j. Then each process i issues a call to MPI_WIN_POST($ingroup_i, \ldots$), followed by a call to MPI_WIN_START($outgroup_i, \ldots$), where $outgroup_i = \{j : ij \in E\}$ and $ingroup_i = \{j : ji \in E\}$. A call is a noop, and can be skipped, if the group argument is empty. After the communications calls, each process that issued a start will issue a complete. Finally, each process that issued a post will issue a wait.

Note that each process may call with a group argument that has different members. (*End of advice to users.*)

11.5.3 Lock

MPI WIN LOCK(lock type rank assert win)				
		27		
lock_type		28		
· ·	MPI_LOCK_SHARED (state)	29		
rank	rank of locked window (non-negative integer)	30		
assert	program assertion (integer)	31 32		
win	window object (handle)	33		
		34		
C binding				
int in i_win_rook(int rook_type; int rank; int about; in i_win win)				
Fortran 2008 binding				
MPI_Win_lock(lock_type, rank, assert, win, ierror)				
<pre>INTEGER, INTENT(IN) :: lock_type, rank, assert</pre>		40		
(MPI_Win), INTENT(IN) ::	win	41		
GER, OPTIONAL, INTENT(OUT) :: ierror	42		
Fortran binding				
INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR				
			s an RMA access epoch. The w	indow at the process with rank rank can be accessed
	<pre>lock_type rank assert win g Win_lock(int lock_type, i 2008 binding lock(lock_type, rank, ass GER, INTENT(IN) :: lock_t (MPI_Win), INTENT(IN) :: GER, OPTIONAL, INTENT(OUT binding LOCK(LOCK_TYPE, RANK, ASS GER LOCK_TYPE, RANK, ASSE</pre>	mPl_LOCK_SHARED (state) rank rank of locked window (non-negative integer) assert program assertion (integer) win window object (handle) vg Win_lock(int lock_type, int rank, int assert, MPI_Win win) 2008 binding lock(lock_type, rank, assert, win, ierror) GER, INTENT(IN) :: lock_type, rank, assert (MPI_Win), INTENT(IN) :: win GER, OPTIONAL, INTENT(OUT) :: ierror binding LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR)		

Starts an RMA access epoch. The window at the process with rank rank can be accessed by RMA operations on win during that epoch. Multiple RMA access epochs (with calls

Unofficial Draft for Comment Only

 $\mathbf{2}$

```
1
     to MPI_WIN_LOCK) can occur simultaneously; however, each access epoch must target a
\mathbf{2}
     different process.
3
4
     MPI_WIN_LOCK_ALL(assert, win)
5
6
       IN
                 assert
                                             program assertion (integer)
7
       IN
                 win
                                             window object (handle)
8
9
     C binding
10
     int MPI_Win_lock_all(int assert, MPI_Win win)
11
12
     Fortran 2008 binding
13
     MPI_Win_lock_all(assert, win, ierror)
14
          INTEGER, INTENT(IN) :: assert
15
          TYPE(MPI_Win), INTENT(IN) :: win
16
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
     Fortran binding
18
     MPI_WIN_LOCK_ALL(ASSERT, WIN, IERROR)
19
          INTEGER ASSERT, WIN, IERROR
20
21
          Starts an RMA access epoch to all processes in win, with a lock type of
22
     MPI_LOCK_SHARED. During the epoch, the calling process can access the window memory on
23
     all processes in win by using RMA operations. A window locked with MPI_WIN_LOCK_ALL
^{24}
     must be unlocked with MPI_WIN_UNLOCK_ALL. This routine is not collective — the ALL
25
     refers to a lock on all members of the group of the window.
26
27
           Advice to users. There may be additional overheads associated with using
28
           MPI_WIN_LOCK and MPI_WIN_LOCK_ALL concurrently on the same window. These
29
           overheads could be avoided by specifying the assertion MPI_MODE_NOCHECK when
30
           possible (see Section 11.5.5). (End of advice to users.)
^{31}
32
33
     MPI_WIN_UNLOCK(rank, win)
34
35
       IN
                 rank
                                             rank of window (non-negative integer)
36
       IN
                 win
                                             window object (handle)
37
38
     C binding
39
     int MPI_Win_unlock(int rank, MPI_Win win)
40
41
     Fortran 2008 binding
42
     MPI_Win_unlock(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_UNLOCK(RANK, WIN, IERROR)
48
```

INTEGER RANK, WIN, IERROR

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

MPI_WIN_UNLOCK_ALL(win)

win

IN

window object (handle)

C binding

int MPI_Win_unlock_all(MPI_Win win)

Fortran 2008 binding

```
MPI_Win_unlock_all(win, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

MPI_WIN_UNLOCK_ALL(WIN, IERROR) INTEGER WIN, IERROR

INTEGER WIN, IERROR

Completes a shared RMA access epoch started by a call to MPI_WIN_LOCK_ALL on window win. RMA operations issued during this epoch will have completed both at the origin and at the target when the call returns.

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

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

Rationale. An alternative is to require MPI to enforce mutual exclusion between exposure epochs and locking periods. But this would entail additional overheads when locks or active target synchronization do not interact in support of those rare interactions between the two mechanisms. The programming style that we encourage here is that a set of windows is used with only one synchronization mechanism at a time, with shifts from one mechanism to another being rare and involving global synchronization. (*End of rationale.*)

Advice to users. Users need to use explicit synchronization code in order to enforce mutual exclusion between locking periods and exposure epochs on a window. (End of advice to users.)

Unofficial Draft for Comment Only

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	* •	of RMA communication that is synchronized by	
2		ted by MPI_ALLOC_MEM (Section 8.2),	
3	MPI_WIN_ALLOCATE (Section 11.2.2),	MPI_WIN_ALLOCATE_SHARED (Section 11.2.3),	
4	or attached with MPI_WIN_ATTACH (S	ection 11.2.4). Locks can be used portably only in	
5	such memory.		
6			
7	-	of passive target communication when memory	
8	· · · · ·	nchronous software agent. Such an agent can be	
9	implemented more easily, and can achieve better performance, if restricted to specially		
10	allocated memory. It can be avoided altogether if shared memory is used. It seems		
11		allows one to use shared memory for third party	
12	communication in shared memory	machines.	
13	(End of rationale.)		
14			
15	Consider the sequence of calls in the	e example below.	
16			
17	Example 11.5		
18	MPI_Win_lock(MPI_LOCK_EXCLUSIVE,)	cank, assert, win):	
19	MPI_Put(, rank,, win);		
20	MPI_Win_unlock(rank, win);		
21			
22	The call to MPI_WIN_UNLOCK will not return until the put transfer has completed at		
23	the origin and at the target. This still leaves much freedom to implementors. The call to		
24	MPI_WIN_LOCK may block until an exclusive lock on the window is acquired; or, the first		
25	two calls may not block, while MPI_WIN_UNLOCK blocks until a lock is acquired — the		
26	update of the target window is then postponed until the call to MPI_WIN_UNLOCK occurs.		
27		is used to lock a local window, then the call must	
28		e lock may protect local load/store accesses to the	
29	window issued after the lock call returns		
30			
31	11.5.4 Flush and Sync		
32	All flush and sync functions can be calle	d only within passive target epochs	
33	The function of the functions can be care	a only while passive target epochs.	
34			
35	MPI_WIN_FLUSH(rank, win)		
36	IN rank	rank of target window (non-negative integer)	
37			
38	IN win	window object (handle)	
39			
40	C binding		
41	<pre>int MPI_Win_flush(int rank, MPI_With the second secon</pre>	in win)	
42	Fortran 2008 binding		
43	MPI_Win_flush(rank, win, ierror)		
44	INTEGER, INTENT(IN) :: rank		
45	TYPE(MPI_Win), INTENT(IN) :: win		
46	INTEGER, OPTIONAL, INTENT(OUT) :: ierror		
47			
48	Fortran binding		

MPI_WIN_FLUSH(RANK, WIN, IERROR)			
INTEGER RANK, WIN, IERROR	2		
MPL WIN FLUSH completes all outstandi	ng RMA operations initiated by the calling $\frac{3}{4}$		
MPI_WIN_FLUSH completes all outstanding RMA operations initiated by the process to the target rank on the specified window. The operations are completed			
the origin and at the target.	5		
the origin and at the target.	6		
	7		
MPI_WIN_FLUSH_ALL(win)	8		
IN win windo	w object (handle)		
iii wiido			
	11		
C binding	12		
int MPI_Win_flush_all(MPI_Win win)	13		
Fortran 2008 binding	14		
MPI_Win_flush_all(win, ierror)	15		
TYPE(MPI_Win), INTENT(IN) :: win	16		
INTEGER, OPTIONAL, INTENT(OUT) :: i	error ¹⁷		
	18		
Fortran binding	19		
MPI_WIN_FLUSH_ALL(WIN, IERROR)	20		
INTEGER WIN, IERROR	21		
All RMA operations issued by the calling p	cocess to any target on the specified window		
prior to this call and in the specified window w	· · · · · · · · · · · · · · · · · · ·		
the target when this call returns.			
J	25		
	26		
MPI_WIN_FLUSH_LOCAL(rank, win)	27		
IN rank rank	of target window (non-negative integer)		
	29		
IN WIN WINGO	w object (handle) 30		
	31		
C binding	32		
<pre>int MPI_Win_flush_local(int rank, MPI_W</pre>			
Fortran 2008 binding	34		
MPI_Win_flush_local(rank, win, ierror)	35		
INTEGER, INTENT(IN) :: rank	36		
TYPE(MPI_Win), INTENT(IN) :: win	37		
INTEGER, OPTIONAL, INTENT(OUT) :: i	error 38		
	39		
Fortran binding	40		
MPI_WIN_FLUSH_LOCAL(RANK, WIN, IERROR)	41		
INTEGER RANK, WIN, IERROR	42		
Locally completes at the origin all outstand	ling RMA operations initiated by the calling 43		
	Locally completes at the origin all outstanding RMA operations initiated by the calling cess to the target process specified by rank on the specified window. For example, after		
this routine completes, the user may reuse any	huffers provided to put get or accumulate		
operations.	¥ , C , 40		
	47		

```
1
     MPI_WIN_FLUSH_LOCAL_ALL(win)
\mathbf{2}
       IN
                                             window object (handle)
                 win
3
4
     C binding
5
     int MPI_Win_flush_local_all(MPI_Win win)
6
\overline{7}
     Fortran 2008 binding
8
     MPI_Win_flush_local_all(win, ierror)
9
          TYPE(MPI_Win), INTENT(IN) :: win
10
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
     Fortran binding
12
     MPI_WIN_FLUSH_LOCAL_ALL(WIN, IERROR)
13
          INTEGER WIN, IERROR
14
15
          All RMA operations issued to any target prior to this call in this window will have
16
     completed at the origin when MPI_WIN_FLUSH_LOCAL_ALL returns.
17
18
     MPI_WIN_SYNC(win)
19
20
       IN
                                             window object (handle)
                 win
21
22
     C binding
23
     int MPI_Win_sync(MPI_Win win)
^{24}
25
     Fortran 2008 binding
26
     MPI_Win_sync(win, ierror)
          TYPE(MPI_Win), INTENT(IN) :: win
27
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
     Fortran binding
30
     MPI_WIN_SYNC(WIN, IERROR)
^{31}
          INTEGER WIN, IERROR
32
33
          The call MPI_WIN_SYNC synchronizes the private and public window copies of win.
34
     For the purposes of synchronizing the private and public window, MPI_WIN_SYNC has the
35
     effect of ending and reopening an access and exposure epoch on the window (note that it
     does not actually end an epoch or complete any pending MPI RMA operations).
36
37
38
     11.5.5 Assertions
39
     The assert argument in the calls MPI_WIN_POST, MPI_WIN_START, MPI_WIN_FENCE,
40
     MPI_WIN_LOCK, and MPI_WIN_LOCK_ALL is used to provide assertions on the context of
41
     the call that may be used to optimize performance. The assert argument does not change
42
     program semantics if it provides correct information on the program — it is erroneous to
43
     provide incorrect information. Users may always provide assert = 0 to indicate a general
44
     case where no guarantees are made.
45
46
           Advice to users. Many implementations may not take advantage of the information
47
           in assert; some of the information is relevant only for noncoherent shared memory ma-
48
```

chines. Users should consult their implementation's manual to find which information is useful on each system. On the other hand, applications that provide correct assertions whenever applicable are portable and will take advantage of assertion specific optimizations whenever available. (*End of advice to users.*)

Advice to implementors. Implementations can always ignore the assert argument. Implementors should document which assert values are significant on their implementation. (End of advice to implementors.)

assert is the bit-vector OR of zero or more of the following integer constants: MPI_MODE_NOCHECK, MPI_MODE_NOSTORE, MPI_MODE_NOPUT, MPI_MODE_NOPRECEDE, and MPI_MODE_NOSUCCEED. The significant options are listed below for each call.

Advice to users. C/C++ users can use bit vector or (|) to combine these constants; Fortran 90 users can use the bit-vector IOR intrinsic. Alternatively, Fortran users can portably use integer addition to OR the constants (each constant should appear at most once in the addition!). (End of advice to users.)

MPI_WIN_START:

MPI_MODE_NOCHECK — the matching calls to MPI_WIN_POST have already completed on all target processes when the call to MPI_WIN_START is made. The nocheck option can be specified in a start call if and only if it is specified in each matching post call. This is similar to the optimization of "ready-send" that may save a handshake when the handshake is implicit in the code. However, ready-send is matched by a regular receive, whereas both start and post must specify the nocheck option.

MPI_WIN_POST:

- MPI_MODE_NOCHECK the matching calls to MPI_WIN_START have not yet occurred on any origin processes when the call to MPI_WIN_POST is made. The nocheck option can be specified by a post call if and only if it is specified by each matching start call.
- MPI_MODE_NOSTORE the local window was not updated by stores (or local get or receive calls) since last synchronization. This may avoid the need for cache synchronization at the post call.
- MPI_MODE_NOPUT the local window will not be updated by put or accumulate calls after the post call, until the ensuing (wait) synchronization. This may avoid the need for cache synchronization at the wait call.

MPI_WIN_FENCE:

- MPI_MODE_NOSTORE the local window was not updated by stores (or local get or receive calls) since last synchronization.
- MPI_MODE_NOPUT the local window will not be updated by put or accumulate calls after the fence call, until the ensuing (fence) synchronization.

47 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

41

42 43

44

45

1 2 3	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.
4 5 6 7	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.
8	MPI_WIN_LOCK, MPI_WIN_LOCK_ALL:
9 10 11 12 13	MPI_MODE_NOCHECK — no other process holds, or will attempt to acquire, a con- flicting 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.
14 15 16 17	Advice to users. Note that the nostore and noprecede flags provide information on what happened <i>before</i> the call; the noput and nosucceed flags provide information on what will happen <i>after</i> the call. (<i>End of advice to users.</i>)
18 19	11.5.6 Miscellaneous Clarifications
20 21 22 23 24 25	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.
26 27	11.6 Error Handling
28 29	11.6.1 Error Handlers
 30 31 32 33 34 35 36 37 	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 8.3).
38 39	11.6.2 Error Classes
40 41 42 43	The error classes for one-sided communication are defined in Table 11.2. RMA routines may (and almost certainly will) use other MPI error classes, such as MPI_ERR_OP or MPI_ERR_RANK.
44	11.7 Semantics and Correctness
45 46 47 48	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

CHAPTER 11. ONE-SIDED COMMUNICATIONS

Unofficial Draft for Comment Only

MPI_ERR_WIN	invalid win argument	1
MPI_ERR_BASE	invalid base argument	2
MPI_ERR_SIZE	invalid size argument	3
MPI_ERR_DISP	invalid disp argument	4
MPI_ERR_LOCKTYPE	invalid locktype argument	5
MPI_ERR_ASSERT	invalid assert argument	6
MPI_ERR_RMA_CONFLICT	conflicting accesses to window	7
MPI_ERR_RMA_SYNC	invalid synchronization of RMA calls	8
MPI_ERR_RMA_RANGE	target memory is not part of the window (in the case	9
	of a window created with	10
	MPI_WIN_CREATE_DYNAMIC, target memory is not	11
	attached)	12
MPI_ERR_RMA_ATTACH	memory cannot be attached (e.g., because of resource	13
	exhaustion)	14
MPI_ERR_RMA_SHARED	memory cannot be shared (e.g., some process in the	15
	group of the specified communicator cannot expose	16
	shared memory)	17
MPI_ERR_RMA_FLAVOR	passed window has the wrong flavor for the called	18
	function	19
		20
		21

Table 11.2: Error classes in one-sided communication routines

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.
- 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

Unofficial Draft for Comment Only

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

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.

10 The MPI_WIN_FENCE or MPI_WIN_WAIT call that completes the transfer from public 11 copy to private copy (6) is the same call that completes the put or accumulate operation in 12the window copy (2, 3). If a put or accumulate access was synchronized with a lock, then 13 the update of the public window copy is complete as soon as the updating process executed 14MPI_WIN_UNLOCK or MPI_WIN_UNLOCK_ALL. In the RMA separate memory model, the 15update of a private copy in the process memory may be delayed until the target process 16executes a synchronization call on that window (6). Thus, updates to process memory can 17always be delayed in the RMA separate memory model until the process executes a suitable 18 synchronization call, while they must complete in the RMA unified model without additional 19 synchronization calls. If fence or post-start-complete-wait synchronization is used, updates 20to a public window copy can be delayed in both memory models until the window owner 21executes a synchronization call. When passive target synchronization is used, it is necessary 22 to update the public window copy even if the window owner does not execute any related 23synchronization call. 24

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 31 example, the result of several origin processes performing concurrent MPI_PUT operations 32 to the same target location is undefined. In addition, the result of a single origin process 33 performing multiple MPI_PUT operations to the same target location within the same 34access epoch is also undefined. The result at the target may have all of the data from one 35 of the MPI_PUT operations (the "last" one, in some sense), bytes from some of each of the 36 operations, or something else. In MPI-2, such operations were erroneous. That meant that 37 an MPI implementation was permitted to signal an MPI exception. Thus, user programs or 38 tools that used MPI RMA could not portably permit such operations, even if the application 39 code could function correctly with such an undefined result. In MPI-3, these operations are 40 not erroneous, but do not have a defined behavior. 41

Rationale. As discussed in [7], requiring operations such as overlapping puts to
 be erroneous makes it difficult to use MPI RMA to implement programming models—
 such as Unified Parallel C (UPC) or SHMEM—that permit these operations. Further,
 while MPI-2 defined these operations as erroneous, the MPI Forum is unaware of any
 implementation that enforces this rule, as it would require significant overhead. Thus,
 relaxing this condition does not impact existing implementations or applications. (End
 of rationale.)

Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8

Advice to implementors. Overlapping accesses are undefined. However, to assist users in debugging code, implementations may wish to provide a mode in which such operations are detected and reported to the user. Note, however, that in MPI-3, such operations must not generate an MPI exception. (*End of advice to implementors.*)

A program with a well-defined outcome in the MPI_WIN_SEPARATE memory model must obey the following rules.

- S1. A location in a window must not be accessed with load/store operations once an update to that location has started, until the update becomes visible in the private window copy in process memory.
- S2. A location in a window must not be accessed as a target of an RMA operation once an update to that location has started, until the update becomes visible in the public window copy. There is one exception to this rule, in the case where the same variable is updated by two concurrent accumulates with the same predefined datatype, on the same window. Additional restrictions on the operation apply, see the info key accumulate_ops in Section 11.2.1.
- S3. A put or accumulate must not access a target window once a store or a put or accumulate update to another (overlapping) target window has started on a location in the target window, until the update becomes visible in the public copy of the window. Conversely, a store to process memory to a location in a window must not start once a put or accumulate update to that target window has started, until the put or accumulate update becomes visible in process memory. In both cases, the restriction applies to operations even if they access disjoint locations in the window.

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

Note that MPI_WIN_SYNC may be used within a passive target epoch to synchronize the private and public window copies (that is, updates to one are made visible to the other).

In the MPI_WIN_UNIFIED memory model, the rules are simpler because the public and private windows are the same. However, there are restrictions to avoid concurrent access to the same memory locations by different processes. The rules that a program with a well-defined outcome must obey in this case are:

- U1. A location in a window must not be accessed with load/store operations once an update to that location has started, until the update is complete, subject to the following special case.
- U2. Accessing a location in the window that is also the target of a remote update is valid (not erroneous) but the precise result will depend on the behavior of the implementation. Updates from a remote process will appear in the memory of the target, but 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

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 16of a remote read (but not update) is valid (not erroneous) but the precise result 17 will depend on the behavior of the implementation. Store updates will appear in 18 memory, but there are no atomicity or ordering guarantees if more than one byte is 19 updated. Updates are stable in the sense that once data appears in memory, the data 20remains until replaced by another update. This permits updates to memory with 21store operations without requiring an RMA epoch. Users are cautioned that remote 22 accesses to a window that is updated by the local process has defined behavior only 23if the other rules given here and elsewhere in this chapter are followed. 24
 - U4. A location in a window must not be accessed as a target of an RMA operation once an update to that location has started and until the update completes at the target. There is one exception to this rule: in the case where the same location is updated by two concurrent accumulates with the same predefined datatype on the same window. Additional restrictions on the operation apply; see the info key accumulate_ops in Section 11.2.1.
 - U5. A put or accumulate must not access a target window once a store, put, or accumulate update to another (overlapping) target window has started on the same location in the target window and until the update completes at the target window. Conversely, a store operation to a location in a window must not start once a put or accumulate update to the same location in that target window has started and until the put or accumulate update completes at the target.
 - Advice to users. In the unified memory model, in the case where the window is in shared memory, MPI_WIN_SYNC can be used to order store operations and make store updates to the window visible to other processes and threads. Use of this routine is necessary to ensure portable behavior when point-to-point, collective, or shared memory synchronization is used in place of an RMA synchronization routine. MPI_WIN_SYNC should be called by the writer before the non-RMA synchronization operation and by the reader after the non-RMA synchronization, as shown in Example 11.21. (End of advice to users.)
- ⁴⁸ A program that violates these rules has undefined behavior.

Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8

9

10 11

12

13

14 15

25

26

27

28

29

30

31 32

33

34

35

36

37

38 39

40

41

42

43 44

45

46

Advice to users. A user can write correct programs by following the following rules:

- **fence:** During each period between fence calls, each window is either updated by put or accumulate calls, or updated by stores, but not both. Locations updated by put or accumulate calls should not be accessed during the same period (with the exception of concurrent updates to the same location by accumulate calls). Locations accessed by get calls should not be updated during the same period.
- **post-start-complete-wait:** A window should not be updated with store operations while posted if it is being updated by put or accumulate calls. Locations updated by put or accumulate calls should not be accessed while the window is posted (with the exception of concurrent updates to the same location by accumulate calls). Locations accessed by get calls should not be updated while the window is posted.

With the post-start synchronization, the target process can tell the origin process that its window is now ready for RMA access; with the complete-wait synchronization, the origin process can tell the target process that it has finished its RMA accesses to the window.

- **lock:** Updates to the window are protected by exclusive locks if they may conflict. Nonconflicting accesses (such as read-only accesses or accumulate accesses) are protected by shared locks, both for load/store accesses and for RMA accesses.
- changing window or synchronization mode: One can change synchronization mode, or change the window used to access a location that belongs to two over-lapping windows, when the process memory and the window copy are guaranteed to have the same values. This is true after a local call to MPI_WIN_FENCE, if RMA accesses to the window are synchronized with fences; after a local call to MPI_WIN_WAIT, if the accesses are synchronized with post-start-complete-wait; after the call at the origin (local or remote) to MPI_WIN_UNLOCK or MPI_WIN_UNLOCK_ALL if the accesses are synchronized with locks.

In addition, a process should not access the local buffer of a get operation until the operation is complete, and should not update the local buffer of a put or accumulate operation until that operation is complete.

The RMA synchronization operations define when updates are guaranteed to become visible in public and private windows. Updates may become visible earlier, but such behavior is implementation dependent. (*End of advice to users.*)

The semantics are illustrated by the following examples:

Example 11.6 The following example demonstrates updating a memory location inside a window for the separate memory model, according to Rule 5. The MPI_WIN_LOCK and MPI_WIN_UNLOCK calls around the store to X in process B are necessary to ensure consistency between the public and private copies of the window.

 $\mathbf{2}$

 $\overline{7}$

 24

 31

```
1
     Process A:
                                    Process B:
\mathbf{2}
                                    window location X
3
4
                                    MPI_Win_lock(EXCLUSIVE, B)
5
                                    store X /* local update to private copy of B */
6
                                    MPI_Win_unlock(B)
7
                                    /* now visible in public window copy */
8
9
     MPI_Barrier
                                    MPI_Barrier
10
11
     MPI_Win_lock(EXCLUSIVE, B)
12
     MPI_Get(X) /* ok, read from public window */
13
     MPI_Win_unlock(B)
14
15
     Example 11.7 In the RMA unified model, although the public and private copies of the
16
     windows are synchronized, caution must be used when combining load/stores and multi-
17
     process synchronization. Although the following example appears correct, the compiler or
18
     hardware may delay the store to X after the barrier, possibly resulting in the MPI_GET
19
     returning an incorrect value of X.
20
21
     Process A:
                               Process B:
22
                               window location X
23
^{24}
                               store X /* update to private & public copy of B */
25
     MPI_Barrier
                               MPI_Barrier
26
     MPI_Win_lock_all
27
     MPI_Get(X) /* ok, read from window */
28
     MPI_Win_flush_local(B)
29
     /* read value in X */
30
     MPI_Win_unlock_all
31
32
     MPI_BARRIER provides process synchronization, but not memory synchronization. The
33
     example could potentially be made safe through the use of compiler- and hardware-specific
34
     notations to ensure the store to X occurs before process B enters the MPI_BARRIER. The
35
     use of one-sided synchronization calls, as shown in Example 11.6, also ensures the correct
36
     result.
37
38
     Example 11.8 The following example demonstrates the reading of a memory location
39
     updated by a remote process (Rule 6) in the RMA separate memory model. Although
40
     the MPI_WIN_UNLOCK on process A and the MPI_BARRIER ensure that the public copy
41
     on process B reflects the updated value of X, the call to MPI_WIN_LOCK by process B is
42
     necessary to synchronize the private copy with the public copy.
43
                                    Process B:
     Process A:
44
                                    window location X
45
46
     MPI_Win_lock(EXCLUSIVE, B)
47
     MPI_Put(X) /* update to public window */
48
```

MPI_Win_unlock(B)		
MPI_Barrier	MPI_Barrier	
	MPI_Win_lock(EXCLUSIVE, B) /* now visible in private copy of B */ load X MPI_Win_unlock(B)	
Note that in this example, the barrier is not critical to the semantic correctness. The use of exclusive locks guarantees a remote process will not modify the public copy after MPI_WIN_LOCK synchronizes the private and public copies. A polling implementation looking for changes in X on process B would be semantically correct. The barrier is required to ensure that process A performs the put operation before process B performs the load of X .		
model, because the load of X can Process B does not need to ex MPI_WIN_LOCK as the MPI_F window, the scheduling of the la and hardware specific notations	mple 11.7, the following example is unsafe even in the unified in not be guaranteed to occur after the MPI_BARRIER. While eplicitly synchronize the public and private copies through PUT will update both the public and private copies of the boad could result in old values of X being returned. Compiler is could ensure the load occurs after the data is updated, or it on calls can be used to ensure the proper result.	
Process A:	Process B: window location X	
<pre>MPI_Win_lock_all MPI_Put(X) /* update to wi MPI_Win_flush(B)</pre>		
MPI_Barrier	MPI_Barrier	
MPI_Win_unlock_all	load X	
Example 11.10 The following	ng example further clarifies Rule 5. MPI_WIN_LOCK and	

Example 11.10 The following example further clarifies Rule 5. MPI_WIN_LOCK and MPI_WIN_LOCK_ALL do *not* update the public copy of a window with changes to the private copy. Therefore, there is no guarantee that process A in the following sequence will see the value of X as updated by the local store by process B before the lock.

-		39
Process A:	Process B:	40
	window location X	41
		42
	store X /* update to private copy of B */	43
	MPI_Win_lock(SHARED, B)	44
MPI_Barrier	MPI_Barrier	45
		46
MPI_Win_lock(SHARED, B)		47
MPI_Get(X) /* X may be the	X before the store */	48

Unofficial Draft for Comment Only

1 MPI_Win_unlock(B) $\mathbf{2}$ MPI_Win_unlock(B) 3 /* update on X now visible in public window */ 4 The addition of an MPI_WIN_SYNC before the call to MPI_BARRIER by process B would 5guarantee process A would see the updated value of X, as the public copy of the window 6 would be explicitly synchronized with the private copy. 7 8 **Example 11.11** Similar to the previous example, Rule 5 can have unexpected implications 9 for general active target synchronization with the RMA separate memory model. It is not 10 guaranteed that process B reads the value of **X** as per the local update by process A, because 11 neither MPI_WIN_WAIT nor MPI_WIN_COMPLETE calls by process A ensure visibility in 12the public window copy. 13 14Process A: Process B: 15window location X 16window location Y 1718 store Y 19 MPI_Win_post(A, B) /* Y visible in public window */ 20MPI_Win_start(A) MPI_Win_start(A) 2122store X /* update to private window */ 23 24 MPI_Win_complete MPI_Win_complete 25MPI_Win_wait 26/* update on X may not yet visible in public window */ 2728MPI_Barrier MPI_Barrier 29 30 MPI_Win_lock(EXCLUSIVE, A) 31MPI_Get(X) /* may return an obsolete value */ 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 35 36 a local MPI_PUT that updates the public window copy. Note that by this replacement X 37 may become visible in the private copy of process A only after the MPI_WIN_WAIT call in 38 process A. The update to Y made before the MPI_WIN_POST call is visible in the public 39 window after the MPI_WIN_POST call and therefore process B will read the proper value 40of Y. The MPI_GET(Y) call could be moved to the epoch started by the MPI_WIN_START 41 operation, and process B would still get the value stored by process A. 42**Example 11.12** The following example demonstrates the interaction of general active 43 target synchronization with local read operations with the RMA separate memory model. 44Rules 5 and 6 do not guarantee that the private copy of X at process B has been updated 45before the load takes place. 464748

Process A:

Process B: window location X

```
MPI_Win_lock(EXCLUSIVE, B)
MPI_Put(X) /* update to public window */
MPI_Win_unlock(B)
```

MPI_Barrier MPI_Barrier

> MPI_Win_post(B) MPI_Win_start(B) load X /* access to private window */

/* may return an obsolete value */

MPI_Win_complete MPI_Win_wait

To ensure that the value put by process A is read, the local load must be replaced with a local MPI_GET operation, or must be placed after the call to MPI_WIN_WAIT.

11.7.1 Atomicity

The outcome of concurrent accumulate operations to the same location with the same predefined datatype is as if the accumulates were done at that location in some serial order. Additional restrictions on the operation apply; see the info key accumulate_ops in Section 11.2.1. Concurrent accumulate operations with different origin and target pairs are not ordered. Thus, there is no guarantee that the entire call to an accumulate operation is executed atomically. The effect of this lack of atomicity is limited: The previous correctness conditions imply that a location updated by a call to an accumulate operation cannot be accessed by a load or an RMA call other than accumulate until the accumulate operation has completed (at the target). Different interleavings can lead to different results only to the extent that computer arithmetics are not truly associative or commutative. The outcome of accumulate operations with overlapping types of different sizes or target displacements is undefined.

11.7.2 Ordering

Accumulate calls enable element-wise atomic read and write to remote memory locations. 38MPI specifies ordering between accumulate operations from an origin process to the same (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 MPI_GET_ACCUMULATE) and writes (e.g., with MPI_ACCUMULATE) are executed and 4344committed in program order. Ordering only applies to operations originating at the same origin that access overlapping target memory regions. MPI does not provide any guarantees for accesses or updates from different origin processes to overlapping target memory regions.

47The default strict ordering may incur a significant performance penalty. MPI specifies 48 the info key accumulate_ordering to allow relaxation of the ordering semantics when specified

Unofficial Draft for Comment Only

2

3

Z

5 6

9 10

11

12

13

141516

17

18

19

2021

22 23

24

25

26

27

28

29

30

 31

32

33

34

3536

37

39

40

1 to any window creation function. The values for this key are as follows. If set to none, $\mathbf{2}$ then no ordering will be guaranteed for accumulate calls. This was the behavior for RMA 3 in MPI-2 but is not the default in MPI-3. The key can be set to a comma-separated list 4 of required access orderings at the target. Allowed values in the comma-separated list $\mathbf{5}$ are rar, war, raw, and waw for read-after-read, write-after-read, read-after-write, and write-6 after-write ordering, respectively. These indicate whether operations of the specified type $\overline{7}$ complete in the order they were issued. For example, raw means that any writes must 8 complete at the target before subsequent reads. These ordering requirements apply only to 9 operations issued by the same origin process and targeting the same target process. The 10 default value for accumulate_ordering is rar,raw,war,waw, which implies that writes complete at 11the 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 13that 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 15of the accumulate_ordering key could be set to rar, waw. The order of values is not significant. 16Note that the above ordering semantics apply only to accumulate operations, not put 17

and get. Put and get within an epoch are unordered.

18 19

20

11.7.3 Progress

One-sided communication has the same progress requirements as point-to-point communi-21cation: once a communication is enabled it is guaranteed to complete. RMA calls must have 22local semantics, except when required for synchronization with other RMA calls. 23

There is some fuzziness in the definition of the time when a RMA communication 24 becomes enabled. This fuzziness provides to the implementor more flexibility than with 25point-to-point communication. Access to a target window becomes enabled once the corre-26sponding synchronization (such as MPI_WIN_FENCE or MPI_WIN_POST) has executed. On 27the origin process, an RMA communication may become enabled as soon as the correspond-28ing put, get or accumulate call has executed, or as late as when the ensuing synchronization 29 call is issued. Once the communication is enabled both at the origin and at the target, the 30 communication must complete. 31

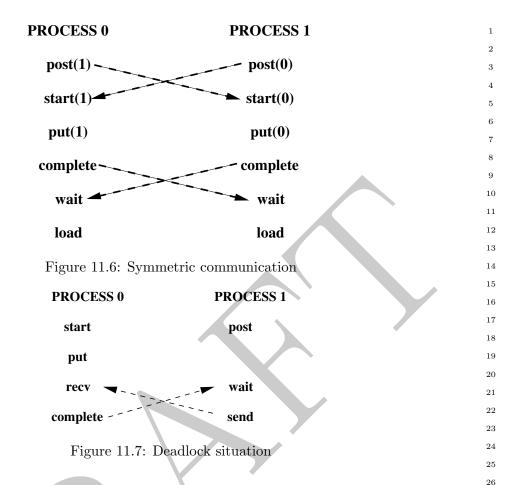
Consider the code fragment in Example 11.4. Some of the calls may block if the target 32 window is not posted. However, if the target window is posted, then the code fragment 33 must complete. The data transfer may start as soon as the put call occurs, but may be 34delayed until the ensuing complete call occurs. 35

Consider the code fragment in Example 11.5. Some of the calls may block if another 36 process holds a conflicting lock. However, if no conflicting lock is held, then the code 37 fragment must complete. 38

Consider the code illustrated in Figure 11.6. Each process updates the window of 39 the other process using a put operation, then accesses its own window. The post calls are 40 nonblocking, and should complete. Once the post calls occur, RMA access to the windows is 41 enabled, so that each process should complete the sequence of calls start-put-complete. Once 42these are done, the wait calls should complete at both processes. Thus, this communication 43 should not deadlock, irrespective of the amount of data transferred. 44

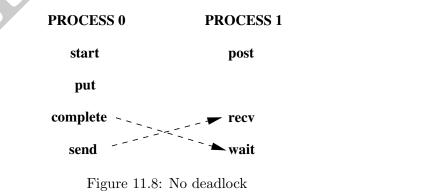
Assume, in the last example, that the order of the post and start calls is reversed at 45each process. Then, the code may deadlock, as each process may block on the start call, 46 waiting for the matching post to occur. Similarly, the program will deadlock if the order of 47the complete and wait calls is reversed at each process. 48

Unofficial Draft for Comment Only



The following two examples illustrate the fact that the synchronization between complete and wait is not symmetric: the wait call blocks until the complete executes, but not vice versa. Consider the code illustrated in Figure 11.7. This code will deadlock: the wait of process 1 blocks until process 0 calls complete, and the receive of process 0 blocks until process 1 calls send. Consider, on the other hand, the code illustrated in Figure 11.8. This code will not deadlock. Once process 1 calls post, then the sequence start, put, complete on process 0 can proceed to completion. Process 0 will reach the send call, allowing the receive call of process 1 to complete.

Rationale. MPI implementations must guarantee that a process makes progress on all enabled communications it participates in, while blocked on an MPI call. This is true



 $46 \\ 47$

for send-receive communication and applies to RMA communication as well. Thus, in the example in Figure 11.8, the put and complete calls of process 0 should complete while process 1 is blocked on the receive call. This may require the involvement of process 1, e.g., to transfer the data put, while it is blocked on the receive call.

A similar issue is whether such progress must occur while a process is busy computing, or blocked in a non-MPI call. Suppose that in the last example the send-receive pair is replaced by a write-to-socket/read-from-socket pair. Then MPI does not specify whether deadlock is avoided. Suppose that the blocking receive of process 1 is replaced by a very long compute loop. Then, according to one interpretation of the MPI standard, process 0 must return from the complete call after a bounded delay, even if process 1 does not reach any MPI call in this period of time. According to another interpretation, the complete call may block until process 1 reaches the wait call, or reaches another MPI call. The qualitative behavior is the same, under both interpretations, unless a process is caught in an infinite compute loop, in which case the difference may not matter. However, the quantitative expectations are different. Different MPI implementations reflect these different interpretations. While this ambiguity is unfortunate, the MPI Forum decided not to define which interpretation of the standard is the correct one, since the issue is contentious. (*End of rationale.*)

11.7.4 Registers and Compiler Optimizations

Advice to users. All the material in this section is an advice to users. (End of advice to users.)

A coherence problem exists between variables kept in registers and the memory values of these variables. An RMA call may access a variable in memory (or cache), while the up-to-date value of this variable is in register. A get will not return the latest variable value, and a put may be overwritten when the register is stored back in memory. Note that these issues are unrelated to the RMA memory model; that is, these issues apply even if the memory model is MPI_WIN_UNIFIED.

The problem is illustrated by the following code:

02			
33	Source of Process 1	Source of Process 2	Executed in Process 2
34	bbbb = 777	buff = 999	reg_A:=999
35	call MPI_WIN_FENCE	call MPI_WIN_FENCE	
36	call MPI_PUT(bbbb		stop appl.thread
37	into buff of process 2)		buff:=777 in PUT handler
38			continue appl.thread
39	call MPI_WIN_FENCE	call MPI_WIN_FENCE	
40		ccc = buff	ccc:=reg_A
41			

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 18.1.16.

Programs written in C avoid this problem, because of the semantics of C. Many Fortran
 compilers will avoid this problem, without disabling compiler optimizations. However, in
 order to avoid register coherence problems in a completely portable manner, users should

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19 20

21 22

23

24

31 32

42

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 18.1.10-18.1.20. Sections 18.1.17 to 18.1.17discuss several solutions for the problem in this example.

11.8 Examples

Example 11.13 The following example shows a generic loosely synchronous, iterative code, using fence synchronization. The window at each process consists of array A, which contains the origin and target buffers of the put calls.

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

The same code could be written with get rather than put. Note that, during the communication phase, each window is concurrently read (as origin buffer of puts) and written (as target buffer of puts). This is OK, provided that there is no overlap between the target buffer of a put and another communication buffer.

Example 11.14 Same generic example, with more computation/communication overlap. We assume that the update phase is broken into two subphases: the first, where the "boundary," which is involved in communication, is updated, and the second, where the "core," which neither uses nor provides communicated data, is updated.

```
. . .
while (!converged(A)) {
 update_boundary(A);
 MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win);
 for(i=0; i < fromneighbors; i++)</pre>
    MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
                    fromdisp[i], 1, fromtype[i], win);
 update_core(A);
 MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
}
```

The get communication can be concurrent with the core update, since they do not access the 43 44same 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

1

 $\mathbf{2}$

3

4 56

7 8 9

10

11

1213

14

15

16

17

18

19

20

2122

23

 24

25

2627

28

29

30

3132

33

34

35

36

37

38

39

40

41

42

4546

47

```
1
     Example 11.15 Same code as in Example 11.13, rewritten using post-start-complete-wait.
\mathbf{2}
3
     while (!converged(A)) {
4
       update(A);
5
       MPI_Win_post(fromgroup, 0, win);
6
       MPI_Win_start(togroup, 0, win);
7
       for(i=0; i < toneighbors; i++)</pre>
8
         MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
9
                  todisp[i], 1, totype[i], win);
10
       MPI_Win_complete(win);
11
       MPI_Win_wait(win);
12
     }
13
14
15
     Example 11.16 Same example, with split phases, as in Example 11.14.
16
17
     . . .
     while (!converged(A)) {
18
       update_boundary(A);
19
       MPI_Win_post(togroup, MPI_MODE_NOPUT, win);
20
       MPI_Win_start(fromgroup, 0, win);
21
       for(i=0; i < fromneighbors; i++)</pre>
22
         MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
23
                  fromdisp[i], 1, fromtype[i], win);
24
       update_core(A);
25
       MPI_Win_complete(win);
26
       MPI_Win_wait(win);
27
     }
28
29
30
     Example 11.17 A checkerboard, or double buffer communication pattern, that allows
^{31}
     more computation/communication overlap. Array A0 is updated using values of array A1,
32
     and vice versa. We assume that communication is symmetric: if process A gets data from
33
     process B, then process B gets data from process A. Window wini consists of array Ai.
34
35
     . . .
36
     if (!converged(A0,A1))
37
       MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
38
     MPI_Barrier(comm0);
     /* the barrier is needed because the start call inside the
39
40
     loop uses the nocheck option */
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
47
                      fromdisp0[i], 1, fromtype0[i], win0);
48
       update1(A1); /* local update of A1 that is
```

Unofficial Draft for Comment Only

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

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

Put calls can be used, instead of get calls, if the area of array A0 (resp. A1) used by the update(A1, A0) (resp. update(A0, A1)) call is disjoint from the area modified by the RMA communication. On some systems, a put call may be more efficient than a get call, as it requires information exchange only in one direction.

In the next several examples, for conciseness, the expression

```
z = MPI_Get_accumulate(...)
```

means to perform an MPI_GET_ACCUMULATE with the result buffer (given by result_addr in the description of MPI_GET_ACCUMULATE) on the left side of the assignment, in this case, z. This format is also used with MPI_COMPARE_AND_SWAP.

Example 11.18 The following example implements a naive, non-scalable counting semaphore. The example demonstrates the use of MPI_WIN_SYNC to manipulate the public copy of X, as well as MPI_WIN_FLUSH to complete operations without ending the access epoch opened with MPI_WIN_LOCK_ALL. To avoid the rules regarding synchronization of the public and private copies of windows, MPI_ACCUMULATE and MPI_GET_ACCUMULATE are used to write to or read from the local public copy.

Process A:	Process B: 4	3
MPI_Win_lock_all	MPI_Win_lock_all 4	4
window location X	4	5
X=2	4	6
MPI_Win_sync	4	7
MPI_Barrier	MPI_Barrier 4	8

1

 $\mathbf{2}$

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

```
1
\mathbf{2}
     MPI_Accumulate(X, MPI_SUM, -1)
                                                   MPI_Accumulate(X, MPI_SUM, -1)
3
4
     stack variable z
                                                    stack variable z
5
     do
                                                    do
6
       z = MPI_Get_accumulate(X,
                                                      z = MPI_Get_accumulate(X,
7
            MPI_NO_OP, 0)
                                                           MPI_NO_OP, 0)
8
       MPI_Win_flush(A)
                                                      MPI_Win_flush(A)
9
     while(z!=0)
                                                    while(z!=0)
10
11
     MPI_Win_unlock_all
                                                   MPI_Win_unlock_all
12
```

Example 11.19 Implementing a critical region between two processes (Peterson's algorithm). Despite their appearance in the following example, MPI_WIN_LOCK_ALL and MPI_WIN_UNLOCK_ALL are not collective calls, but it is frequently useful to start shared access epochs to all processes from all other processes in a window. Once the access epochs are established, accumulate communication operations and flush and sync synchronization operations can be used to read from or write to the public copy of the window.

```
20
     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
                                              Y=1
     X=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}
     y=MPI_Get_accumulate(Y,
                                              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, 0)
36
       t=MPI_Get_accumulate(T,
                                                t=MPI_Get_accumulate(T,
37
          MPI_NO_OP, 0)
                                                    MPI_NO_OP, 0)
38
       MPI_Win_flush_all
                                                MPI_Win_flush(A)
39
     done
                                              done
40
     // critical region
                                              // critical region
41
     MPI_Accumulate(X, MPI_REPLACE, 0)
                                              MPI_Accumulate(Y, MPI_REPLACE, 0)
42
     MPI_Win_unlock_all
                                              MPI_Win_unlock_all
43
```

Example 11.20 Implementing a critical region between multiple processes with compare
 and swap. The call to MPI_WIN_SYNC is necessary on Process A after local initialization
 of A to guarantee the public copy has been updated with the initialization value found in
 the private copy. It would also be valid to call MPI_ACCUMULATE with MPI_REPLACE to

Unofficial Draft for Comment Only

13

14

15

16

17

18

19

directly initialize the public copy. A call to MPI_WIN_FLUSH would be necessary to assure **A** in the public copy of Process A had been updated before the barrier.

Process A: MPI_Win_lock_all atomic location A	Process B: MPI_Win_lock_all
A=O MPI_Win_sync	
MPI_Barrier	MPI_Barrier
stack variable r=1	stack variable r=1
while(r != 0) do	while(r != 0) do
r = MPI_Compare_and_swap(A, 0, 1)	r = MPI_Compare_and_swap(A, 0, 1)
MPI_Win_flush(A)	MPI_Win_flush(A)
done	done
// critical region	// critical region
<pre>r = MPI_Compare_and_swap(A, 1, 0)</pre>	<pre>r = MPI_Compare_and_swap(A, 1, 0)</pre>
MPI_Win_unlock_all	MPI_Win_unlock_all

Example 11.21 The following example demonstrates the proper synchronization in the unified memory model when a data transfer is implemented with load and store in the case of windows in shared memory (instead of MPI_PUT or MPI_GET) and the synchronization between processes is performed using point-to-point communication. The synchronization between processes must be supplemented with a memory synchronization through calls to MPI_WIN_SYNC, which act locally as a processor-memory barrier. In Fortran, if MPI_ASYNC_PROTECTS_NONBLOCKING is .FALSE. or the variable X is not declared as ASYNCHRONOUS, reordering of the accesses to the variable X must be prevented with MPI_F_SYNC_REG operations. (No equivalent function is needed in C.)

The variable X is contained within a shared memory window and X corresponds to the same memory location at both processes. The MPI_WIN_SYNC operation performed by process A ensures completion of the load/store operations issued by process A. The MPI_WIN_SYNC operation performed by process B ensures that process A's updates to X are visible to process B.

Process A	Process B	34
		35
MPI_WIN_LOCK_ALL(MPI_WIN_LOCK_ALL(36
MPI_MODE_NOCHECK,win)	MPI_MODE_NOCHECK,win)	37
		38
DO	DO	39
X=		40
		41
MPI_F_SYNC_REG(X)		42
MPI_WIN_SYNC(win)		43
MPI_SEND	MPI_RECV	44
	MPI_WIN_SYNC(win)	45
	MPI_F_SYNC_REG(X)	46
		47
	print X	48

Unofficial Draft for Comment Only

 24

```
1
\mathbf{2}
                                              MPI_F_SYNC_REG(X)
3
       MPI_RECV
                                              MPI_SEND
4
       MPI_F_SYNC_REG(X)
\mathbf{5}
     END DO
                                           END DO
6
7
     MPI_WIN_UNLOCK_ALL(win)
                                           MPI_WIN_UNLOCK_ALL(win)
8
9
     Example 11.22 The following example shows how request-based operations can be used
10
     to overlap communication with computation. Each process fetches, processes, and writes
11
     the result for NSTEPS chunks of data. Instead of a single buffer, M local buffers are used to
12
     allow up to M communication operations to overlap with computation.
13
14
     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++) {</pre>
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
43
44
     Example 11.23 The following example constructs a distributed shared linked list using
45
     dynamic windows. Initially process 0 creates the head of the list, attaches it to the window,
46
     and broadcasts the pointer to all processes. All processes then concurrently append N new
```

CHAPTER 11. ONE-SIDED COMMUNICATIONS

552

47

48

elements to the list. When a process attempts to attach its element to the tail of the

list it may discover that its tail pointer is stale and it must chase ahead to the new tail

before the element can be attached. This example requires some modification to work in an environment where the layout of the structures is different on different processes.

```
4
#define NUM_ELEMS 10
                                                                                     5
                                                                                     6
#define LLIST_ELEM_NEXT_RANK ( offsetof(llist_elem_t, next) + \
                                                                                     7
                                 offsetof(llist_ptr_t, rank) )
#define LLIST_ELEM_NEXT_DISP ( offsetof(llist_elem_t, next) + \
                                                                                     9
                                 offsetof(llist_ptr_t, disp) )
                                                                                     10
                                                                                     11
/* Linked list pointer */
                                                                                     12
typedef struct {
                                                                                     13
 MPI_Aint disp;
                                                                                     14
  int
           rank;
                                                                                     15
} llist_ptr_t;
                                                                                     16
                                                                                     17
/* Linked list element */
                                                                                     18
typedef struct {
                                                                                     19
  llist_ptr_t next;
                                                                                     20
  int value;
                                                                                     21
} llist_elem_t;
                                                                                     22
                                                                                     23
const llist_ptr_t nil = { (MPI_Aint) MPI_BOTTOM,
                                                                                     ^{24}
                                                                                     25
/* List of locally allocated list elements. */
                                                                                     26
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
  llist_elem_t *elem_ptr;
                                                                                     34
                                                                                     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
  /* Add the element to the list of local elements so we can free
                                                                                     42
     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
```

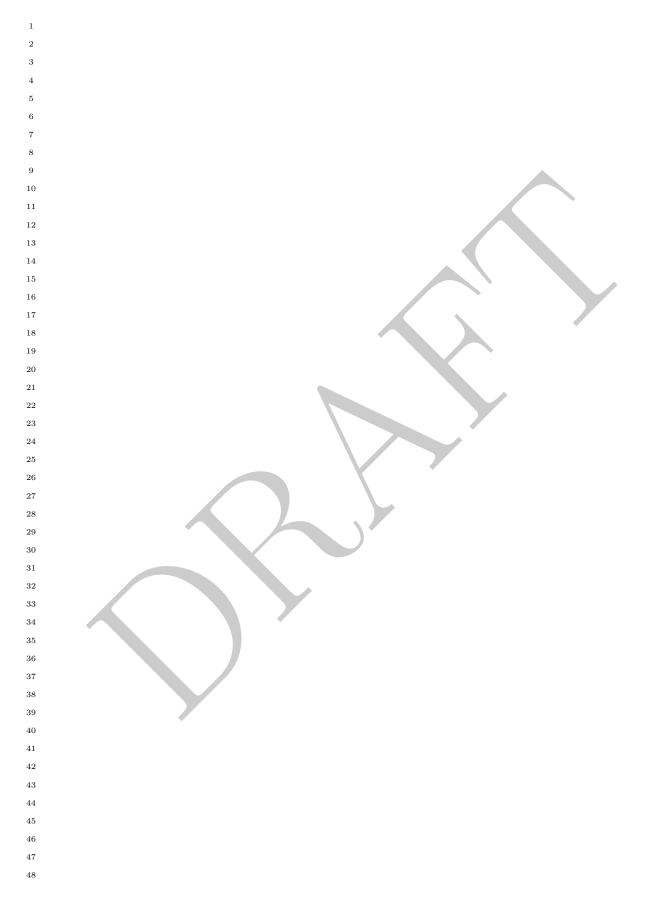
1

 $\mathbf{2}$

```
1
       my_elems_count++;
2
3
       MPI_Get_address(elem_ptr, &disp);
4
       return disp;
5
     }
6
7
     int main(int argc, char *argv[]) {
8
       int
                      procid, nproc, i;
9
       MPI_Win
                      llist_win;
10
                     head_ptr, tail_ptr;
       llist_ptr_t
11
12
       MPI_Init(&argc, &argv);
13
14
       MPI_Comm_rank(MPI_COMM_WORLD, &procid);
15
       MPI_Comm_size(MPI_COMM_WORLD, &nproc);
16
17
       MPI_Win_create_dynamic(MPI_INFO_NULL, MPI_COMM_WORLD, &llist_win);
18
19
       /* Process 0 creates the head node */
20
       if (procid == 0)
21
         head_ptr.disp = alloc_elem(-1, llist_win);
22
       /* Broadcast the head pointer to everyone */
23
^{24}
       head_ptr.rank = 0;
25
       MPI_Bcast(&head_ptr.disp, 1, MPI_AINT, 0, MPI_COMM_WORLD);
26
       tail_ptr = head_ptr;
27
28
       /* Lock the window for shared access to all targets */
29
       MPI_Win_lock_all(0, llist_win);
30
31
       /* All processes concurrently append NUM_ELEMS elements to the list */
32
       for (i = 0; i < NUM_ELEMS; i++) {</pre>
33
         llist_ptr_t new_elem_ptr;
34
        int success;
35
36
         /* Create a new list element and attach it to the window */
37
         new_elem_ptr.rank = procid;
38
         new_elem_ptr.disp = alloc_elem(procid, llist_win);
39
40
         /* Append the new node to the list. This might take multiple
41
            attempts if others have already appended and our tail pointer
42
            is stale. */
43
         do {
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);
                                                                                    \mathbf{2}
                                                                                    3
    MPI_Win_flush(tail_ptr.rank, llist_win);
    success = (next_tail_ptr.rank == nil.rank);
                                                                                    4
                                                                                    5
    if (success) {
                                                                                    6
      MPI_Accumulate(&new_elem_ptr.disp, 1, MPI_AINT, tail_ptr.rank,
          MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_DISP), 1,
          MPI_AINT, MPI_REPLACE, llist_win);
                                                                                    9
                                                                                    10
                                                                                    11
      MPI_Win_flush(tail_ptr.rank, llist_win);
      tail_ptr = new_elem_ptr;
                                                                                    12
                                                                                    13
    } else {
                                                                                    14
                                                                                    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,
             MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_DISP),
                                                                                    20
                                                                                   21
             1, MPI_AINT, MPI_NO_OP, llist_win);
                                                                                   22
        MPI_Win_flush(tail_ptr.rank, llist_win);
                                                                                   23
                                                                                    ^{24}
      } while (next_tail_ptr.disp == nil.disp);
      tail_ptr = next_tail_ptr;
                                                                                    25
                                                                                    26
    }
  } while (!success);
                                                                                    27
}
                                                                                    28
                                                                                    29
                                                                                    30
MPI_Win_unlock_all(llist_win);
MPI_Barrier(MPI_COMM_WORLD);
                                                                                    31
                                                                                    32
                                                                                    33
/* Free all the elements in the list */
for ( ; my_elems_count > 0; my_elems_count--) {
                                                                                   34
  MPI_Win_detach(llist_win,my_elems[my_elems_count-1]);
                                                                                   35
  MPI_Free_mem(my_elems[my_elems_count-1]);
                                                                                   36
                                                                                   37
}
                                                                                    38
MPI_Win_free(&llist_win);
                                                                                    39
                                                                                    40
                                                                                    41
                                                                                    42
                                                                                    43
                                                                                    44
                                                                                    45
                                                                                    46
```



Chapter 12

External Interfaces

12.1 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 12.3 deals with setting the information found in **status**. This functionality is needed for generalized requests.

12.2 Generalized Requests

The goal of generalized requests is to allow users to define new nonblocking operations. Such an outstanding nonblocking operation is represented by a (generalized) request. A fundamental property of nonblocking operations is that progress toward the completion of this operation occurs asynchronously, i.e., concurrently with normal program execution. Typically, this requires execution of code concurrently with the execution of the user code, e.g., in a separate thread or in a signal handler. Operating systems provide a variety of mechanisms in support of concurrent execution. MPI does not attempt to standardize or to replace these mechanisms: it is assumed programmers who wish to define new asynchronous operations will use the mechanisms provided by the underlying operating system. Thus, the calls in this section only provide a means for defining the effect of MPI calls such as MPI_WAIT or MPI_CANCEL when they apply to generalized requests, and for signaling to MPI the completion of a generalized operation.

Rationale. It is tempting to also define an MPI standard mechanism for achieving concurrent execution of user-defined nonblocking operations. However, it is difficult to define such a mechanism without consideration of the specific mechanisms used in the operating system. The Forum feels that concurrency mechanisms are a proper part of the underlying operating system and should not be standardized by MPI; the MPI standard should only deal with the interaction of such mechanisms with MPI. (*End of rationale.*)

For a regular request, the operation associated with the request is performed by the MPI implementation, and the operation completes without intervention by the application. For a generalized request, the operation associated with the request is performed by the application; therefore, the application must notify MPI through a call to

 $45 \\ 46$

```
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

 $\mathbf{2}$

 $\frac{44}{45}$

```
12.2.1 Examples
```

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

```
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;
}
```

12.3 Associating Information with Status

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

Each MPI call fills in the appropriate fields in the status object. Any unused fields will have undefined values. A call to MPI_{TEST|WAIT}{ANY|SOME|ALL} can modify any of the fields in the status object. Specifically, it can modify fields that are undefined. The fields with meaningful values for a given request are defined in the sections with the new request.

Generalized requests raise additional considerations. Here, the user provides the functions to deal with the request. Unlike other MPI calls, the user needs to provide the information to be returned in the status. The status argument is provided directly to the callback function where the status needs to be set. Users can directly set the values in 3 of the 5 status values. The count and cancel fields are opaque. To overcome this, these calls are provided:

```
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	P
33	int MPI_S		Status *status, MPI_Datatype datatype,
34		int count)	
35	Fortron 2	2008 binding	
36		Ŭ	detet
37			datatype, count, ierror)
38		(MPI_Status), INTENT(INO	
39		(MPI_Datatype), INTENT(I	v1
40		ER, INTENT(IN) :: count	
41	INTEG	ER, OPTIONAL, INTENT(OU	T) :: ierror
42	Fortran b	oinding	
43		0	DATATYPE, COUNT, IERROR)
44		-	ZE), DATATYPE, COUNT, IERROR
45			DATATITE, OUNT, TEMOT
46			
47			
48			

564

1

 $\mathbf{2}$

3

4 5

> 6 7

8

9

10

11

MPI_STAT	US_SET_ELEMENTS_X(state	us, datatype, count)	1
INOUT	status	status with which to associate count (Status)	2
IN	datatype	datatype associated with count (handle)	$\frac{3}{4}$
IN	count	number of elements to associate with status (integer)	5
	count	number of elements to associate with status (integer)	6
C bindin	σ		7
	-	_Status *status, MPI_Datatype datatype,	8
-	MPI_Count count)		9
Fortran S	2008 binding		10 11
	0	datatype, count, ierror)	11
	(MPI_Status), INTENT(INOU		13
TYPE	(MPI_Datatype), INTENT(IN) :: datatype	14
	GER(KIND=MPI_COUNT_KIND),		15
INTEC	GER, OPTIONAL, INTENT(OUT) :: ierror	16
Fortran k	binding		17
MPI_STATU	JS_SET_ELEMENTS_X(STATUS,	DATATYPE, COUNT, IERROR)	18 19
	GER STATUS (MPI_STATUS_SIZ		20
INTEC	GER(KIND=MPI_COUNT_KIND)	COUNT	21
	· · · ·	e part of status so that a call to	22
		EMENTS_X will return count. MPI_GET_COUNT	23
will return a compatible value. 24			
Rati	onale. The number of eleme	ents is set instead of the count because the former	25 26
		er of datatypes. (<i>End of rationale.</i>)	20
	2		
			29
			30
MPI_GET_ELEMENTS_X(status, datatype, count) must use a datatype argument that has the same type signature as the datatype argument that was used in the call to			31
		_STATUS_SET_ELEMENTS_X.	32
			33 34
Rati	onale. The requirement of	matching type signatures for these calls is similar	35
		a count is set by a receive operation: in that case,	36
		PI_GET_ELEMENTS, and MPI_GET_ELEMENTS_X	37
	l of rationale.)	e signature as the datatype used in the receive call.	38
(Lint			39
			40
ΜΟΙ ΟΤΛΤ		flag)	41 42
	US_SET_CANCELLED(status	-,	43
INOUT	status	status with which to associate cancel flag (Status)	44
IN	flag	if true, indicates request was cancelled (logical)	45
			46
C bindin	•		47
int MPI_S	status_set_cancelled(MPI_	Status *status, int flag)	48

Unofficial Draft for Comment Only

Fortran 2008 binding $\mathbf{2}$ MPI_Status_set_cancelled(status, flag, ierror) TYPE(MPI_Status), INTENT(INOUT) :: status LOGICAL, INTENT(IN) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), IERROR LOGICAL FLAG If flag is set to true then a subsequent call to MPI_TEST_CANCELLED(status, flag) will also return flag = true, otherwise it will return false. Advice to users. Users are advised not to reuse the status fields for values other than those for which they were intended. Doing so may lead to unexpected results when using the status object. For example, calling MPI_GET_ELEMENTS may cause an error if the value is out of range or it may be impossible to detect such an error. The extra_state argument provided with a generalized request can be used to return information that does not logically belong in status. Furthermore, modifying the values in a status set internally by MPI, e.g., MPI_RECV, may lead to unpredictable results and is strongly discouraged. (End of advice to users.)

Chapter 13

I/O

13.1 Introduction

POSIX provides a model of a widely portable file system, but the portability and optimization needed for parallel I/O cannot be achieved with the POSIX interface.

The significant optimizations required for efficiency (e.g., grouping [50], collective buffering [8, 16, 51, 55, 62], and disk-directed I/O [45]) can only be implemented if the parallel I/O system provides a high-level interface supporting partitioning of file data among processes and a collective interface supporting complete transfers of global data structures between process memories and files. In addition, further efficiencies can be gained via support for asynchronous I/O, strided accesses, and control over physical file layout on storage devices (disks). The I/O environment described in this chapter provides these facilities.

Instead of defining I/O access modes to express the common patterns for accessing a shared file (broadcast, reduction, scatter, gather), we chose another approach in which data partitioning is expressed using derived datatypes. Compared to a limited set of predefined access patterns, this approach has the advantage of added flexibility and expressiveness.

13.1.1 Definitions

- file An MPI file is an ordered collection of typed data items. MPI supports random or sequential access to any integral set of these items. A file is opened collectively by a group of processes. All collective I/O calls on a file are collective over this group.
- **displacement** A file *displacement* is an absolute byte position relative to the beginning of a file. The displacement defines the location where a *view* begins. Note that a "file displacement" is distinct from a "typemap displacement."
- etype An *etype* (*elementary* datatype) is the unit of data access and positioning. It can be any MPI predefined or derived datatype. Derived etypes can be constructed using any of the MPI datatype constructor routines, provided all resulting typemap displacements are non-negative and monotonically nondecreasing. Data access is performed in etype units, reading or writing whole data items of type etype. Offsets are expressed as a count of etypes; file pointers point to the beginning of etypes. Depending on context, the term "etype" is used to describe one of three aspects of an elementary datatype: a particular MPI type, a data item of that type, or the extent of that type.

 24

filetype A *filetype* is the basis for partitioning a file among processes and defines a template for accessing the file. A filetype is either a single etype or a derived MPI datatype constructed from multiple instances of the same etype. In addition, the extent of any hole in the filetype must be a multiple of the etype's extent. The displacements in the typemap of the filetype are not required to be distinct, but they must be non-negative and monotonically nondecreasing.

view A *view* defines the current set of data visible and accessible from an open file as an ordered set of etypes. Each process has its own view of the file, defined by three quantities: a displacement, an etype, and a filetype. The pattern described by a filetype is repeated, beginning at the displacement, to define the view. The pattern of repetition is defined to be the same pattern that MPI_TYPE_CONTIGUOUS would produce if it were passed the filetype and an arbitrarily large count. Figure 13.1 shows how the tiling works; note that the filetype in this example must have explicit lower and upper bounds set in order for the initial and final holes to be repeated in the view. Views can be changed by the user during program execution. The default view is a linear byte stream (displacement is zero, etype and filetype equal to MPL_BYTE).

tiling a file with the filetype:
holes -
tiling a file with the filetype:
filing a file with the filetype.
displacement accessible data
Figure 13.1: Etypes and filetypes
A group of processes can use complementary views to achieve a global data distribution
such as a scatter/gather pattern (see Figure 13.2).
etype
process 0 filetype
process 1 filetype
process 2 filetype
tiling a file with the filetypes:
displacement
Figure 13.2: Partitioning a file among parallel processes

offset An offset is a position in the file relative to the current view, expressed as a count of etypes. Holes in the view's filetype are skipped when calculating this position. Offset 0 is the location of the first etype visible in the view (after skipping the displacement and any initial holes in the view). For example, an offset of 2 for process 1 in Figure 13.2 is the position of the eighth etype in the file after the displacement. An "explicit offset" is an offset that is used as an argument in explicit data access routines.

Unofficial Draft for Comment Only

- file size and end of file The *size* of an MPI file is measured in bytes from the beginning of the file. A newly created file has a size of zero bytes. Using the size as an absolute displacement gives the position of the byte immediately following the last byte in the file. For any given view, the *end of file* is the offset of the first etype accessible in the current view starting after the last byte in the file.
- file pointer A *file pointer* is an implicit offset maintained by MPI. "Individual file pointers" are file pointers that are local to each process that opened the file. A "shared file pointer" is a file pointer that is shared by the group of processes that opened the file.
- file handle A *file handle* is an opaque object created by MPI_FILE_OPEN and freed by MPI_FILE_CLOSE. All operations on an open file reference the file through the file handle.

13.2 File Manipulation

13.2.1 Opening a File

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

C binding

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

 $\frac{44}{45}$

1 the same file. (Values for info may vary.) comm must be an intracommunicator; it is $\mathbf{2}$ erroneous to pass an intercommunicator to MPI_FILE_OPEN. Errors in MPI_FILE_OPEN 3 are raised using the default file error handler (see Section 13.7). When using the World 4 Model (Section 10.1), a process can open a file independently of other processes by using 5the MPI_COMM_SELF communicator. Applications using the Sessions Model (Section 10.3) 6 can achieve the same result using communicators created from the mpi://SELF process set. $\overline{7}$ 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 [41]. Exactly one of MPI_MODE_RDONLY, MPI_MODE_RDWR, or MPI_MODE_WRONLY, must be specified. It is erroneous to specify MPI_MODE_CREATE or MPI_MODE_EXCL in conjunction with MPI_MODE_RDONLY; it is erroneous to specify MPI_MODE_SEQUENTIAL together with MPI_MODE_RDWR.

The MPI_MODE_DELETE_ON_CLOSE mode causes the file to be deleted (equivalent to performing an MPI_FILE_DELETE) when the file is closed.

The MPI_MODE_UNIQUE_OPEN mode allows an implementation to optimize access by eliminating the overhead of file locking. It is erroneous to open a file in this mode unless the file will not be concurrently opened elsewhere.

Advice to users. For MPI_MODE_UNIQUE_OPEN, not opened elsewhere includes both inside and outside the MPI environment. In particular, one needs to be aware of potential external events which may open files (e.g., automated backup facilities). When MPI_MODE_UNIQUE_OPEN is specified, the user is responsible for ensuring that no such external events take place. (End of advice to users.)

The MPI_MODE_SEQUENTIAL mode allows an implementation to optimize access to some sequential devices (tapes and network streams). It is erroneous to attempt nonsequential access to a file that has been opened in this mode.

Specifying MPI_MODE_APPEND only guarantees that all shared and individual file pointers are positioned at the initial end of file when MPI_FILE_OPEN returns. Subsequent positioning of file pointers is application dependent. In particular, the implementation does not ensure that all writes are appended.

Errors related to the access mode are raised in the class MPI_ERR_AMODE.

The info argument is used to provide information regarding file access patterns and file system specifics (see Section 13.2.8). The constant MPI_INFO_NULL can be used when no info needs to be specified.

Advice to users. Some file attributes are inherently implementation dependent (e.g., file permissions). These attributes must be set using either the info argument or facilities outside the scope of MPI. (End of advice to users.)

Files are opened by default using nonatomic mode file consistency semantics (see Section 13.6.1). The more stringent atomic mode consistency semantics, required for atomicity of conflicting accesses, can be set using MPI_FILE_SET_ATOMICITY.

Unofficial Draft for Comment Only

 $\overline{7}$

 24

```
1
     13.2.2 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
      13.2.3
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
48
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

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 13.2.8). The constant MPI_INFO_NULL refers to the null info, and can be used when no info needs to be specified.

If a process currently has the file open, the behavior of any access to the file (as well as the behavior of any outstanding accesses) is implementation dependent. In addition, whether an open file is deleted or not is also implementation dependent. If the file is not deleted, an error in the class MPI_ERR_FILE_IN_USE or MPI_ERR_ACCESS will be raised. Errors are raised using the default file error handler (see Section 13.7).

13.2.4 Resizing a File

MPI_FILE_	SET_SIZE(fh, size)	
INOUT	fh	file handle (handle)
IN	size	size to truncate or expand file (integer)

C binding

int MPI_File_set_size(MPI_File fh, MPI_Offset size)

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 13.6.1). 1213 13.2.5 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 expensive. (End of advice to users.)

47 48

45

13.2.6 Querying the Size of a File		1
		2
		3
MPI_FILE_GET_SIZE(fh, size)		4
IN fh	file handle (handle)	5
OUT size		6 7
SOT SIZE	size of the file in bytes (integer)	8
C binding		9
int MPI_File_get_size(MPI_File fh,	MPI Offset *size)	10
<u> </u>		11
Fortran 2008 binding		12
<pre>MPI_File_get_size(fh, size, ierror TYPE(MPI_File), INTENT(IN) ::</pre>		13
INTEGER(KIND=MPI_OFFSET_KIND),		14
INTEGER, OPTIONAL, INTENT(OUT)		15 16
		10
Fortran binding		18
MPI_FILE_GET_SIZE(FH, SIZE, IERROF INTEGER FH, IERROR		19
INTEGER(KIND=MPI_OFFSET_KIND)	SIZE	20
		21
	e, the current size in bytes of the file associated with	22
data access operation (see Section 13.6.1	semantics are concerned, MPI_FILE_GET_SIZE is a	23
data access operation (see Section 15.0.1).	24 25
13.2.7 Querying File Parameters		26
		27
		28
MPI_FILE_GET_GROUP(fh, group)		29
IN fh	file handle (handle)	30
		31
OUT group	group which opened the file (handle)	32
C binding		33 34
int MPI_File_get_group(MPI_File fr	MPI Group *group)	35
	, in 1_droup (group)	36
Fortran 2008 binding	、 、	37
MPI_File_get_group(fh, group, ierr		38
<pre>TYPE(MPI_File), INTENT(IN) :: TYPE(MPI_Group), INTENT(OUT) :</pre>		39
INTEGER, OPTIONAL, INTENT(OUT)	.	40
		41
Fortran binding		42 43
MPI_FILE_GET_GROUP(FH, GROUP, IERF INTEGER FH, GROUP, IERROR	(UK)	43 44
		45
	uplicate of the group of the communicator used to	46

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 13.1 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
```

```
<sup>48</sup> example, the following code fragment would check for MPI_MODE_RDONLY.
```

```
CALL BIT_QUERY(MPI_MODE_RDONLY, 30, AMODE, BIT_FOUND)
IF (BIT_FOUND .EQ. 1) THEN
PRINT *, ' FOUND READ-ONLY BIT IN AMODE=', AMODE
ELSE
PRINT *, ' READ-ONLY BIT NOT FOUND IN AMODE=', AMODE
END IF
```

13.2.8 File Info

Hints specified via info (see Chapter 9) allow a user to provide information such as file access patterns and file system specifics to direct optimization. Providing hints may enable an implementation to deliver increased I/O performance or minimize the use of system resources. An implementation is free to ignore all hints; however, applications must comply with any info hints they provide that are used by the MPI implementation (i.e., are returned by a call to MPI_FILE_GET_INFO) and that place a restriction on the behavior of the application. Hints are specified on a per file basis, in MPI_FILE_OPEN, MPI_FILE_DELETE, MPI_FILE_SET_VIEW, and MPI_FILE_SET_INFO, via the opaque info object. When an info object that specifies a subset of valid hints is passed to MPI_FILE_SET_VIEW or MPI_FILE_SET_INFO, there will be no effect on previously set or defaulted hints that the info does not specify.

Advice to implementors. It may happen that a program is coded with hints for one system, and later executes on another system that does not support these hints. In general, unsupported hints should simply be ignored. Needless to say, no hint can be mandatory. However, for each hint used by a specific implementation, a default value must be provided when the user does not specify a value for this hint. (*End of advice to implementors.*)

	29
MPI_FILE_SET_INFO(fh, info)	30
	31
INOUT fh file handle (handle)	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	38
MPI_File_set_info(fh, info, ierror)	39
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)	44
INTEGER FH, INFO, IERROR	45
	46
MPL FILE SET INFO updates 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 48

 $\mathbf{2}$

 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

fh

info_used

int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)

²⁴ Fortran 2008 binding

MPI_File_get_info(fh, info_used, ierror)
 TYPE(MPI_File), INTENT(IN) :: fh
 TYPE(MPI_Info), INTENT(OUT) :: info_used

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

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 9.)

These hints mainly affect access patterns and the layout of data on parallel I/O devices. For each hint name introduced, we describe the purpose of the hint, and the type of the hint

value. The "[**SAME**]" annotation specifies that the hint values provided by all participating processes must be identical; otherwise the program is erroneous. In addition, some hints are context dependent, and are only used by an implementation at specific times (e.g., file_perm is only useful during file creation).

- access_style (comma separated list of strings): This hint specifies the manner in which the file will be accessed until the file is closed or until the access_style key value is altered. The hint value is a comma separated list of the following: read_once, write_once, read_mostly, write_mostly, sequential, reverse_sequential, and random.
- collective_buffering (boolean) [SAME]: This hint specifies whether the application may benefit from collective buffering. Collective buffering is an optimization performed on collective accesses. Accesses to the file are performed on behalf of all processes in the group by a number of target nodes. These target nodes coalesce small requests into large disk accesses. Valid values for this key are true and false. Collective buffering parameters are further directed via additional hints: cb_block_size, cb_buffer_size, and cb_nodes.
- cb_block_size (integer) [SAME]: This hint specifies the block size to be used for collective buffering file access. *Target nodes* access data in chunks of this size. The chunks are distributed among target nodes in a round-robin (cyclic) pattern.
- cb_buffer_size (integer) [SAME]: This hint specifies the total buffer space that can be used for collective buffering on each target node, usually a multiple of cb_block_size.
- cb_nodes (integer) [SAME]: This hint specifies the number of target nodes to be used for collective buffering.
- chunked (comma separated list of integers) [SAME]: This hint specifies that the file consists of a multidimentional array that is often accessed by subarrays. The value for this hint is a comma separated list of array dimensions, starting from the most significant one (for an array stored in row-major order, as in C, the most significant dimension is the first one; for an array stored in column-major order, as in Fortran, the most significant dimension is the last one, and array dimensions should be reversed).
- chunked_item (comma separated list of integers) [SAME]: This hint specifies the size of each array entry, in bytes.
- chunked_size (comma separated list of integers) [SAME]: This hint specifies the dimensions of the subarrays. This is a comma separated list of array dimensions, starting from the most significant one.
- filename (string): This hint specifies the file name used when the file was opened. If the implementation is capable of returning the file name of an open file, it will be returned using this key by MPI_FILE_GET_INFO. This key is ignored when passed to MPI_FILE_OPEN, MPI_FILE_SET_VIEW, MPI_FILE_SET_INFO, and MPI_FILE_DELETE.
- file_perm (string) [SAME]: This hint specifies the file permissions to use for file creation. Setting this hint is only useful when passed to MPI_FILE_OPEN with an amode that includes MPI_MODE_CREATE. The set of valid values for this key is implementation dependent.

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 2 3	I	. –	rated list of strings) [SAME]: This hint specifies the list of ald be used to store the file. This hint is most relevant when the
4 5 6 7	t		C]: This hint specifies the number of parallel processes that will to run programs that access this file. This hint is most relevant ed.
8 9 10		, .	AME]: This hint specifies the number of I/O devices in the most relevant when the file is created.
11 12			AME]: This hint specifies the number of I/O devices that the across, and is relevant only when the file is created.
13 14 15 16 17 18	fe d	or this file. The stri levice before progres	[ME]: This hint specifies the suggested striping unit to be used ping unit is the amount of consecutive data assigned to one I/O sing to the next device, when striping across a number of devices. es. This hint is relevant only when the file is created.
19 20	13.3	File Views	
21			
22 23	MPI_F	ILE_SET_VIEW(fh,	disp, etype, filetype, datarep, info)
24	INOL	JT fh	file handle (handle)
25	IN	disp	displacement (integer)
26 27	IN	etype	elementary datatype (handle)
28	IN	filetype	filetype (handle)
29	IN	datarep	data representation (string)
30	IN	info	info object (handle)
31 32		init	into object (nandie)
33	C bin	ding	
34	int MF	PI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
35		MPI_Datat	ype filetype, const char *datarep, MPI_Info info)
36	Fortra	an 2008 binding	
37 38	MPI_Fi	ile_set_view(fh,	disp, etype, filetype, datarep, info, ierror)
39		<pre>/PE(MPI_File), IN</pre>	
40			FFSET_KIND), INTENT(IN) :: disp
41		• -	, INTENT(IN) :: etype, filetype
42		PE(MPI_Info), IN	INTENT(IN) :: datarep
43			INTENT(OUT) :: ierror
44			
45 46		an binding	
40		-	DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR) FILETYPE, INFO, IERROR
	11 	······································	

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 13.5.1 for further details.

If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, the special displacement MPI_DISPLACEMENT_CURRENT must be passed in disp. This sets the displacement to the current position of the shared file pointer. MPI_DISPLACEMENT_CURRENT is invalid unless the amode for the file has MPI_MODE_SEQUENTIAL set.

Rationale. For some sequential files, such as those corresponding to magnetic tapes or streaming network connections, the *displacement* may not be meaningful. MPI_DISPLACEMENT_CURRENT allows the view to be changed for these types of files. (*End of rationale.*)

Advice to implementors. It is expected that a call to MPI_FILE_SET_VIEW will immediately follow MPI_FILE_OPEN in numerous instances. A high-quality implementation will ensure that this behavior is efficient. (*End of advice to implementors.*)

The disp displacement argument specifies the position (absolute offset in bytes from the beginning of the file) where the view begins.

Advice to users. disp can be used to skip headers or when the file includes a sequence of data segments that are to be accessed in different patterns (see Figure 13.3). Separate views, each using a different displacement and filetype, can be used to access each segment.

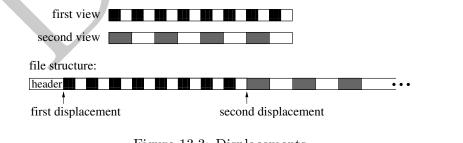


Figure 13.3: Displacements

(End of advice to users.)

An *etype* (*elementary* datatype) is the unit of data access and positioning. It can be any MPI predefined or derived datatype. Derived etypes can be constructed by using any

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $46 \\ 47$

of the MPI datatype constructor routines, provided all resulting typemap displacements are
 non-negative and monotonically nondecreasing. Data access is performed in etype units,
 reading or writing whole data items of type etype. Offsets are expressed as a count of
 etypes; file pointers point to the beginning of etypes.

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

A filetype is either a single etype or a derived MPI datatype constructed from multiple instances of the same etype. In addition, the extent of any hole in the filetype must be a multiple of the etype's extent. These displacements are not required to be distinct, but they cannot be negative, and they must be monotonically nondecreasing.

If the file is opened for writing, neither the etype nor the filetype is permitted to contain overlapping regions. This restriction is equivalent to the "datatype used in a receive cannot specify overlapping regions" restriction for communication. Note that filetypes from different processes may still overlap each other.

If a filetype has holes in it, then the data in the holes is inaccessible to the calling
 process. However, the disp, etype, and filetype arguments can be changed via future calls to
 MPI_FILE_SET_VIEW to access a different part of the file.

It is erroneous to use absolute addresses in the construction of the etype and filetype. The info argument is used to provide information regarding file access patterns and file system specifics to direct optimization (see Section 13.2.8). The constant MPI_INFO_NULL refers to the null info and can be used when no info needs to be specified.

The datarep argument is a string that specifies the representation of data in the file.
 See the file interoperability section (Section 13.5) for details and a discussion of valid values.
 The user is responsible for ensuring that all nonblocking requests and split collective
 operations on fh have been completed before calling MPI_FILE_SET_VIEW — otherwise,
 the call to MPI_FILE_SET_VIEW is erroneous.

30 31

32

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

33	IN	fh	file handle (handle)
34	OUT	disp	displacement (integer)
35 36	OUT	etype	elementary datatype (handle)
37	OUT	filetype	filetype (handle)
38	OUT	datarep	data representation (string)
39			
40	C bindin	g	
41 42	int MPI_H	File_get_view(MPI_File fl	n, MPI_Offset *disp, MPI_Datatype *etype,
42		MPI_Datatype *filet	ype, char *datarep)
44	Fortran 2	2008 binding	
45		Ũ	e, filetype, datarep, ierror)
46		(MPI_File), INTENT(IN) :	
47	INTEG	GER(KIND=MPI_OFFSET_KIND)), INTENT(OUT) :: disp
48	TYPE	(MPI_Datatype), INTENT(OU	JT) :: etype, filetype

Unofficial Draft for Comment Only

5 6

7

8 9

10

11

12

```
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.

13.4 Data Access

13.4.1 Data Access Routines

Data is moved between files and processes by issuing read and write calls. There are three orthogonal aspects to data access: positioning (explicit offset *vs.* implicit file pointer), synchronism (blocking *vs.* nonblocking and split collective), and coordination (noncollective *vs.* collective). The following combinations of these data access routines, including two types of file pointers (individual and shared) are provided in Table 13.1.

positioning	synchronism		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_BEGIN
			MPI_FILE_WRITE_ORDERED_END

Table 13.1: Data access routines

POSIX read()/fread() and write()/fwrite() are blocking, noncollective operations and use individual file pointers. The MPI equivalents are MPI_FILE_READ and

Unofficial Draft for Comment Only

 $\mathbf{2}$

¹ MPI_FILE_WRITE.

Implementations of data access routines may buffer data to improve performance. This
 does not affect reads, as the data is always available in the user's buffer after a read operation
 completes. For writes, however, the MPI_FILE_SYNC routine provides the only guarantee
 that data has been transferred to the storage device.

⁷ Positioning

6

MPI provides three types of positioning for data access routines: **explicit offsets**, **individual file pointers**, and **shared file pointers**. The different positioning methods may be mixed within the same program and do not affect each other.

The data access routines that accept explicit offsets contain _AT in their name (e.g., MPI_FILE_WRITE_AT). Explicit offset operations perform data access at the file position given directly as an argument — no file pointer is used nor updated. Note that this is not equivalent to an atomic seek-and-read or seek-and-write operation, as no "seek" is issued. Operations with explicit offsets are described in Section 13.4.2.

The names of the individual file pointer routines contain no positional qualifier (e.g., MPI_FILE_WRITE). Operations with individual file pointers are described in Section 13.4.3. The data access routines that use shared file pointers contain _SHARED or _ORDERED in their name (e.g., MPI_FILE_WRITE_SHARED). Operations with shared file pointers are described in Section 13.4.4.

The main semantic issues with MPI-maintained file pointers are how and when they are updated by I/O operations. In general, each I/O operation leaves the file pointer pointing to the next data item after the last one that is accessed by the operation. In a nonblocking or split collective operation, the pointer is updated by the call that initiates the I/O, possibly before the access completes.

More formally,

27 28

29

37

38

 $new_file_offset = old_file_offset + \frac{elements(datatype)}{elements(etype)} \times count$

where *count* is the number of *datatype* items to be accessed, elements(X) is the number of predefined datatypes in the typemap of X, and *old_file_offset* is the value of the implicit offset before the call. The file position, new_file_offset , is in terms of a count of etypes relative to the current view.

- 35 36 Synchronism
 - MPI supports blocking and nonblocking I/O routines.
 - A blocking I/O call will not return until the I/O request is completed.

³⁹ A nonblocking I/O call initiates an I/O operation, but does not wait for it to complete. ⁴⁰ Given suitable hardware, this allows the transfer of data out of and into the user's buffer ⁴¹ to proceed concurrently with computation. A separate *request complete* call (MPI_WAIT, ⁴² MPI_TEST, or any of their variants) is needed to complete the I/O request, i.e., to confirm ⁴³ that the data has been read or written and that it is safe for the user to reuse the buffer. ⁴⁴ The nonblocking versions of the routines are named MPI_FILE_IXXX, where the I stands ⁴⁵ for immediate.

It is erroneous to access the local buffer of a nonblocking data access operation, or to
 use that buffer as the source or target of other communications, between the initiation and
 completion of the operation.

The split collective routines support a restricted form of "nonblocking" operations for collective data access (see Section 13.4.5).

Coordination

Every noncollective data access routine MPI_FILE_XXX has a collective counterpart. For most routines, this counterpart is MPI_FILE_XXX_ALL or a pair of MPI_FILE_XXX_BEGIN and MPI_FILE_XXX_END. The counterparts to the MPI_FILE_XXX_SHARED routines are MPI_FILE_XXX_ORDERED.

The completion of a noncollective call only depends on the activity of the calling process. However, the completion of a collective call (which must be called by all members of the process group) may depend on the activity of the other processes participating in the collective call. See Section 13.6.4 for rules on semantics of collective calls.

Collective operations may perform much better than their noncollective counterparts, as global data accesses have significant potential for automatic optimization.

Data Access Conventions

Data is moved between files and processes by calling read and write routines. Read routines move data from a file into memory. Write routines move data from memory into a file. The file is designated by a file handle, fh. The location of the file data is specified by an offset into the current view. The data in memory is specified by a triple: buf, count, and datatype. Upon completion, the amount of data accessed by the calling process is returned in a status.

An offset designates the starting position in the file for an access. The offset is always in etype units relative to the current view. Explicit offset routines pass offset as an argument (negative values are erroneous). The file pointer routines use implicit offsets maintained by MPI.

A data access routine attempts to transfer (read or write) count data items of type datatype between the user's buffer buf and the file. The datatype passed to the routine must be a committed datatype. The layout of data in memory corresponding to buf, count, datatype is interpreted the same way as in MPI communication functions; see Section 3.2.2 and Section 4.1.11. The data is accessed from those parts of the file specified by the current view (Section 13.3). The type signature of datatype must match the type signature of some number of contiguous copies of the etype of the current view. As in a receive, it is erroneous to specify a datatype for reading that contains overlapping regions (areas of memory which would be stored into more than once).

The nonblocking data access routines indicate that MPI can start a data access and associate a request handle, request, with the I/O operation. Nonblocking operations are completed via MPI_TEST, MPI_WAIT, or any of their variants.

Data access operations, when completed, return the amount of data accessed in status.

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 18.1.10–18.1.20. (End of advice to users.)

For blocking routines, status is returned directly. For nonblocking routines and split collective routines, status is returned when the operation is completed. The number of datatype entries and predefined elements accessed by the calling process can be extracted from status by using MPI_GET_COUNT and MPI_GET_ELEMENTS (or

Unofficial Draft for Comment Only

 $\overline{7}$

 24

 31

 $45 \\ 46$

1 2 3 4 5 6 7 8 9 10 11	MPI_GET_ELEMENTS_X), respectively. The interpretation of the MPI_ERROR field is the same as for other operations — normally undefined, but meaningful if an MPI routine returns MPI_ERR_IN_STATUS. The user can pass (in C and Fortran) MPI_STATUS_IGNORE in the status argument if the return value of this argument is not needed. The status can be passed to MPI_TEST_CANCELLED to determine if the operation was cancelled. All other fields of status are undefined. When reading, a program can detect the end of file by noting that the amount of data read is less than the amount requested. Writing past the end of file increases the file size. The amount of data accessed will be the amount requested, unless an error is raised (or a read reaches the end of file).		
12 13	13.4.2 D	Pata Access with Explicit Offs	ets
14 15 16 17	call the ro	utines in this section.	becified when the file was opened, it is erroneous to
18		_READ_AT(fh, offset, buf, cou	,
19 20	IN	fh	file handle (handle)
20		offset	file offset (integer)
22	OUT	buf	initial address of buffer (choice)
23	IN	count	number of elements in buffer (integer)
24 25	IN	datatype	datatype of each buffer element (handle)
26	OUT	status	status object (Status)
27 28 29 30	C bindin int MPI_H	File_read_at(MPI_File fh,	MPI_Offset offset, void *buf, int count, be, MPI_Status *status)
31	Fortran 2	2008 binding	
32 33			count, datatype, status, ierror)
34		<pre>(MPI_File), INTENT(IN) :: GER(KIND=MPI_OFFSET_KIND)</pre>	
35		(*), DIMENSION() :: buf	-
36		GER, INTENT(IN) :: count	
37 38		(MPI_Datatype), INTENT(IN) :: datatype
39		(MPI_Status) :: status GER, OPTIONAL, INTENT(OUT) :: jerror
40			, 101101
41 42	Fortran l	U U	COUNT, DATATYPE, STATUS, IERROR)
42			STATUS(MPI_STATUS_SIZE), IERROR
44		GER(KIND=MPI_OFFSET_KIND)	OFFSET
45	<type< td=""><td>e> BUF(*)</td><td></td></type<>	e> BUF(*)	
46 47	MPI_	$FILE_READ_AT$ reads a file be	eginning at the position specified by offset.
47			

	READ_AT_ALL(fh, offset, buf	count datature status)	1
	fh	file handle (handle)	2
IN			3
IN	offset	file offset (integer)	4
OUT	buf	initial address of buffer (choice)	5 6
IN	count	number of elements in buffer (integer)	7
IN	datatype	datatype of each buffer element (handle)	8
OUT	status	status object (Status)	9
a 1 • 11			10 11
C binding		fh, MPI_Offset offset, void *buf,	12
IIIC MFI_F		ype datatype, MPI_Status *status)	13
Fontnon 9	008 binding		14
	U	ouf, count, datatype, status, ierror)	15 16
	MPI_File), INTENT(IN) ::		17
	ER(KIND=MPI_OFFSET_KIND),	INTENT(IN) :: offset	18
	*), DIMENSION() :: buf		19
	ER, INTENT(IN) :: count MPI_Datatype), INTENT(IN)	·· datatuna	20
	MPI_Status) :: status	uatatype	21 22
	ER, OPTIONAL, INTENT(OUT)	:: ierror	23
Fortran b	inding		24
		BUF, COUNT, DATATYPE, STATUS, IERROR)	25
		TATUS(MPI_STATUS_SIZE), IERROR	26 27
	INTEGER(KIND=MPI_OFFSET_KIND) OFFSET		
<type< td=""><td>> BUF(*)</td><td></td><td>29</td></type<>	> BUF(*)		29
	$TLE_READ_AT_ALL$ is a colle	ctive version of the blocking MPI_FILE_READ_AT	30
interface.			31
			32
MPI_FILE_	WRITE_AT(fh, offset, buf, cou	int, datatype, status)	$33 \\ 34$
INOUT	fh	file handle (handle)	35
IN	offset	file offset (integer)	36
IN	buf	initial address of buffer (choice)	37 38
IN	count	number of elements in buffer (integer)	39
IN	datatype	datatype of each buffer element (handle)	40
OUT	status	status object (Status)	41
001			42 43
C binding	5		43 44
int MPI_F	ile_write_at(MPI_File fh,	MPI_Offset offset, const void *buf,	45
	int count, MPI_Dataty	ype datatype, MPI_Status *status)	46
Fortran 2	008 binding		47
			48

```
588
```

```
1
     MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror)
\mathbf{2}
         TYPE(MPI_File), INTENT(IN) :: fh
3
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
4
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
5
         INTEGER, INTENT(IN) :: count
6
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
7
         TYPE(MPI_Status) :: status
8
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
     Fortran binding
10
     MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
11
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
12
          INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
13
          <type> BUF(*)
14
15
         MPI_FILE_WRITE_AT writes a file beginning at the position specified by offset.
16
17
     MPI_FILE_WRITE_AT_ALL(fh, offset, buf, count, datatype, status)
18
19
       INOUT
                fh
                                            file handle (handle)
20
       IN
                offset
                                            file offset (integer)
21
       IN
                buf
                                            initial address of buffer (choice)
22
23
                                            number of elements in buffer (integer)
       IN
                count
24
                                            datatype of each buffer element (handle)
       IN
                datatype
25
       OUT
                status
                                            status object (Status)
26
27
28
     C binding
     int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf,
29
30
                    int count, MPI_Datatype datatype, MPI_Status *status)
^{31}
     Fortran 2008 binding
32
     MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)
33
         TYPE(MPI_File), INTENT(IN) :: fh
34
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
35
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
36
         INTEGER, INTENT(IN) :: count
37
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
38
         TYPE(MPI_Status) :: status
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     Fortran binding
42
     MPI_FILE_WRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
43
          INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
44
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
45
          <type> BUF(*)
46
         MPI_FILE_WRITE_AT_ALL is a collective version of the blocking
47
     MPI_FILE_WRITE_AT interface.
48
```

MPI_FILE	_IREAD_AT(fh, offset, buf, cou	unt, datatype, request)	1	
IN	fh	file handle (handle)	2	
IN	offset	file offset (integer)	3 4	
OUT	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	
OUT	request	request object (handle)	8 9	
001	request	request object (nandie)	10	
C bindin	g		11	
	•	, MPI_Offset offset, void *buf, int count,	12	
	MPI_Datatype datatyp	e, MPI_Request *request)	13 14	
Fortran 2	2008 binding		15	
		, count, datatype, request, ierror)	16	
	<pre>(MPI_File), INTENT(IN) :: GER(KIND=MPI_OFFSET_KIND)</pre>		17	
	(*), DIMENSION(), ASYNC		18 19	
	GER, INTENT(IN) :: count		20	
	(MPI_Datatype), INTENT(IN		21	
	(MPI_Request), INTENT(OUT		22	
	GER, OPTIONAL, INTENT(OUT) :: lerror	23 24	
Fortran l	0		25	
MPI_FILE_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR) INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR				
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET			27	
<type> BUF(*)</type>			28 29	
MPI_	FILE_IREAD_AT is a nonblock	sing version of the MPI_FILE_READ_AT interface.	30	
			31	
MPL FILF	_IREAD_AT_ALL(fh, offset, bu	if count datatype request)	32	
IN	fh	file handle (handle)	33 34	
IN	offset		35	
		file offset (integer)	36	
OUT	buf	initial address of buffer (choice)	37	
IN	count	number of elements in buffer (integer)	38 39	
IN	datatype	datatype of each buffer element (handle)	40	
OUT	request	request object (handle)	41	
	_		42	
int MDT File impedent all (MDT File fr MDT Offact offact word thuf			43 44	
_ _		ype datatype, MPI_Request *request)	44 45	
Fortran '	2008 binding		46	
		buf, count, datatype, request, ierror)	47	

```
1
         TYPE(MPI_File), INTENT(IN) :: fh
2
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
3
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
4
         INTEGER, INTENT(IN) :: count
5
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
6
         TYPE(MPI_Request), INTENT(OUT) :: request
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
     Fortran binding
9
     MPI_FILE_IREAD_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
10
          INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
11
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
12
          <type> BUF(*)
13
14
         MPI_FILE_IREAD_AT_ALL is a nonblocking version of MPI_FILE_READ_AT_ALL. See
15
     Section 13.6.5 for semantics of nonblocking collective file operations.
16
17
     MPI_FILE_IWRITE_AT(fh, offset, buf, count, datatype, request)
18
19
       INOUT
                fh
                                            file handle (handle)
20
       IN
                offset
                                            file offset (integer)
21
       IN
                buf
                                            initial address of buffer (choice)
22
23
       IN
                count
                                            number of elements in buffer (integer)
24
                                            datatype of each buffer element (handle)
       IN
                datatype
25
                                            request object (handle)
       OUT
                request
26
27
     C binding
28
     int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, const void *buf,
29
30
                    int count, MPI_Datatype datatype, MPI_Request *request)
^{31}
     Fortran 2008 binding
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, 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
     Fortran binding
42
     MPI_FILE_IWRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
43
          INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
44
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
45
         <type> BUF(*)
46
         MPI_FILE_IWRITE_AT is a nonblocking version of the MPI_FILE_WRITE_AT interface.
47
48
```

MPI_FILE	_IWRITE_AT_ALL(fh, offset,	buf, count, datatype, request)	1	
INOUT	fh	file handle (handle)	2	
IN	offset	file offset (integer)	3 4	
IN	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 9	
OUT	request	request object (handle)	9 10	
C bindin	σ		11	
	0	le fh, MPI_Offset offset, const void *buf,	12	
-		type datatype, MPI_Request *request)	13	
Fortran 2	2008 binding		14 15	
	0	, buf, count, datatype, request, ierror)	16	
TYPE	(MPI_File), INTENT(IN) ::	fh	17	
	GER(KIND=MPI_OFFSET_KIND)		18	
	(*), DIMENSION(), INTEN GER, INTENT(IN) :: count	NT(IN), ASYNCHRONOUS :: buf	19	
	(MPI_Datatype), INTENT(IN	I) :: datatvpe	20 21	
	(MPI_Request), INTENT(OUT		22	
INTE	GER, OPTIONAL, INTENT(OUT	C) :: ierror	23	
Fortran l	oinding		24	
	0	, BUF, COUNT, DATATYPE, REQUEST, IERROR)	25	
	GER FH, COUNT, DATATYPE,		26 27	
	GER(KIND=MPI_OFFSET_KIND)	OFFSET	28	
01	e> BUF(*)		29	
MPI_	FILE_IWRITE_AT_ALL is a m	onblocking version of MPI_FILE_WRITE_AT_ALL.	30	
1242 0		ile Deintern	31 32	
13.4.3 L	ata Access with Individual F	lie Pointers	33	
		ter per process per file handle. The current value	34	
-		offset in the data access routines described in this date the individual file pointers maintained by MPI.	35	
	d file pointer is not used nor	· · · · ·	36	
	-	s have the same semantics as the data access with	37 38	
explicit of	fset routines described in Sec	tion $13.4.2$, with the following modification:	39	
		urrent value of the MPI-maintained individual file	40 41	
poin	ter.		42	
		on is initiated, the individual file pointer is updated	43	
-		one that will be accessed. The file pointer is updated	44	
	the current view of the file.	vas specified when the file was opened, it is erroneous	$45 \\ 46$	
		the exception of MPI_FILE_GET_BYTE_OFFSET.	40	
	48			

```
1
     MPI_FILE_READ(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 of each buffer element (handle)
                datatype
7
       OUT
                                            status object (Status)
                status
8
9
     C binding
10
     int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
11
                    MPI_Status *status)
12
13
     Fortran 2008 binding
14
     MPI_File_read(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
     Fortran binding
22
     MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
23
          INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
24
25
          <type> BUF(*)
26
         MPI_FILE_READ reads a file using the individual file pointer.
27
28
     Example 13.2 The following Fortran code fragment is an example of reading a file until
29
     the end of file is reached:
30
31
         Read a preexisting input file until all data has been read.
     Ţ
32
         Call routine "process_input" if all requested data is read.
     !
33
     1
         The Fortran 90 "exit" statement exits the loop.
34
35
                bufsize, numread, totprocessed, status(MPI_STATUS_SIZE)
     integer
     parameter (bufsize=100)
36
37
                localbuffer(bufsize)
     real
38
     integer (kind=MPI_OFFSET_KIND) zero
39
40
     zero = 0
41
42
     call MPI_FILE_OPEN(MPI_COMM_WORLD, 'myoldfile', &
43
                          MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr)
44
     call MPI_FILE_SET_VIEW(myfh, zero, MPI_REAL, MPI_REAL, 'native', &
45
                              MPI_INFO_NULL, ierr)
46
     totprocessed = 0
47
     do
48
        call MPI_FILE_READ(myfh, localbuffer, bufsize, MPI_REAL, &
```

```
1
                        status, ierr)
                                                                                         \mathbf{2}
   call MPI_GET_COUNT(status, MPI_REAL, numread, ierr)
   call process_input(localbuffer, numread)
   totprocessed = totprocessed + numread
                                                                                         4
   if (numread < bufsize) exit
                                                                                         5
                                                                                         6
end do
write(6, 1001) numread, bufsize, totprocessed
1001 format("No more data: read", I3, "and expected", I3, &
                                                                                         9
                                                                                         10
              "Processed total of", I6, "before terminating job.")
                                                                                         11
call MPI_FILE_CLOSE(myfh, ierr)
                                                                                         12
                                                                                         13
                                                                                         14
                                                                                         15
MPI_FILE_READ_ALL(fh, buf, count, datatype, status)
                                                                                         16
 INOUT
           fh
                                       file handle (handle)
                                                                                         17
                                                                                         18
 OUT
           buf
                                       initial address of buffer (choice)
                                                                                         19
 IN
           count
                                       number of elements in buffer (integer)
                                                                                        20
                                      datatype of each buffer element (handle)
 IN
           datatype
                                                                                        21
                                                                                        22
 OUT
                                       status object (Status)
           status
                                                                                        23
                                                                                         ^{24}
C binding
                                                                                         25
int MPI_File_read_all(MPI_File fh, void *buf, int count,
                                                                                         26
              MPI_Datatype datatype, MPI_Status *status)
                                                                                        27
Fortran 2008 binding
                                                                                        28
MPI_File_read_all(fh, buf, count, datatype, status, ierror)
                                                                                        29
                                                                                        30
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
                                                                                         31
    INTEGER, INTENT(IN) :: count
                                                                                         32
                                                                                         33
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                        34
   TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                        35
                                                                                        36
Fortran binding
                                                                                        37
MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
                                                                                        38
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
                                                                                        39
    <type> BUF(*)
                                                                                         40
                                                                                        41
    MPI_FILE_READ_ALL is a collective version of the blocking MPI_FILE_READ interface.
                                                                                        42
                                                                                        43
                                                                                         44
                                                                                         45
                                                                                         46
                                                                                         47
                                                                                         48
```

```
1
     MPI_FILE_WRITE(fh, buf, count, datatype, status)
2
       INOUT
                 fh
                                             file handle (handle)
3
       IN
                 buf
                                             initial address of buffer (choice)
4
5
       IN
                 count
                                             number of elements in buffer (integer)
6
       IN
                                             datatype of each buffer element (handle)
                 datatype
7
       OUT
                 status
                                             status object (Status)
8
9
     C binding
10
11
     int MPI_File_write(MPI_File fh, const void *buf, int count,
                    MPI_Datatype datatype, MPI_Status *status)
12
13
     Fortran 2008 binding
14
     MPI_File_write(fh, buf, count, datatype, status, ierror)
15
          TYPE(MPI_File), INTENT(IN) :: fh
16
          TYPE(*), DIMENSION(...), INTENT(IN) :: 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
22
     Fortran binding
23
     MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
24
          INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
25
          <type> BUF(*)
26
          MPI_FILE_WRITE writes a file using the individual file pointer.
27
28
29
     MPI_FILE_WRITE_ALL(fh, buf, count, datatype, status)
30
       INOUT
                 fh
                                             file handle (handle)
^{31}
       IN
                 buf
                                             initial address of buffer (choice)
32
33
       IN
                                             number of elements in buffer (integer)
                 count
34
       IN
                 datatype
                                             datatype of each buffer element (handle)
35
       OUT
                                             status object (Status)
                 status
36
37
38
     C binding
39
     int MPI_File_write_all(MPI_File fh, const void *buf, int count,
40
                    MPI_Datatype datatype, MPI_Status *status)
41
     Fortran 2008 binding
42
     MPI_File_write_all(fh, buf, count, datatype, status, ierror)
43
          TYPE(MPI_File), INTENT(IN) :: fh
44
          TYPE(*), DIMENSION(..), INTENT(IN) :: buf
45
          INTEGER, INTENT(IN) :: count
46
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
47
          TYPE(MPI_Status) :: status
48
```

```
1
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                        2
Fortran binding
MPI_FILE_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    <type> BUF(*)
                                                                                        6
    MPI_FILE_WRITE_ALL is a collective version of the blocking MPI_FILE_WRITE inter-
                                                                                        7
                                                                                        8
face.
                                                                                        9
                                                                                        10
MPI_FILE_IREAD(fh, buf, count, datatype, request)
                                                                                        11
 INOUT
                                                                                        12
                                      file handle (handle)
           fh
                                                                                        13
 OUT
           buf
                                      initial address of buffer (choice)
                                                                                        14
 IN
           count
                                      number of elements in buffer (integer)
                                                                                        15
                                                                                        16
                                      datatype of each buffer element (handle)
 IN
           datatype
                                                                                        17
 OUT
                                      request object (handle)
           request
                                                                                        18
                                                                                        19
C binding
                                                                                        20
int MPI_File_iread(MPI_File fh, void *buf, int count,
                                                                                       21
              MPI_Datatype datatype, MPI_Request *request)
                                                                                        22
                                                                                        23
Fortran 2008 binding
                                                                                        24
MPI_File_iread(fh, buf, count, datatype, request, ierror)
                                                                                        25
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                        26
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                        27
    INTEGER, INTENT(IN) :: count
                                                                                        28
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                        29
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                        30
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                        31
Fortran binding
                                                                                        32
MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
                                                                                        33
   INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
                                                                                       34
    <type> BUF(*)
                                                                                       35
                                                                                       36
    MPI_FILE_IREAD is a nonblocking version of the MPI_FILE_READ interface.
                                                                                       37
Example 13.3 The following Fortran code fragment illustrates file pointer update seman-
                                                                                       38
                                                                                        39
tics:
                                                                                        40
    Read the first twenty real words in a file into two local
                                                                                        41
I.
!
    buffers. Note that when the first MPI_FILE_IREAD returns,
                                                                                        42
!
    the file pointer has been updated to point to the
                                                                                        43
    eleventh real word in the file.
!
                                                                                        44
                                                                                        45
          bufsize, req1, req2
integer
                                                                                        46
integer, dimension(MPI_STATUS_SIZE) :: status1, status2
                                                                                        47
parameter (bufsize=10)
                                                                                        48
```

```
1
     real
                buf1(bufsize), buf2(bufsize)
\mathbf{2}
     integer (kind=MPI_OFFSET_KIND) zero
3
4
     zero = 0
5
     call MPI_FILE_OPEN(MPI_COMM_WORLD, 'myoldfile', &
6
                          MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr)
7
     call MPI_FILE_SET_VIEW(myfh, zero, MPI_REAL, MPI_REAL, 'native', &
8
                               MPI_INFO_NULL, ierr)
9
     call MPI_FILE_IREAD(myfh, buf1, bufsize, MPI_REAL, &
10
                           req1, ierr)
^{11}
     call MPI_FILE_IREAD(myfh, buf2, bufsize, MPI_REAL, &
12
                           req2, ierr)
13
14
     call MPI_WAIT(req1, status1, ierr)
15
     call MPI_WAIT(req2, status2, ierr)
16
17
     call MPI_FILE_CLOSE(myfh, ierr)
18
19
20
     MPI_FILE_IREAD_ALL(fh, buf, count, datatype, request)
21
                                            file handle (handle)
22
       INOUT
                fh
23
       OUT
                                            initial address of buffer (choice)
                buf
24
       IN
                                            number of elements in buffer (integer)
                count
25
26
       IN
                                            datatype of each buffer element (handle)
                datatype
27
       OUT
                request
                                            request object (handle)
28
29
     C binding
30
     int MPI_File_iread_all(MPI_File fh, void *buf, int count,
^{31}
                    MPI_Datatype datatype, MPI_Request *request)
32
33
     Fortran 2008 binding
34
     MPI_File_iread_all(fh, buf, count, datatype, request, ierror)
35
         TYPE(MPI_File), INTENT(IN) :: fh
36
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
37
         INTEGER, INTENT(IN) :: count
38
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
39
         TYPE(MPI_Request), INTENT(OUT) :: request
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
     Fortran binding
42
     MPI_FILE_IREAD_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
43
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
44
          <type> BUF(*)
45
46
         MPI_FILE_IREAD_ALL is a nonblocking version of MPI_FILE_READ_ALL.
47
48
```

	_IWRITE(fh, buf, count, dat		1 2
INOUT	fh	file handle (handle)	3
IN	buf	initial address of buffer (choice)	4
IN	count	number of elements in buffer (integer)	5
IN	datatype	datatype of each buffer element (handle)	6 7
OUT	request	request object (handle)	8
			9
C bindin	•		10
int MPI_B		, const void *buf, int count, ype, MPI_Request *request)	11 12
T (ype, in i_nequest wrequest)	13
	2008 binding	datatype, request, ierror)	14
	(MPI_File), INTENT(IN)		15
	-	ENT(IN), ASYNCHRONOUS :: buf	16 17
	GER, INTENT(IN) :: coun		18
	(MPI_Datatype), INTENT((MPI_Request), INTENT(O		19
	GER, OPTIONAL, INTENT(O	-	20
			21 22
Fortran b		DATATYPE, REQUEST, IERROR)	22
	GER FH, COUNT, DATATYPE		24
<type< td=""><td>e> BUF(*)</td><td></td><td>25</td></type<>	e> BUF(*)		25
MPI_	FILE_IWRITE is a nonblock	ing version of the MPI_FILE_WRITE interface.	26
			27 28
MPL FILE	_IWRITE_ALL(fh, buf, coun	t datatype request)	29
INOUT	(,) fh	file handle (handle)	30
			31
IN	buf	initial address of buffer (choice)	32 33
IN	count	number of elements in buffer (integer)	34
IN	datatype	datatype of each buffer element (handle)	35
OUT	request	request object (handle)	36
			37 38
C bindin	•	e fh, const void *buf, int count,	39
		ype, MPI_Request *request)	40
Fortran 2	2008 binding		41
		unt, datatype, request, ierror)	42 43
TYPE	(MPI_File), INTENT(IN)	:: fh	43
		ENT(IN), ASYNCHRONOUS :: buf	45
	GER, INTENT(IN) :: coun (MPI_Datatype), INTENT(46
	(MPI_Datatype), INTENI() (MPI_Request), INTENT()		47
			48

```
1
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     Fortran binding
3
     MPI_FILE_IWRITE_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
4
          INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
5
          <type> BUF(*)
6
7
          MPI_FILE_IWRITE_ALL is a nonblocking version of MPI_FILE_WRITE_ALL.
8
9
     MPI_FILE_SEEK(fh, offset, whence)
10
11
       INOUT
                 fh
                                              file handle (handle)
12
       IN
                 offset
                                              file offset (integer)
13
       IN
                 whence
                                              update mode (state)
14
15
16
     C binding
17
     int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
18
     Fortran 2008 binding
19
     MPI_File_seek(fh, offset, whence, ierror)
20
          TYPE(MPI_File), INTENT(IN) :: fh
21
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
22
          INTEGER, INTENT(IN) :: whence
23
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
^{24}
25
     Fortran binding
26
     MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)
27
          INTEGER FH, WHENCE, IERROR
28
          INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
29
          MPI_FILE_SEEK updates the individual file pointer according to whence, which has the
30
     following possible values:
^{31}
32
         • MPI_SEEK_SET: the pointer is set to offset
33
         • MPI_SEEK_CUR: the pointer is set to the current pointer position plus offset
34
35
         • MPI_SEEK_END: the pointer is set to the end of file plus offset
36
37
          The offset can be negative, which allows seeking backwards. It is erroneous to seek to
     a negative position in the view.
38
39
40
     MPI_FILE_GET_POSITION(fh, offset)
41
42
       IN
                 fh
                                              file handle (handle)
43
       OUT
                 offset
                                              offset of individual pointer (integer)
44
45
     C binding
46
     int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)
47
48
     Fortran 2008 binding
```

MPI_File_get_position(fh, offset, ierror)	1
TYPE(MPI_File), INTENT(IN) :: fh	2
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset	3
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	4
Fortran binding	5
MPI_FILE_GET_POSITION(FH, OFFSET, IERROR)	6
INTEGER FH, IERROR	7
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	8 9
	9 10
MPI_FILE_GET_POSITION returns, in offset, the current position of the individual file	10
pointer in etype units relative to the current view.	12
Advice to users. The offset can be used in a future call to MPI_FILE_SEEK using	13
whence = MPI_SEEK_SET to return to the current position. To set the displacement to	14
the current file pointer position, first convert offset into an absolute byte position using	15
MPI_FILE_GET_BYTE_OFFSET, then call MPI_FILE_SET_VIEW with the resulting	16
displacement. (End of advice to users.)	17
	18
	19
MPI_FILE_GET_BYTE_OFFSET(fh, offset, disp)	20
	21
IN fh file handle (handle)	22
IN offset offset (integer)	23
OUT disp absolute byte position of offset (integer)	24
	25 26
C binding	20
int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset,	28
MPI_Offset *disp)	29
	30
Fortran 2008 binding	31
<pre>MPI_File_get_byte_offset(fh, offset, disp, ierror) TYPE(MPI_File) INTENT(IN) fh</pre>	32
TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset	33
INTEGER(KIND=MPI_OFFSET_KIND), INTENI(IN) .: OFFSET INTEGER(KIND=MPI_OFFSET_KIND), INTENI(OUT) :: disp	34
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	35
	36
Fortran binding	37
MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR)	38
INTEGER FH, IERROR	39
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP	40
$MPI_FILE_GET_BYTE_OFFSET$ converts a view-relative offset into an absolute byte	41 42
position. The absolute byte position (from the beginning of the file) of offset relative to the	42 43
current view of fh is returned in disp .	43

13.4.4 Data Access with Shared File Pointers

MPI maintains exactly one shared file pointer per collective MPI_FILE_OPEN (shared among processes in the communicator group). The current value of this pointer implicitly specifies 48

Unofficial Draft for Comment Only

44

1 2 3		e shared file pointer	outines described in this section. These routines only use and maintained by MPI. The individual file pointers are not used
4	The s	hared file pointer ro	utines have the same semantics as the data access with explicit ection $13.4.2$, with the following modifications:
6 7	• the c	offset is defined to b	be the current value of the MPI-maintained shared file pointer,
8 9		effect of multiple ca were serialized, an	lls to shared file pointer routines is defined to behave as if the d
10 11 12	• the file v	-	ointer routines is erroneous unless all processes use the same
13 14 15 16 17 18	istic. The After point to th	user needs to use of a shared file pointe	ile pointer routines, the serialization ordering is not determin- ther synchronization means to enforce a specific order. er operation is initiated, the shared file pointer is updated to the last one that will be accessed. The file pointer is updated f the file.
19 20	Noncollect	ive Operations	
21 22			
23	MPI_FILE	_READ_SHARED(fl	n, buf, count, datatype, status)
24	INOUT	fh	file handle (handle)
25 26	OUT	buf	initial address of buffer (choice)
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 33 34	C bindin int MPI_H	File_read_shared	(MPI_File fh, void *buf, int count, e datatype, MPI_Status *status)
35		2008 binding	
36 37			buf, count, datatype, status, ierror)
38		(MPI_File), INTE (*), DIMENSION(.	
39		GER, INTENT(IN)	
40			INTENT(IN) :: datatype
41 42		(MPI_Status) :: :	status NTENT(OUT) :: ierror
43			
44	Fortran l		BUF, COUNT, DATATYPE, STATUS, IERROR)
45 46	INTEG	GER FH, COUNT, D.	ATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
47 48	01	e> BUF(*) FILE READ SHARI	ED reads a file using the shared file pointer.
			reads a me doing the onared me pointer.

MPI_FILE_WRITE_SHARED(fh, buf, count, datatype, status)			
INOUT	fh	file handle (handle)	2
IN	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	status	status object (Status)	7 8
001			9
C binding	5		10
int MPI_F	ile_write_shared(MPI_File	e fh, const void *buf, int count,	11
	MPI_Datatype datatype	e, MPI_Status *status)	12
Fortran 2	008 binding		13 14
		nt, datatype, status, ierror)	14
	<pre>MPI_File), INTENT(IN) ::</pre>		16
	*), DIMENSION(), INTENT ER, INTENT(IN) :: count	(IN) :: DUI	17
	MPI_Datatype), INTENT(IN)	:: datatype	18
	MPI_Status) :: status		19 20
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	20 21
Fortran b	inding		22
	5	INT, DATATYPE, STATUS, IERROR)	23
		STATUS(MPI_STATUS_SIZE), IERROR	24
<type< td=""><td>> BUF(*)</td><td></td><td>25</td></type<>	> BUF(*)		25
MPI_F	ILE_WRITE_SHARED writes	a file using the shared file pointer.	26 27
			28
MPI_FILE_	IREAD_SHARED(fh, buf, cour	nt, datatype, request)	29
INOUT	fh	file handle (handle)	30 31
OUT	buf	initial address of buffer (choice)	32
IN	count	number of elements in buffer (integer)	33
IN	datatype	datatype of each buffer element (handle)	34
OUT	request	request object (handle)	35 36
001	lequest	request object (nanue)	37
C binding	,		38
		e fh, void *buf, int count,	39
	MPI_Datatype datatype	e, MPI_Request *request)	40
Fortran 2	008 binding		41
	0	nt, datatype, request, ierror)	42 43
TYPE(MPI_File), INTENT(IN) :: fh			
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf			
INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype			
	MPI_Datatype), INTENT(IN) MPI_Request), INTENT(OUT)		47
IIFE()	In I_Request, INIENI(001)		48

```
602
```

```
1
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
2
     Fortran binding
3
     MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
4
          INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
5
          <type> BUF(*)
6
7
          MPI_FILE_IREAD_SHARED is a nonblocking version of the MPI_FILE_READ_SHARED
8
     interface.
9
10
     MPI_FILE_IWRITE_SHARED(fh, buf, count, datatype, request)
11
12
       INOUT
                 fh
                                              file handle (handle)
13
       IN
                 buf
                                              initial address of buffer (choice)
14
       IN
                 count
                                              number of elements in buffer (integer)
15
16
                                              datatype of each buffer element (handle)
       IN
                 datatype
17
       OUT
                 request
                                              request object (handle)
18
19
     C binding
20
     int MPI_File_iwrite_shared(MPI_File fh, const void *buf, int count,
21
                     MPI_Datatype datatype, MPI_Request *request)
22
23
     Fortran 2008 binding
^{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, INTENT(IN) :: count
28
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
          TYPE(MPI_Request), INTENT(OUT) :: request
30
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
     Fortran binding
32
     MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
33
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
34
          <type> BUF(*)
35
36
          MPI_FILE_IWRITE_SHARED is a nonblocking version of the
37
     MPI_FILE_WRITE_SHARED interface.
38
39
     Collective Operations
40
     The semantics of a collective access using a shared file pointer is that the accesses to the
41
42
     file will be in the order determined by the ranks of the processes within the group. For each
     process, the location in the file at which data is accessed is the position at which the shared
43
     file pointer would be after all processes whose ranks within the group less than that of this
44
     process had accessed their data. In addition, in order to prevent subsequent shared offset
45
     accesses by the same processes from interfering with this collective access, the call might
46
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,	count, datatype, status)		
INOUT fh	file handle (handle)	18 19	
OUT buf	initial address of buffer (choice)	20	
OOT DUI	linuar address of builer (choice)	21	
IN count	number of elements in buffer (integer)	21	
IN datatype	datatype of each buffer element (handle)	23	
OUT status	status object (Status)	24	
		25	
C binding		26	
int MPI_File_read_ordered(MPI_File fh, void *buf, int count,			
	ype, MPI_Status *status)	28	
		29	
Fortran 2008 binding		30	
	count, datatype, status, ierror)	31	
TYPE(MPI_File), INTENT(IN)		32	
TYPE(*), DIMENSION() :: buf			
INTEGER, INTENT(IN) :: count		34	
TYPE(MPI_Datatype), INTENT()	IN) :: datatype	35	
TYPE(MPI_Status) :: status		36	
INTEGER, OPTIONAL, INTENT(O	UT) :: ierror	37	
Fortran binding		38	
C	COUNT, DATATYPE, STATUS, IERROR)	39	
	, STATUS(MPI_STATUS_SIZE), IERROR	40	
<pre><type> BUF(*)</type></pre>	, DIATOD (III 1_DIATOD_DIZD), IDIATOIC	41	
		42	
MPI_FILE_READ_ORDERED is a	a collective version of the MPI_FILE_READ_SHARED	43	
interface.		44	
		45	
		46	
		47	

```
1
     MPI_FILE_WRITE_ORDERED(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 of each buffer element (handle)
                datatype
7
       OUT
                status
                                            status object (Status)
8
9
     C binding
10
11
     int MPI_File_write_ordered(MPI_File fh, const void *buf, int count,
                    MPI_Datatype datatype, MPI_Status *status)
12
13
     Fortran 2008 binding
14
     MPI_File_write_ordered(fh, buf, count, datatype, status, ierror)
15
          TYPE(MPI_File), INTENT(IN) :: fh
16
          TYPE(*), DIMENSION(...), INTENT(IN) :: 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
22
     Fortran binding
23
     MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
24
          INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
25
          <type> BUF(*)
26
          MPI_FILE_WRITE_ORDERED is a collective version of the MPI_FILE_WRITE_SHARED
27
     interface.
28
29
     Seek
30
^{31}
     If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous
32
     to call the following two routines (MPI_FILE_SEEK_SHARED and
33
     MPI_FILE_GET_POSITION_SHARED).
34
35
     MPI_FILE_SEEK_SHARED(fh, offset, whence)
36
37
       INOUT
                fh
                                            file handle (handle)
38
       IN
                 offset
                                            file offset (integer)
39
       IN
                whence
                                            update mode (state)
40
41
42
     C binding
43
     int MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence)
44
     Fortran 2008 binding
45
     MPI_File_seek_shared(fh, offset, whence, ierror)
46
          TYPE(MPI_File), INTENT(IN) :: fh
47
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
48
```

INTEGER, INTENT(IN) :: whence INTEGER, OPTIONAL, INTENT(OUT) :: ierror	$\frac{1}{2}$		
Fortran binding			
MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR)			
INTEGER FH, WHENCE, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	6		
INIEGER(KIND-MFI_OFFSEI_KIND) OFFSEI	7		
MPI_FILE_SEEK_SHARED updates the shared file pointer according to whence, which has the following possible values:			
• MPI_SEEK_SET: the pointer is set to offset			
• MPI_SEEK_CUR: the pointer is set to the current pointer position plus offset	12 13		
• MPI_SEEK_END: the pointer is set to the end of file plus offset	14 15		
MPI_FILE_SEEK_SHARED is collective; all the processes in the communicator group	16		
associated with the file handle fh must call MPI_FILE_SEEK_SHARED with the same values	17		
for offset and whence.	18		
The offset can be negative, which allows seeking backwards. It is erroneous to seek to	19		
a negative position in the view.	20		
	21		
MPI_FILE_GET_POSITION_SHARED(fh, offset)	22		
, , , , , , , , , , , , , , , , , , ,	23		
IN fh file handle (handle)	24		
OUToffsetoffset of shared pointer (integer)	25		
	26		
C binding	27		
<pre>int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)</pre>	28 29		
Fortran 2008 binding	30		
MPI_File_get_position_shared(fh, offset, ierror)	31		
TYPE(MPI_File), INTENT(IN) :: fh	32		
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset	33		
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	34		
Fortran binding	35		
MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERROR)	36		
INTEGER FH, IERROR	37		
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	38		
	39		
MPI_FILE_GET_POSITION_SHARED returns, in offset, the current position of the	40		
shared file pointer in etype units relative to the current view.	41		
Advice to users. The offset can be used in a future call to MPI_FILE_SEEK_SHARED	42		
$Aabice to users$. The onset can be used in a future can to MPI_FILE_SEEK_SHARED using whence = MPI_SEEK_SET to return to the current position. To set the displace-	43 44		
ment to the current file pointer position, first convert offset into an absolute byte	44 45		
position using MPI_FILE_GET_BYTE_OFFSET, then call MPI_FILE_SET_VIEW with	46		

47 48

the resulting displacement. (End of advice to users.)

1 13.4.5 Split Collective Data Access Routines 2 MPI provides a restricted form of "nonblocking collective" I/O operations for all data ac-3 cesses using split collective data access routines. These routines are referred to as "split" 4 collective routines because a single collective operation is split in two: a begin routine and 5an end routine. The begin routine begins the operation, much like a nonblocking data access 6 (e.g., MPI_FILE_IREAD). The end routine completes the operation, much like the matching 7 test or wait (e.g., MPI_WAIT). As with nonblocking data access operations, the user must 8 not use the buffer passed to a begin routine while the routine is outstanding; the operation 9 must be completed with an end routine before it is safe to free buffers, etc. 10 Split collective data access operations on a file handle fh are subject to the semantic 11 rules given below. 1213 • On any MPI process, each file handle may have at most one active split collective 14operation at any time. 1516• Begin calls are collective over the group of processes that participated in the collective 17 open and follow the ordering rules for collective calls. 18 • End calls are collective over the group of processes that participated in the collective 19 open and follow the ordering rules for collective calls. Each end call matches the 20preceding begin call for the same collective operation. When an "end" call is made, 21exactly one unmatched "begin" call for the same operation must precede it. 2223• An implementation is free to implement any split collective data access routine using 24the corresponding blocking collective routine when either the begin call (e.g., 25MPI_FILE_READ_ALL_BEGIN) or the end call (e.g., MPI_FILE_READ_ALL_END) is 26issued. The begin and end calls are provided to allow the user and MPI implementation 27to optimize the collective operation. 2829 • Split collective operations do not match the corresponding regular collective opera-30 tion. For example, in a single collective read operation, an MPI_FILE_READ_ALL 31on one process does not match an MPI_FILE_READ_ALL_BEGIN/ 32 MPI_FILE_READ_ALL_END pair on another process. 33 • Split collective routines must specify a buffer in both the begin and end routines. 34 By specifying the buffer that receives data in the end routine, we can avoid the 35problems described in "A Problem with Code Movements and Register Optimization," 36 Section 18.1.17, but not all of the problems, such as those described in Sections 18.1.12, 37 18.1.13, and 18.1.16. 38 39 • No collective I/O operations are permitted on a file handle concurrently with a split 40 collective access on that file handle (i.e., between the begin and end of the access). 41 That is 4243 MPI_File_read_all_begin(fh, ...); 4445MPI_File_read_all(fh, ...); 46 . . . 47 MPI_File_read_all_end(fh, ...); 48

is erroneous.

• In a multithreaded implementation, any split collective begin and end operation called by a process must be called from the same thread. This restriction is made to simplify the implementation in the multithreaded case. (Note that we have already disallowed having two threads begin a split collective operation on the same file handle since only one split collective operation can be active on a file handle at any time.)

The arguments for these routines have the same meaning as for the equivalent collective versions (e.g., the argument definitions for MPI_FILE_READ_ALL_BEGIN and MPI_FILE_READ_ALL_END are equivalent to the arguments for MPI_FILE_READ_ALL). The begin routine (e.g., MPI_FILE_READ_ALL_BEGIN) begins a split collective operation that, when completed with the matching end routine (i.e., MPI_FILE_READ_ALL_END) produces the result as defined for the equivalent collective routine (i.e., MPI_FILE_READ_ALL).

For the purpose of consistency semantics (Section 13.6.1), a matched pair of split collective data access operations (e.g., MPI_FILE_READ_ALL_BEGIN and MPI_FILE_READ_ALL_END) compose a single data access.

MPI_FILE_READ_AT_ALL_BEGIN(fh, offset, buf, count, datatype)

IN	fh	file handle (handle)	21 22
IN	offset	file offset (integer)	23
OUT	buf	initial address of buffer (choice)	24
IN	count	number of elements in buffer (integer)	25 26
IN	datatype	datatype of each buffer element (handle)	27
			28
C binding			
int MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,			30
int count, MPI_Datatype datatype)			31
			32
Fortran 2008 binding			33
<pre>MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror)</pre>			34
TYPE(MPI_File), INTENT(IN) :: fh			
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset			36
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf			37
INTEGER, INTENT(IN) :: count			
TYPE(MPI_Datatype), INTENT(IN) :: datatype			

Fortran binding

MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
INTEGER FH, COUNT, DATATYPE, IERROR
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
<type> BUF(*)

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

 $\mathbf{2}$

```
1
     MPI_FILE_READ_AT_ALL_END(fh, buf, status)
2
       IN
                fh
                                            file handle (handle)
3
       OUT
                 buf
                                            initial address of buffer (choice)
4
5
       OUT
                                            status object (Status)
                status
6
7
     C binding
8
     int MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)
9
     Fortran 2008 binding
10
     MPI_File_read_at_all_end(fh, buf, status, ierror)
11
          TYPE(MPI_File), INTENT(IN) :: fh
12
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
13
          TYPE(MPI_Status) :: status
14
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     Fortran binding
17
     MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)
18
          INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
19
          <type> BUF(*)
20
21
22
     MPI_FILE_WRITE_AT_ALL_BEGIN(fh, offset, buf, count, datatype)
23
       INOUT
                                            file handle (handle)
^{24}
                fh
25
       IN
                offset
                                            file offset (integer)
26
       IN
                 buf
                                            initial address of buffer (choice)
27
       IN
                                            number of elements in buffer (integer)
                count
28
29
       IN
                datatype
                                            datatype of each buffer element (handle)
30
^{31}
     C binding
32
     int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset,
33
                    const void *buf, int count, MPI_Datatype datatype)
34
     Fortran 2008 binding
35
     MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror)
36
37
          TYPE(MPI_File), INTENT(IN) :: fh
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
38
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
39
40
          INTEGER, INTENT(IN) :: count
41
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
42
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
     Fortran binding
44
     MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
45
          INTEGER FH, COUNT, DATATYPE, IERROR
46
          INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
47
          <type> BUF(*)
48
```

MPI_FILE	_WRITE_AT_ALL	_END(fh, buf, status)	1	
INOUT	fh	file handle (handle)	2	
IN	buf	initial address of buffer (choice)	3 4	
OUT	status	status object (Status)	5	
			6	
C bindin	g		7	
int MPI_H		<pre>ll_end(MPI_File fh, const void *buf,</pre>	8 9	
	MPI_Status	*status)	10	
Fortran 2	2008 binding		11	
MPI_File_write_at_all_end(fh, buf, status, ierror)				
	(MPI_File), INT		13	
	(*), DIMENSION((MPI_Status) ::), INTENT(IN), ASYNCHRONOUS :: buf	14	
		INTENT(OUT) :: ierror	15 16	
			17	
Fortran l			18	
		ND(FH, BUF, STATUS, IERROR) MPI_STATUS_SIZE), IERROR	19	
	e> BUF(*)	III_DIRIOD_DIZE/, IERON	20	
51			21	
			22 23	
MPI_FILE	_READ_ALL_BEG	IN(fh, buf, count, datatype)	23 24	
INOUT	fh	file handle (handle)	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	
Chindin			30	
C bindin	9	egin(MPI_File fh, void *buf, int count,	32	
int mi_i		pe datatype)	33	
			34	
	2008 binding	(fh, buf, count, datatype, ierror)	35	
	(MPI_File), INT		36	
), ASYNCHRONOUS :: buf	37 38	
INTEGER, INTENT(IN) :: count				
TYPE	(MPI_Datatype),	INTENT(IN) :: datatype	40	
INTEG	GER, OPTIONAL, I	INTENT(OUT) :: ierror	41	
Fortran l	binding		42	
	•	(FH, BUF, COUNT, DATATYPE, IERROR)	43	
INTEGER FH, COUNT, DATATYPE, IERROR				
<type< td=""><td>e> BUF(*)</td><td></td><td>45 46</td></type<>	e> BUF(*)		45 46	
			40 47	
			48	

```
1
     MPI_FILE_READ_ALL_END(fh, buf, status)
2
       INOUT
                 fh
                                            file handle (handle)
3
       OUT
                 buf
                                            initial address of buffer (choice)
4
5
       OUT
                                            status object (Status)
                 status
6
7
     C binding
8
     int MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status)
9
     Fortran 2008 binding
10
     MPI_File_read_all_end(fh, buf, status, ierror)
11
          TYPE(MPI_File), INTENT(IN) :: fh
12
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
13
          TYPE(MPI_Status) :: status
14
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     Fortran binding
17
     MPI_FILE_READ_ALL_END(FH, BUF, STATUS, IERROR)
18
          INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
19
          <type> BUF(*)
20
21
22
     MPI_FILE_WRITE_ALL_BEGIN(fh, buf, count, datatype)
23
       INOUT
                                            file handle (handle)
^{24}
                 fh
25
                 buf
       IN
                                            initial address of buffer (choice)
26
       IN
                                            number of elements in buffer (integer)
                 count
27
       IN
                                            datatype of each buffer element (handle)
                 datatype
28
29
30
     C binding
^{31}
     int MPI_File_write_all_begin(MPI_File fh, const void *buf, int count,
32
                    MPI_Datatype datatype)
33
     Fortran 2008 binding
34
     MPI_File_write_all_begin(fh, buf, count, datatype, ierror)
35
          TYPE(MPI_File), INTENT(IN) :: fh
36
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
37
          INTEGER, INTENT(IN) :: count
38
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
39
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
^{41}
     Fortran binding
42
     MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
43
          INTEGER FH, COUNT, DATATYPE, IERROR
44
          <type> BUF(*)
45
46
47
48
```

MPI_FILE_WRITE_ALL_END(fh, buf, status) ¹			
INOUT	fh	file handle (handle)	2
IN	buf	initial address of buffer (choice)	3 4
OUT	status	status object (Status)	5
			6
C binding	g		7
int MPI_F	File_write_all_end(MPI_Fil	le fh, const void *buf,	8
	MPI_Status *status)		9 10
Fortran 2	2008 binding		11
MPI_File_	write_all_end(fh, buf, st	catus, ierror)	12
	(MPI_File), INTENT(IN) ::		13
	(*), DIMENSION(), INTEN (MPI_Status) :: status	I(IN), ASYNCHRONOUS :: buf	14
	ER, OPTIONAL, INTENT(OUT)	:: jerror	15 16
			17
Fortran b			18
	_WRITE_ALL_END(FH, BUF, ST GER FH, STATUS(MPI_STATUS_		19
	e> BUF(*)		20
• -			21 22
			22
MPI_FILE	_READ_ORDERED_BEGIN(fh,	buf, count, datatype)	24
INOUT	fh	file handle (handle)	25
OUT	buf	initial address of buffer (choice)	26
IN	count	number of elements in buffer (integer)	27
IN	datatype	datatype of each buffer element (handle)	28 29
	uatatype	datatype of each builer element (nandle)	30
C binding	g		31
		PI_File fh, void *buf, int count,	32
	MPI_Datatype datatyp	e)	33
Fortran 2	2008 binding		34
		ıf, count, datatype, ierror)	35 36
TYPE((MPI_File), INTENT(IN) ::	fh	37
	(*), DIMENSION(), ASYNCH	IRONOUS :: buf	38
	SER, INTENT(IN) :: count	A deteture	39
	<pre>[MPI_Datatype), INTENT(IN) EER, OPTIONAL, INTENT(OUT)</pre>		40
			41 42
	Fortran binding		
	MPI_FILE_READ_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR) INTEGER FH, COUNT, DATATYPE, IERROR		
	<pre><type> BUF(*)</type></pre>		
51			
			47
			48

```
1
     MPI_FILE_READ_ORDERED_END(fh, buf, status)
2
       INOUT
                fh
                                            file handle (handle)
3
       OUT
                buf
                                            initial address of buffer (choice)
4
5
       OUT
                                            status object (Status)
                status
6
7
     C binding
8
     int MPI_File_read_ordered_end(MPI_File fh, void *buf, MPI_Status *status)
9
     Fortran 2008 binding
10
     MPI_File_read_ordered_end(fh, buf, status, ierror)
11
         TYPE(MPI_File), INTENT(IN) :: fh
12
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
13
         TYPE(MPI_Status) :: status
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     Fortran binding
17
     MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR)
18
         INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
19
         <type> BUF(*)
20
21
22
     MPI_FILE_WRITE_ORDERED_BEGIN(fh, buf, count, datatype)
23
       INOUT
                                            file handle (handle)
^{24}
                fh
25
       IN
                buf
                                            initial address of buffer (choice)
26
       IN
                                            number of elements in buffer (integer)
                count
27
       IN
                                            datatype of each buffer element (handle)
                datatype
28
29
30
     C binding
^{31}
     int MPI_File_write_ordered_begin(MPI_File fh, const void *buf, int count,
32
                    MPI_Datatype datatype)
33
     Fortran 2008 binding
34
     MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror)
35
         TYPE(MPI_File), INTENT(IN) :: fh
36
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
37
         INTEGER, INTENT(IN) :: count
38
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     Fortran binding
42
     MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
43
         INTEGER FH, COUNT, DATATYPE, IERROR
44
         <type> BUF(*)
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(*)
```

13.5 File Interoperability

At the most basic level, file interoperability is the ability to read the information previously written to a file — not just the bits of data, but the actual information the bits represent. MPI guarantees full interoperability within a single MPI environment, and supports increased interoperability outside that environment through the external data representation (Section 13.5.2) as well as the data conversion functions (Section 13.5.3).

Interoperability within a single MPI environment (which could be considered "operability") ensures that file data written by one MPI process can be read by any other MPI process, subject to the consistency constraints (see Section 13.6.1), provided that it would have been possible to start the two processes simultaneously and have them reside in a single MPI_COMM_WORLD. Furthermore, both processes must see the same data values at every absolute byte offset in the file for which data was written.

This single environment file interoperability implies that file data is accessible regardless of the number of processes.

There are three aspects to file interoperability:

- transferring the bits,
- converting between different file structures, and
- converting between different machine representations.

The first two aspects of file interoperability are beyond the scope of this standard, as both are highly machine dependent. However, transferring the bits of a file into and out of the MPI environment (e.g., by writing a file to tape) is required to be supported by all MPI implementations. In particular, an implementation must specify how familiar

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 14they run. MPI supports multiple data representations: "native," "internal," and "external32." 1516An implementation may support additional data representations. MPI also supports user-

defined data representations (see Section 13.5.3). The "native" and "internal" data representations are implementation dependent, while the "external32" representation is common
 to all MPI implementations and facilitates file interoperability. The data representation is specified in the datarep argument to MPI_FILE_SET_VIEW.

- Advice to users. MPI is not guaranteed to retain knowledge of what data representation was used when a file is written. Therefore, to correctly retrieve file data, an MPI application is responsible for specifying the same data representation as was used to create the file. (*End of advice to users.*)
- "native" Data in this representation is stored in a file exactly as it is in memory. The advantage of this data representation is that data precision and I/O performance are not lost in type conversions with a purely homogeneous environment. The disadvantage is the loss of transparent interoperability within a heterogeneous MPI environment.

Advice to users. This data representation should only be used in a homogeneous MPI environment, or when the MPI application is capable of performing the data type conversions itself. (*End of advice to users.*)

Advice to implementors. When implementing read and write operations on top of MPI message-passing, the message data should be typed as MPI_BYTE to ensure that the message routines do not perform any type conversions on the data. (*End of advice to implementors.*)

"internal" This data representation can be used for I/O operations in a homogeneous or heterogeneous environment; the implementation will perform type conversions if necessary. The implementation is free to store data in any format of its choice, with the restriction that it will maintain constant extents for all predefined datatypes in any one file. The environment in which the resulting file can be reused is 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.*)

Unofficial Draft for Comment Only

21

22

23

 24

25 26

27

28

29

30

31

32

33 34

35

36

37

38 39

40

41

42

43

44

 $\mathbf{2}$

 31

Advice to implementors. Since "external32" is a superset of the functionality provided by "internal," an implementation may choose to implement "internal" as "external32." (*End of advice to implementors.*)

"external32" This data representation states that read and write operations convert all data from and to the "external32" representation defined in Section 13.5.2. The data conversion rules for communication also apply to these conversions (see Section 3.3.2). The data on the storage medium is always in this canonical representation, and the data in memory is always in the local process's native representation.

This data representation has several advantages. First, all processes reading the file in a heterogeneous MPI environment will automatically have the data converted to their respective native representations. Second, the file can be exported from one MPI environment and imported into any other MPI environment with the guarantee that the second environment will be able to read all the data in the file.

The disadvantage of this data representation is that data precision and I/O performance may be lost in data type conversions.

Advice to implementors. When implementing read and write operations on top of MPI message-passing, the message data should be converted to and from the "external32" representation in the client, and sent as type MPI_BYTE. This will avoid possible double data type conversions and the associated further loss of precision and performance. (*End of advice to implementors.*)

13.5.1 Datatypes for File Interoperability

If the file data representation is other than "native," care must be taken in constructing etypes and filetypes. Any of the datatype constructor functions may be used; however, for those functions that accept displacements in bytes, the displacements must be specified in terms of their values in the file for the file data representation being used. MPI will interpret these byte displacements as is; no scaling will be done. The function MPI_FILE_GET_TYPE_EXTENT can be used to calculate the extents of datatypes in the file. For etypes and filetypes that are portable datatypes (see Section 2.4), MPI will scale any displacements in the datatypes to match the file data representation. Datatypes passed as arguments to read/write routines specify the data layout in memory; therefore, they must always be constructed using displacements corresponding to displacements in memory.

Advice to users. One can logically think of the file as if it were stored in the memory of a file server. The etype and filetype are interpreted as if they were defined at this file server, by the same sequence of calls used to define them at the calling process. If the data representation is "native", then this logical file server runs on the same architecture as the calling process, so that these types define the same data layout on the file as they would define in the memory of the calling process. If the etype and filetype are portable datatypes, then the data layout defined in the file is the same as would be defined in the calling process memory, up to a scaling factor. The routine MPI_FILE_GET_TYPE_EXTENT can be used to calculate this scaling factor. Thus, two equivalent, portable datatypes will define the same data layout in the file, even in a heterogeneous environment with "internal", "external32", or user defined data representations. Otherwise, the etype and filetype must be constructed so that

Unofficial Draft for Comment Only

1 their typemap and extent are the same on any architecture. This can be achieved 2 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.) 202122MPI_FILE_GET_TYPE_EXTENT(fh, datatype, extent) 23 24 IN fh file handle (handle) 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 MPI_Aint *extent) 31 32 Fortran 2008 binding 33 MPI_File_get_type_extent(fh, datatype, extent, ierror) 34 TYPE(MPI_File), INTENT(IN) :: fh 35 TYPE(MPI_Datatype), INTENT(IN) :: datatype 36 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: extent 37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 38 39 Fortran binding MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERROR) 40INTEGER FH, DATATYPE, IERROR 41 INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT 4243 Returns the extent of datatype in the file fh. This extent will be the same for all 44processes accessing the file fh. If the current view uses a user-defined data representation 45(see Section 13.5.3), MPI uses the dtype_file_extent_fn callback to calculate the extent. 46 47Advice to implementors. In the case of user-defined data representations, the extent 48 of a derived datatype can be calculated by first determining the extents of the predefined datatypes in this derived datatype using dtype_file_extent_fn (see Section 13.5.3). (*End of advice to implementors.*)

13.5.2 External Data Representation: "external32"

All MPI implementations are required to support the data representation defined in this section. Support of optional datatypes (e.g., MPI_INTEGER2) is not required.

All floating point values are in big-endian IEEE format [39] 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 [40]. Wide characters (of type MPI_WCHAR) are in Unicode format [63].

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 [39], 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 [61]. (End of advice to implementors.)

All data is byte aligned, regardless of type. All data items are stored contiguously in the file (if the file view is contiguous).

Advice to implementors. All bytes of LOGICAL and bool must be checked to determine the value. (End of advice to implementors.)

Advice to users. The type MPI_PACKED is treated as bytes and is not converted. The user should be aware that MPI_PACK has the option of placing a header in the beginning of the pack buffer. (*End of advice to users.*)

The sizes of the predefined datatypes returned from MPI_TYPE_CREATE_F90_REAL, MPI_TYPE_CREATE_F90_COMPLEX, and MPI_TYPE_CREATE_F90_INTEGER are defined in Section 18.1.9, page 729.

Advice to implementors. When converting a larger size integer to a smaller size integer, only the least significant bytes are moved. Care must be taken to preserve the sign bit value. This allows no conversion errors if the data range is within the range of the smaller size integer. (End of advice to implementors.)

Table 13.2, 13.3, and 13.4 specify the sizes of predefined, optional, and C++ datatypes in "external32" format, respectively.

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

 $41 \\ 42$

43

44

45

 $46 \\ 47$

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	$\frac{1}{2}$	
8	MPI_SHORT	$\frac{2}{2}$	
9	MPI_SHOKT MPI_UNSIGNED_SHORT	$\frac{2}{2}$	
10	MPI_UNSIGNED_SHORT	$\frac{2}{4}$	
11	MPI_LONG	4	
12	MPI_LONG MPI_UNSIGNED	4	
13	MPI_UNSIGNED_LONG	4 4	
14			
14	MPI_LONG_LONG_INT	8 8	
16	MPI_UNSIGNED_LONG_LONG MPI_FLOAT	8 4	
17	MPI_FLOAT MPI_DOUBLE		
18		8	
19	MPI_LONG_DOUBLE	16	
20	MPI_C_BOOL	1	
20	MPI_INT8_T	$\frac{1}{2}$	
22	MPI_INT16_T		
23	MPI_INT32_T	4	
24	MPI_INT64_T	8	
25	MPI_UINT8_T	$\frac{1}{2}$	
26	MPI_UINT16_T	2	
27	MPI_UINT32_T	4	
28	MPI_UINT64_T	8	
29	MPI_AINT	8	
30	MPI_COUNT	8	
31	MPI_OFFSET	$\frac{8}{2*4}$	
32	MPI_C_COMPLEX	2^{*4} 2^{*4}	
33	MPI_C_FLOAT_COMPLEX	$\frac{2^{+}4}{2^{*}8}$	
34	MPI_C_DOUBLE_COMPLEX		
35	MPI_C_LONG_DOUBLE_COMPLEX	2*16	
36	MPI_CHARACTER	1	
37	MPI_LOGICAL	4	
38	MPI_INTEGER	4	
39	MPI_REAL	4	
	MPI_DOUBLE_PRECISION	8	
40		2*4	
41	MPI_DOUBLE_COMPLEX	2*8	
42	MPI_CXX_BOOL	1	
43	MPI_CXX_FLOAT_COMPLEX	2*4	
44	MPI_CXX_DOUBLE_COMPLEX	2*8	
45	MPI_CXX_LONG_DOUBLE_COMPLEX	2*16	
46			

Table 13.2: "external32" sizes of predefined datatypes

Unofficial Draft for Comment Only

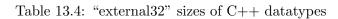
47

 48

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 13.3: "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



Unofficial Draft for Comment Only

1	13.5.3 User-Defined Data Representations				
2 3	There are two situations that cannot be handled by the required representations:				
4	1. a user wants to write a file in a representation unknown to the implementation, and				
5 6	2. a us	2. a user wants to read a file written in a representation unknown to the implementation.			
7 8 9	User-defined data representations allow the user to insert a third party converter into the I/O stream to do the data representation conversion.				
10 11 12	MPI_REC	GISTER_DATAREP(datarep, r dtype_file_extent_fn, e	ead_conversion_fn, write_conversion_fn, extra_state)		
13	IN	datarep	data representation identifier (string)		
14 15	IN	read_conversion_fn	function invoked to convert from file representation to native representation (function)		
16 17 18	IN	write_conversion_fn	function invoked to convert from native representation to file representation (function)		
19 20	IN	dtype_file_extent_fn	function invoked to get the extent of a datatype as represented in the file (function)		
21 22	IN	extra_state	extra state		
23	C bindi	ng			
24		_Register_datarep(const)	char *datarep,		
25			sion_function *read_conversion_fn,		
26 27		MPI_Datarep_conver	sion_function *write_conversion_fn,		
21		MPI_Datarep_extent	_function *dtype_file_extent_fn,		
29		void *extra_state)			
30	Fortran	2008 binding			
31		2	ead_conversion_fn, write_conversion_fn,		
32			fn, extra_state, ierror)		
33	CHAF	RACTER(LEN=*), INTENT(IN)			
34			rsion_function) :: read_conversion_fn,		
35	write_conversion_fn				
36	<pre>PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn</pre>				
37	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state				
38	INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
39	Fortran binding				
40	MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN,				
41	DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR)				
42	CHARACTER*(*) DATAREP				
43	EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN				
44	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE				
45 46	INTEGER IERROR				
40	ጥኬ -	coll accordance wood according	ion for write conversion for and drive file extent for		
48			ion_fn, write_conversion_fn, and dtype_file_extent_fn er datarep. datarep can then be used as an argument		

to MPI_FILE_SET_VIEW, causing subsequent data access operations to call the conversion functions to convert all data items accessed between file data representation and native representation. MPI_REGISTER_DATAREP is a local operation and only registers the data representation for the calling MPI process. If datarep is already defined, an error in the error class MPI_ERR_DUP_DATAREP is raised using the default file error handler (see Section 13.7). The length of a data representation string is limited to the value of MPI_MAX_DATAREP_STRING. MPI_MAX_DATAREP_STRING must have a value of at least 64. No routines are provided to delete data representations and free the associated resources; it is not expected that an application will generate them in significant numbers.

```
Extent Callback
                                                                                      12
typedef int MPI_Datarep_extent_function(MPI_Datatype datatype,
                                                                                      13
              MPI_Aint *extent, void *extra_state);
                                                                                      14
                                                                                      15
ABSTRACT INTERFACE
                                                                                      16
  SUBROUTINE MPI_Datarep_extent_function(datatype, extent, extra_state,
                                                                                      17
               ierror)
                                                                                      18
    TYPE(MPI_Datatype) :: datatype
                                                                                      19
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extent, extra_state
                                                                                      20
    INTEGER :: ierror
                                                                                      21
SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)
                                                                                      22
    INTEGER DATATYPE, IERROR
                                                                                      23
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE
                                                                                      ^{24}
                                                                                      25
    The function dtype_file_extent_fn must return, in file_extent, the number of bytes re-
                                                                                      26
quired to store datatype in the file representation. The function is passed, in extra_state,
                                                                                      27
the argument that was passed to the MPI_REGISTER_DATAREP call. MPI will only call
                                                                                      28
this routine with predefined datatypes employed by the user.
                                                                                      29
                                                                                      30
Datarep Conversion Functions
                                                                                      31
typedef int MPI_Datarep_conversion_function(void *userbuf,
                                                                                      32
              MPI_Datatype datatype, int count, void *filebuf,
                                                                                      33
              MPI_Offset position, void *extra_state);
                                                                                      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
SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF,
                                                                                      45
              POSITION, EXTRA_STATE, IERROR)
                                                                                      46
    <TYPE> USERBUF(*), FILEBUF(*)
                                                                                      47
    INTEGER DATATYPE, COUNT, IERROR
                                                                                      48
```

1

 $\mathbf{2}$

3

4

5 6

 $\overline{7}$

8

1	INTEGER(KIND=MPI_OFF
2	INTEGER(KIND=MPI_ADD
3	
4	The function read_conve
5	tive representation. Before of
6	contiguous data items. The t
7	predefined datatype in the typ
8	the argument that was passe
9	copy all count data items from
10	converting each data item from
11	equivalent to the datatype th
12	is less than the size of the co
13	as being contiguously tiled o
14	converted data at the location
15	Advice to users. Althe
16	and MPI_UNPACK, one
17	and position. In the co
18	of typemap entries of d
19	MPI_PACK, incount refe
20	of bytes. (End of advice
21	of bytes. (End of dable
22	Advice to implementors.
23	
24	1. Get file extent of a
25	
26	2. Allocate a filebuf l
27	3. Read data from fil
28	4. Call read_conversion
29	5. Deallocate filebuf
30	5. Deanocate mebui
31	(End of advice to imple
32	
33	If MPI cannot allocate a
34	a read operation, it may call
35	and userbuf and reading such

SET_KIND) POSITION RESS_KIND) EXTRA_STATE

ersion_fn must convert from file data representation to nacalling this routine, MPI allocates and fills filebuf with count ype of each data item matches the corresponding entry for the pe signature of datatype. The function is passed, in extra state, ed to the MPI_REGISTER_DATAREP call. The function must m filebuf to userbuf in the distribution described by datatype, m file representation to native representation. datatype will be at the user passed to the read function. If the size of datatype ount data items, the conversion function must treat datatype wer the userbuf. The conversion function must begin storing n in userbuf specified by position into the (tiled) datatype.

ough the conversion functions have similarities to MPL PACK should note the differences in the use of the arguments count nversion functions, count is a count of data items (i.e., count latatype), and position is an index into this typemap. In ers to the number of whole datatypes, and position is a number e to users.)

. A converted read operation could be implemented as follows:

- all data items
- large enough to hold all count data items
- le into filebuf
 - on_fn to convert data and place it into userbuf
 - ementors.)

buffer large enough to hold all the data to be converted from l the conversion function repeatedly using the same datatype 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.

40 41

36

37

38

39

42Rationale. Passing the conversion function a position and one datatype for the 43 transfer allows the conversion function to decode the datatype only once and cache an 44 internal representation of it on the datatype. Then on subsequent calls, the conversion 45 function can use the **position** to quickly find its place in the datatype and continue 46 storing converted data where it left off at the end of the previous call. (End of 47 rationale.) 48

Advice to users. Although the conversion function may usefully cache an internal representation on the datatype, it should not cache any state information specific to an ongoing conversion operation, since it is possible for the same datatype to be used concurrently in multiple conversion operations. (*End of advice to users.*)

The function write_conversion_fn must convert from native representation to file data representation. Before calling this routine, MPI allocates filebuf of a size large enough to hold count contiguous data items. The type of each data item matches the corresponding entry for the predefined datatype in the type signature of datatype. The function must copy count data items from userbuf in the distribution described by datatype, to a contiguous distribution in filebuf, converting each data item from native representation to file representation. If the size of datatype is less than the size of count data items, the conversion function must treat datatype as being contiguously tiled over the userbuf.

The function must begin copying at the location in userbuf specified by position into the (tiled) datatype. datatype will be equivalent to the datatype that the user passed to the write function. The function is passed, in extra_state, the argument that was passed to the MPI_REGISTER_DATAREP call.

The predefined constant MPI_CONVERSION_FN_NULL may be used as either write_conversion_fn or read_conversion_fn. In that case, MPI will not attempt to invoke write_conversion_fn or read_conversion_fn, respectively, but will perform the requested data access using the native data representation.

An MPI implementation must ensure that all data accessed is converted, either by using a filebuf large enough to hold all the requested data items or else by making repeated calls to the conversion function with the same datatype argument and appropriate values for position.

An implementation will only invoke the callback routines in this section (read_conversion_fn, write_conversion_fn, and dtype_file_extent_fn) when one of the read or write routines in Section 13.4, or MPI_FILE_GET_TYPE_EXTENT is called by the user. dtype_file_extent_fn will only be passed predefined datatypes employed by the user. The conversion functions will only be passed datatypes equivalent to those that the user has passed to one of the routines noted above.

The conversion functions must be reentrant. User defined data representations are restricted to use byte alignment for all types. Furthermore, it is erroneous for the conversion functions to call any collective routines or to free datatype.

The conversion functions should return an error code. If the returned error code has a value other than MPI_SUCCESS, the implementation will raise an error in the class MPI_ERR_CONVERSION.

13.5.4 Matching Data Representations

It is the user's responsibility to ensure that the data representation used to read data from a file is *compatible* with the data representation that was used to write that data to the file.

In general, using the same data representation name when writing and reading a file does not guarantee that the representation is compatible. Similarly, using different representation names on two different implementations may yield compatible representations.

Compatibility can be obtained when "external32" representation is used, although precision may be lost and the performance may be less than when "native" representation is

 24

- The data access routines directly use types enumerated in Section 13.5.2, that are supported by all implementations participating in the I/O. The predefined type used to write a data item must also be used to read a data item.
- In the case of Fortran 90 programs, the programs participating in the data accesses obtain compatible datatypes using MPI routines that specify precision and/or range (Section 18.1.9).
- 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 18.1.9 defines routines that support the use of matching datatypes in heterogeneous environments and contains examples illustrating their use. (End of advice to users.)

13.6 Consistency and Semantics

²³ 13.6.1 File Consistency

Consistency semantics define the outcome of multiple accesses to a single file. All file 2526 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 27file handle, sequential consistency among all accesses using file handles created from a single 28collective open with atomic mode enabled, and user-imposed consistency among accesses 29other than the above. Sequential consistency means the behavior of a set of operations will 30 be as if the operations were performed in some serial order consistent with program order; 31 32 each access appears atomic, although the exact ordering of accesses is unspecified. Userimposed consistency may be obtained using program order and calls to MPI_FILE_SYNC. 33

34Let 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 35 FOO. Note that nothing restrictive is said about FH_1 and FH_2 : the sizes of FH_1 and 36 FH_2 may be different, the groups of processes used for each open may or may not intersect, 37 the file handles in FH_1 may be destroyed before those in FH_2 are created, etc. Consider 3839 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 40different collective opens (e.g., $fh_1 \in FH_1$ and $fh_2 \in FH_2$). 41

For the purpose of consistency semantics, a matched pair (Section 13.4.5) of split collective data access operations (e.g., MPI_FILE_READ_ALL_BEGIN and

MPI_FILE_READ_ALL_END) compose a single data access operation. Similarly, a non blocking data access routine (e.g., MPI_FILE_IREAD) and the routine which completes the
 request (e.g., MPI_WAIT) also compose a single data access operation. For all cases below,
 these data access operations are subject to the same constraints as blocking data access
 operations.

1

2

3 4

5

6 7

8

9

10 11

12

13 14

15

16 17

18

19

20 21

Advice to users. For an MPI_FILE_IREAD and MPI_WAIT pair, the operation begins when MPI_FILE_IREAD is called and ends when MPI_WAIT returns. (*End of advice to users.*)

Assume that A_1 and A_2 are two data access operations. Let D_1 (D_2) be the set of absolute byte displacements of every byte accessed in A_1 (A_2) . The two data accesses *overlap* if $D_1 \cap D_2 \neq \emptyset$. The two data accesses *conflict* if they overlap and at least one is a write access.

Let SEQ_{fh} be a sequence of file operations on a single file handle, bracketed by MPI_FILE_SYNCs on that file handle. (Both opening and closing a file implicitly perform an MPI_FILE_SYNC.) SEQ_{fh} is a "write sequence" if any of the data access operations in the sequence are writes or if any of the file manipulation operations in the sequence change the state of the file (e.g., MPI_FILE_SET_SIZE or MPI_FILE_PREALLOCATE). Given two sequences, SEQ_1 and SEQ_2 , we say they are not *concurrent* if one sequence is guaranteed to completely precede the other (temporally).

The requirements for guaranteeing sequential consistency among all accesses to a particular file are divided into the three cases given below. If any of these requirements are not met, then the value of all data in that file is implementation dependent.

Case 1: $fh_1 \in FH_1$ All operations on fh_1 are sequentially consistent if atomic mode is set. If nonatomic mode is set, then all operations on fh_1 are sequentially consistent if they are either nonconcurrent, nonconflicting, or both.

Case 2: $fh_{1a} \in FH_1$ and $fh_{1b} \in FH_1$ Assume A_1 is a data access operation using fh_{1a} , and A_2 is a data access operation using fh_{1b} . If for any access A_1 , there is no access A_2 that conflicts with A_1 , then MPI guarantees sequential consistency.

However, unlike POSIX semantics, the default MPI semantics for conflicting accesses do not guarantee sequential consistency. If A_1 and A_2 conflict, sequential consistency can be guaranteed by either enabling atomic mode via the MPI_FILE_SET_ATOMICITY routine, or meeting the condition described in Case 3 below.

Case 3: $fh_1 \in FH_1$ and $fh_2 \in FH_2$ Consider access to a single file using file handles from distinct collective opens. In order to guarantee sequential consistency, MPI_FILE_SYNC must be used (both opening and closing a file implicitly perform an MPI_FILE_SYNC).

Sequential consistency is guaranteed among accesses to a single file if for any write sequence SEQ_1 to the file, there is no sequence SEQ_2 to the file which is *concurrent* with SEQ_1 . To guarantee sequential consistency when there are write sequences, MPI_FILE_SYNC must be used together with a mechanism that guarantees nonconcurrency

of the sequences.

See the examples in Section 13.6.11 for further clarification of some of these consistency semantics.

 $\mathbf{5}$

```
1
     MPI_FILE_SET_ATOMICITY(fh, flag)
2
       INOUT
                 fh
                                              file handle (handle)
3
                                              true to set atomic mode, false to set nonatomic mode
       IN
                 flag
4
                                              (logical)
5
6
7
     C binding
8
     int MPI_File_set_atomicity(MPI_File fh, int flag)
9
     Fortran 2008 binding
10
     MPI_File_set_atomicity(fh, flag, ierror)
11
          TYPE(MPI_File), INTENT(IN) :: fh
12
          LOGICAL, INTENT(IN) :: flag
13
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
     Fortran binding
16
     MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR)
17
          INTEGER FH, IERROR
18
          LOGICAL FLAG
19
          Let FH be the set of file handles created by one collective open. The consistency
20
     semantics for data access operations using FH is set by collectively calling
21
     MPI_FILE_SET_ATOMICITY on FH. MPI_FILE_SET_ATOMICITY is collective; all pro-
22
     cesses in the group must pass identical values for fh and flag. If flag is true, atomic mode is
23
     set; if flag is false, nonatomic mode is set.
24
          Changing the consistency semantics for an open file only affects new data accesses.
25
     All completed data accesses are guaranteed to abide by the consistency semantics in effect
26
     during their execution. Nonblocking data accesses and split collective operations that have
27
     not completed (e.g., via MPI_WAIT) are only guaranteed to abide by nonatomic mode
28
     consistency semantics.
29
30
           Advice to implementors. Since the semantics guaranteed by atomic mode are stronger
31
           than those guaranteed by nonatomic mode, an implementation is free to adhere to
32
           the more stringent atomic mode semantics for outstanding requests. (End of advice
33
           to implementors.)
34
35
36
37
     MPI_FILE_GET_ATOMICITY(fh, flag)
38
       IN
                 fh
                                              file handle (handle)
39
       OUT
                 flag
                                              true if atomic mode, false if nonatomic mode (logical)
40
41
     C binding
42
     int MPI_File_get_atomicity(MPI_File fh, int *flag)
43
44
     Fortran 2008 binding
45
     MPI_File_get_atomicity(fh, flag, ierror)
46
          TYPE(MPI_File), INTENT(IN) :: fh
47
          LOGICAL, INTENT(OUT) :: flag
48
```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR) INTEGER FH, IERROR LOGICAL FLAG

MPI_FILE_GET_ATOMICITY returns the current consistency semantics for data access operations on the set of file handles created by one collective open. If flag is true, atomic mode is enabled; if flag is false, nonatomic mode is enabled.

MPI_FILE_SYNC(fh)

INOUT fh

file handle (handle)

C binding int MPI_File_sync(MPI_File fh)

Fortran 2008 binding

MPI_File_sync(fh, ierror)
 TYPE(MPI_File), INTENT(IN) :: fh
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

MPI_FILE_SYNC(FH, IERROR) INTEGER FH, IERROR

Calling MPI_FILE_SYNC with fh causes all previous writes to fh by the calling process to be transferred to the storage device. If other processes have made updates to the storage device, then all such updates become visible to subsequent reads of fh by the calling process. MPI_FILE_SYNC may be necessary to ensure sequential consistency in certain cases (see above).

MPI_FILE_SYNC is a collective operation.

The user is responsible for ensuring that all nonblocking requests and split collective operations on fh have been completed before calling MPI_FILE_SYNC — otherwise, the call to MPI_FILE_SYNC is erroneous.

13.6.2 Random Access vs. Sequential Files

MPI distinguishes ordinary random access files from sequential stream files, such as pipes and tape files. Sequential stream files must be opened with the MPI_MODE_SEQUENTIAL flag set in the amode. For these files, the only permitted data access operations are shared file pointer reads and writes. Filetypes and etypes with holes are erroneous. In addition, the notion of file pointer is not meaningful; therefore, calls to MPI_FILE_SEEK_SHARED and MPI_FILE_GET_POSITION_SHARED are erroneous, and the pointer update rules specified for the data access routines do not apply. The amount of data accessed by a data access operation will be the amount requested unless the end of file is reached or an error is raised.

Rationale. This implies that reading on a pipe will always wait until the requested 46 amount of data is available or until the process writing to the pipe has issued an end of file. (End of rationale.) 47

Unofficial Draft for Comment Only

2

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

Finally, for some sequential files, such as those corresponding to magnetic tapes or $\mathbf{2}$ streaming network connections, writes to the file may be destructive. In other words, a 3 write may act as a truncate (a MPI_FILE_SET_SIZE with size set to the current position) 4 followed by the write.

13.6.3 Progress

The progress rules of MPI are both a promise to users and a set of constraints on imple-8 mentors. In cases where the progress rules restrict possible implementation choices more 9 than the interface specification alone, the progress rules take precedence. 10

All blocking routines must complete in finite time unless an exceptional condition (such 11 as resource exhaustion) causes an error. 12

Nonblocking data access routines inherit the following progress rule from nonblocking 13 point to point communication: a nonblocking write is equivalent to a nonblocking send for 14which a receive is eventually posted, and a nonblocking read is equivalent to a nonblocking 15receive for which a send is eventually posted. 16

Finally, an implementation is free to delay progress of collective routines until all pro-17cesses in the group associated with the collective call have invoked the routine. Once all 18 processes in the group have invoked the routine, the progress rule of the equivalent noncol-19 lective routine must be followed. 20

13.6.4 **Collective File Operations**

23Collective file operations are subject to the same restrictions as collective communication 24 operations. For a complete discussion, please refer to the semantics set forth in Section 5.14. 25

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.

29 30 31

21

22

26

27

28

Nonblocking Collective File Operations 13.6.5

32 Nonblocking collective file operations are defined only for data access routines with explicit 33 offsets and individual file pointers but not with shared file pointers.

34Nonblocking collective file operations are subject to the same restrictions as blocking 35 collective I/O operations. All processes belonging to the group of the communicator that 36 was used to open the file must call collective I/O operations (blocking and nonblocking) 37 in the same order. This is consistent with the ordering rules for collective operations in 38 threaded environments. For a complete discussion, please refer to the semantics set forth 39 in Section 5.14.

40 Nonblocking collective I/O operations do not match with blocking collective I/O oper- 41 ations. Multiple nonblocking collective I/O operations can be outstanding on a single file 42handle. High quality MPI implementations should be able to support a large number of 43pending nonblocking I/O operations.

44All nonblocking collective I/O calls are local and return immediately, irrespective of the 45status of other processes. The call initiates the operation which may progress independently 46of any communication, computation, or I/O. The call returns a request handle, which must 47be passed to a completion call. Input buffers should not be modified and output buffers 48should not be accessed before the completion call returns. The same progress rules described

Unofficial Draft for Comment Only

1

56

for nonblocking collective operations apply for nonblocking collective I/O operations. For a complete discussion, please refer to the semantics set forth in Section 5.12.

13.6.6 Type Matching

The type matching rules for I/O mimic the type matching rules for communication with one exception: if etype is MPI_BYTE, then this matches any datatype in a data access operation. In general, the etype of data items written must match the etype used to read the items, and for each data access operation, the current etype must also match the type declaration of the data access buffer.

Advice to users. In most cases, use of MPI_BYTE as a wild card will defeat the file interoperability features of MPI. File interoperability can only perform automatic conversion between heterogeneous data representations when the exact datatypes accessed are explicitly specified. (*End of advice to users.*)

13.6.7 Miscellaneous Clarifications

Once an I/O routine completes, it is safe to free any opaque objects passed as arguments to that routine. For example, the comm and info used in an MPI_FILE_OPEN, or the etype and filetype used in an MPI_FILE_SET_VIEW, can be freed without affecting access to the file. Note that for nonblocking routines and split collective operations, the operation must be completed before it is safe to reuse data buffers passed as arguments.

As in communication, datatypes must be committed before they can be used in file manipulation or data access operations. For example, the etype and filetype must be committed before calling MPI_FILE_SET_VIEW, and the datatype must be committed before calling MPI_FILE_READ or MPI_FILE_WRITE.

13.6.8 MPI_Offset Type

MPI_Offset is an integer type of size sufficient to represent the size (in bytes) of the largest file supported by MPI. Displacements and offsets are always specified as values of type MPI_Offset.

In Fortran, the corresponding integer is an integer with kind parameter MPI_OFFSET_KIND, which is defined in the mpi_f08 module, the mpi module and the mpif.h include file.

In Fortran 77 environments that do not support KIND parameters, MPI_Offset arguments should be declared as an INTEGER of suitable size. The language interoperability implications for MPI_Offset are similar to those for addresses (see Section 18.2).

13.6.9 Logical vs. Physical File Layout

MPI specifies how the data should be laid out in a virtual file structure (the view), not how that file structure is to be stored on one or more disks. Specification of the physical file structure was avoided because it is expected that the mapping of files to disks will be system specific, and any specific control over file layout would therefore restrict program portability. However, there are still cases where some information may be necessary to optimize file layout. This information can be provided as *hints* specified via info when a file is created (see Section 13.2.8).

Unofficial Draft for Comment Only

 $\overline{7}$

 24

 31

1	13.6.10 File Size
2 3 4 5 6 7	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 <i>size changing</i> 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.
8 9 10	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 <i>high byte</i> be the byte in that set with the largest displacement. The file size is the larger of
11 12	• One plus the displacement of the high byte.
13	\bullet The size immediately after the size changing routine, or $MPI_FILE_OPEN,$ returned.
14 15 16 17 18 19	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).
20 21 22 23 24	Advice to users. Any sequence of operations containing the collective routines MPI_FILE_SET_SIZE and MPI_FILE_PREALLOCATE is a write sequence. As such, sequential consistency in nonatomic mode is not guaranteed unless the conditions in Section 13.6.1 are satisfied. (End of advice to users.)
25 26 27	File pointer update semantics (i.e., file pointers are updated by the amount accessed) are only guaranteed if file size changes are sequentially consistent.
28 29 30 31 32 33 34	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. (<i>End of advice to users.</i>)
35 36	13.6.11 Examples
37 38	The examples in this section illustrate the application of the MPI consistency and semantics guarantees. These address
39 40	\bullet conflicting accesses on file handles obtained from a single collective open, and
41 42	• all accesses on file handles obtained from two separate collective opens.
43 44 45 46 47	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.

```
1
/* Process 0 */
                                                                                        \mathbf{2}
int i, a[10];
                                                                                        3
int TRUE = 1;
                                                                                        4
for (i=0;i<10;i++)
                                                                                        5
                                                                                        6
   a[i] = 5;
MPI_File_open(MPI_COMM_WORLD, "workfile",
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0);
                                                                                        9
                                                                                        10
MPI_File_set_view(fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
                                                                                        11
MPI_File_set_atomicity(fh0, TRUE);
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status);
                                                                                        12
                                                                                        13
/* MPI_Barrier(MPI_COMM_WORLD); */
                                                                                        14
/* Process 1 */
                                                                                        15
int b[10];
                                                                                        16
int TRUE = 1;
                                                                                        17
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 MPLBARRIER.
                                                                                       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	<pre>MPI_MODE_RDWR MPI_MODE_CREATE, MPI_INFO_NULL, &fh1);</pre>		
2			
	<pre>MPI_File_set_view(fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);</pre>		
3	<pre>MPI_File_sync(fh1);</pre>		
4	<pre>MPI_Barrier(MPI_COMM_WORLD);</pre>		
5	<pre>MPI_File_sync(fh1);</pre>		
6	<pre>MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status);</pre>		
7 8	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			
11 12	• The first sync guarantees that the data written by all processes is transferred to the storage device.		
13 14 15	• The second sync guarantees that all data which has been transferred to the storage device is visible to all processes. (This does not affect process 0 in this example.)		
15 16 17	The following program represents an erroneous attempt to achieve consistency by elim- inating the apparently superfluous second "sync" call for each process.		
18			
19	/* THIS EXAMPLE IS ERRONEOUS */		
20	/* Process 0 */		
21	int i, a[10];		
21	for (i=0;i<10;i++)		
22	a[i] = 5;		
23 24			
25	<pre>MPI_File_open(MPI_COMM_WORLD, "workfile",</pre>		
26	<pre>MPI_MODE_RDWR MPI_MODE_CREATE, MPI_INFO_NULL, &fh0);</pre>		
	<pre>MPI_File_set_view(fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);</pre>		
27	<pre>MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status);</pre>		
28	<pre>MPI_File_sync(fh0);</pre>		
29	<pre>MPI_Barrier(MPI_COMM_WORLD);</pre>		
30	/* Process 1 */		
31			
32			
33	MPI_File_open(MPI_COMM_WORLD, "workfile",		
34	MPI_MODE_RDWR MPI_MODE_CREATE, MPI_INFO_NULL, &fh1);		
35	<pre>MPI_File_set_view(fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);</pre>		
36	MPI_Barrier(MPI_COMM_WORLD);		
37	<pre>MPI_File_sync(fh1);</pre>		
38	<pre>MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status);</pre>		
39	/* THIS EXAMPLE IS ERRONEOUS */		
40	/* THIS EXAMPLE IS ERRONEOUS */		
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.		
43			
44	Advice to users. Some implementations may choose to implement MPI_FILE_SYNC		
45	as a temporally synchronizing function. When using such an implementation, the		
46	"sync-barrier-sync" construct above can be replaced by a single "sync." The results of		
47	using such code with an implementation for which MPI_FILE_SYNC is not temporally		
48	synchronizing is undefined. (End of advice to users.)		

Unofficial Draft for Comment Only

Asynchronous I/O

The behavior of asynchronous I/O operations is determined by applying the rules specified above for synchronous I/O operations.

The following examples all access a preexisting file "myfile." Word 10 in myfile initially contains the integer 2. Each example writes and reads word 10.

First consider the following code fragment:

For asynchronous data access operations, MPI specifies that the access occurs at any time between the call to the asynchronous data access routine and the return from the corresponding request complete routine. Thus, executing either the read before the write, or the write before the read is consistent with program order. If atomic mode is set, then MPI guarantees sequential consistency, and the program will read either 2 or 4 into b. If atomic mode is not set, then sequential consistency is not guaranteed and the program may read something other than 2 or 4 due to the conflicting data access.

Similarly, the following code fragment does not order file accesses:

```
int a = 4, b;
                                                                                     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
int a = 4, b;
                                                                                     39
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(&regs[1], &status);
                                                                                     46
                                                                                     47
```

defines the same ordering as:

1 2

3

4

5

6

7

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24 25

```
1
      int a = 4, b;
\mathbf{2}
     MPI_File_open(MPI_COMM_WORLD, "myfile",
3
                      MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
4
     MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
\mathbf{5}
     MPI_File_write_at(fh, 10, &a, 1, MPI_INT, &status );
6
     MPI_File_read_at(fh, 10, &b, 1, MPI_INT, &status );
7
      Since
8
9
         • 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 13.6.5, nonblocking collective I/O operations have
20
      to be called in the same order on the file handle by all processes.
21
22
          Similar considerations apply to conflicting accesses of the form:
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
^{31}
     MPI_File_write_all_begin(fh,...);
32
     MPI_File_read_all_begin(fh,...);
33
     MPI_File_read_all_end(fh,...);
34
     MPI_File_write_all_end(fh,...);
35
     since split collective operations on the same file handle may not overlap (see Section 13.4.5).
36
37
```

13.7 I/O Error Handling

38

39

44

⁴⁰ By default, communication errors are fatal — MPI_ERRORS_ARE_FATAL is the default error
⁴¹ handler associated with MPI_COMM_WORLD. I/O errors are usually less catastrophic (e.g.,
⁴² "file not found") than communication errors, and common practice is to catch these errors
⁴³ 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

Like communicators, each file handle has an error handler associated with it. The MPI I/O error handling routines are defined in Section 8.3.

When MPI calls a user-defined error handler resulting from an error on a particular file handle, the first two arguments passed to the file error handler are the file handle and the error code. For I/O errors that are not associated with a valid file handle (e.g., in MPI_FILE_OPEN or MPI_FILE_DELETE), the first argument passed to the error handler is MPI_FILE_NULL.

I/O error handling differs from communication error handling in another important aspect. By default, the predefined error handler for file handles is MPI_ERRORS_RETURN. The **default file error** handler has two purposes: when a new file handle is created (by MPI_FILE_OPEN), the error handler for the new file handle is initially set to the default file error handler, and I/O routines that have no valid file handle on which to raise an error (e.g., MPI_FILE_OPEN or MPI_FILE_DELETE) use the default file error handler. The default file error handler can be changed by specifying MPI_FILE_NULL as the fh argument to MPI_FILE_SET_ERRHANDLER. The current value of the default file error handler can be determined by passing MPI_FILE_NULL as the fh argument to MPI_FILE_GET_ERRHANDLER.

Rationale. For communication, the default error handler is inherited from MPI_COMM_WORLD when using the World Model. In I/O, there is no analogous "root" file handle from which default properties can be inherited. Rather than invent a new global file handle, the default file error handler is manipulated as if it were attached to MPI_FILE_NULL. (*End of rationale.*)

```
13.8 I/O Error Classes
```

The implementation dependent error codes returned by the I/O routines can be converted into the error classes defined in Table 13.5.

In addition, calls to routines in this chapter may raise errors in other MPI classes, such as MPI_ERR_TYPE.

13.9 Examples

13.9.1 Double Buffering with Split Collective I/O

This example shows how to overlap computation and output. The computation is performed by the function compute_buffer().

/*=					39
*					40
*	Function:	double_buffer			41
*					42
*	Synopsis:				43
*	void	double_buffer(44
*		MPI_File fh,	**	IN	45
*		MPI_Datatype buftype,	**	IN	46
*		int bufcount	**	IN	47
*)				48

Unofficial Draft for Comment Only

1	MPI_ERR_FILE	Invalid file handle
2	MPI_ERR_NOT_SAME	Collective argument not identical on all
3		processes, or collective routines called in
4		a different order by different processes
5	MPI_ERR_AMODE	Error related to the amode passed to
6		MPI_FILE_OPEN
7 8	MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to MPI_FILE_SET_VIEW
9	MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on
10		a file which supports sequential access only
11	MPI_ERR_NO_SUCH_FILE	File does not exist
12	MPI_ERR_FILE_EXISTS	File exists
13	MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)
14	MPI_ERR_ACCESS	Permission denied
15	MPI_ERR_NO_SPACE	Not enough space
16	MPI_ERR_QUOTA	Quota exceeded
17	MPI_ERR_READ_ONLY	Read-only file or file system
18	MPI_ERR_FILE_IN_USE	File operation could not be completed, as
19		the file is currently open by some process
20	MPI_ERR_DUP_DATAREP	Conversion functions could not be regis-
21		tered because a data representation identi-
22		fier that was already defined was passed to
23		MPI_REGISTER_DATAREP
24	MPI_ERR_CONVERSION	An error occurred in a user supplied data
25		conversion function.
26	MPI_ERR_IO	Other I/O error
27	Table 13 5	: I/O Error Classes
28	Table 13.5.	. I/O EITOI Classes
29	· ·	
30		
31		
32		
33		
34		
35 36		
37		
38		
39		
40		
41		
42		
43		
44		
45		
46		
47		
48		

```
1
 *
                                                                             2
 * Description:
                                                                             3
       Performs the steps to overlap computation with a collective write
 *
       by using a double-buffering technique.
 * Parameters:
       fh
                          previously opened MPI file handle
                                                                             7
 *
                          MPI datatype for memory layout
       buftype
 *
                                                                             9
                         (Assumes a compatible view has been set on fh)
                                                                             10
 *
       bufcount
                        # buftype elements to transfer
 *-----*/
                                                                             11
                                                                             12
/* this macro switches which buffer "x" is pointing to */
                                                                             13
#define TOGGLE_PTR(x) (((x)==(buffer1)) ? (x=buffer2) : (x=buffer1))
                                                                             14
                                                                             15
void double_buffer(MPI_File fh, MPI_Datatype buftype, int bufcount)
                                                                             16
                                                                             17
{
                                                                             18
  MPI_Status status; /* status for MPI calls */
                                                                             19
  float *buffer1, *buffer2; /* buffers to hold results */
                                                                             20
                                                                             21
  float *compute_buf_ptr; /* destination buffer */
                            /* for computing */
                                                                             22
  float *write_buf_ptr; /* source for writing */
                                                                             23
                                                                             24
                            /* determines when to quit */
  int done;
                                                                             25
                                                                             26
  /* buffer initialization */
  buffer1 = (float *)
                                                                             27
                     malloc(bufcount*sizeof(float));
                                                                             28
                                                                             29
  buffer2 = (float *)
                     malloc(bufcount*sizeof(float));
                                                                             30
  compute_buf_ptr = buffer1; /* initially point to buffer1 */
                                                                             31
  write_buf_ptr = buffer1; /* initially point to buffer1 */
                                                                             32
                                                                             33
                                                                             34
  /* DOUBLE-BUFFER prolog:
                                                                             35
       compute buffer1; then initiate writing buffer1 to disk
                                                                             36
   *
                                                                             37
   */
                                                                             38
   compute_buffer(compute_buf_ptr, bufcount, &done);
  MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
                                                                             39
                                                                             40
                                                                             41
  /* DOUBLE-BUFFER steady state:
                                                                            42
   * Overlap writing old results from buffer pointed to by write_buf_ptr
   * with computing new results into buffer pointed to by compute_buf_ptr.
                                                                            43
                                                                            44
   * There is always one write-buffer and one compute-buffer in use
                                                                             45
                                                                             46
   * during steady state.
                                                                             47
   */
                                                                             48
  while (!done) {
```

```
1
             TOGGLE_PTR(compute_buf_ptr);
\mathbf{2}
             compute_buffer(compute_buf_ptr, bufcount, &done);
3
             MPI_File_write_all_end(fh, write_buf_ptr, &status);
4
             TOGGLE_PTR(write_buf_ptr);
5
             MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
6
         }
7
8
         /* DOUBLE-BUFFER epilog:
9
               wait for final write to complete.
          *
10
          */
11
         MPI_File_write_all_end(fh, write_buf_ptr, &status);
12
13
14
         /* buffer cleanup */
15
         free(buffer1);
16
         free(buffer2);
17
     }
18
19
     13.9.2 Subarray Filetype Constructor
20
21
22
23
24
25
26
27
28
29
30
^{31}
                                         Process 0
                                                           Process 2
32
                                          Process 1
                                                            Process 3
33
34
                                Figure 13.4: Example array file layout
35
36
37
38
39
40
41
42
43
44
45
                                          MPI_DOUBLE
                                                             Holes
46
47
                       Figure 13.5: Example local array filetype for process 1
48
```

Unofficial Draft for Comment Only

Assume we are writing out a 100×100 2D array of double precision floating point numbers that is distributed among 4 processes such that each process has a block of 25 columns (e.g., process 0 has columns 0–24, process 1 has columns 25–49, etc.; see Figure 13.4). To create the filetypes for each process one could use the following C program (see Section 4.1.3):

```
double subarray[100][25];
  MPI_Datatype filetype;
   int sizes[2], subsizes[2], starts[2];
   int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   sizes[0]=100; sizes[1]=100;
   subsizes[0]=100; subsizes[1]=25;
   starts[0]=0; starts[1]=rank*subsizes[1];
  MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C,
                             MPI_DOUBLE, &filetype);
    Or, equivalently in Fortran:
double precision subarray(100,25)
integer filetype, rank, ierror
integer sizes(2), subsizes(2), starts(2)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
sizes(1)
            = 100
sizes(2)
            = 100
subsizes(1) = 100
subsizes(2) = 25
            = 0
starts(1)
starts(2)
            = rank*subsizes(2)
call MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts, &
           MPI_ORDER_FORTRAN, MPI_DOUBLE_PRECISION,
                                                            &
           filetype, ierror)
```

The generated filetype will then describe the portion of the file contained within the process's subarray with holes for the space taken by the other processes. Figure 13.5 shows the filetype created for process 1.

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



Chapter 14

Tool Support

14.1 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 14.2), which supports the transparent interception and inspection of MPI calls, and the MPI tool information interface (Section 14.3), 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.

14.2 Profiling Interface

14.2.1 Requirements

To meet the requirements for the MPI profiling interface, an implementation of the MPI functions must

1. provide a mechanism through which all of the MPI defined functions, except those allowed as macros (See Section 2.6.4), may be accessed with a name shift. This requires, in C and Fortran, an alternate entry point name, with the prefix PMPI_ for each MPI function in each provided language binding and language support method. For routines implemented as macros, it is still required that the PMPI_ version be supplied and work as expected, but it is not possible to replace at link time the MPI_ version with a user-defined version.

For Fortran, the different support methods cause several specific procedure names. Therefore, several profiling routines (with these specific procedure names) are needed for each Fortran MPI routine, as described in Section 18.1.5.

- 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 implementing it only for the lowest level routines.

Unofficial Draft for Comment Only

 $45 \\ 46$

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 14.2.2 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 its being used wherever it is useful.

As the issues being addressed here are intimately tied up with the way in which executable images are built, which may differ greatly on different machines, the examples given below should be treated solely as one way of implementing the objective of the MPI profiling interface. The actual requirements made of an implementation are those detailed in the Requirements section above, the whole of the rest of this 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 14.2.3 Logic of the Design

⁴⁶⁴⁷ Provided that an MPI implementation meets the requirements above, it is possible for the implementor of the profiling system to intercept the MPI calls that are made by the

Unofficial Draft for Comment Only

user program. She can then collect whatever information she requires before calling the underlying MPI implementation (through its name shifted entry points) to achieve the desired effects.

14.2.4 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.

14.2.5 Profiler Implementation Example

A profiler can accumulate the total amount of data sent by the MPI_SEND function, along with the total elapsed time spent in the function as the following example shows:

```
<sup>5</sup> Example 14.1
```

```
11
     static int totalBytes = 0;
12
     static double totalTime = 0.0;
13
14
     int MPI_Send(const void* buffer, int count, MPI_Datatype datatype,
15
                   int dest, int tag, MPI_Comm comm)
16
     {
17
        double tstart = MPI_Wtime();
                                              /* Pass on all arguments */
18
        int size;
19
                       = PMPI_Send(buffer,count,datatype,dest,tag,comm);
        int result
20
21
        totalTime += MPI Wtime() - tstart;
                                                        /* and time
                                                                              */
22
23
                                           /* Compute size */
        MPI_Type_size(datatype, &size);
24
        totalBytes += count*size;
25
26
        return result;
27
     }
```

28

29

30

35

```
14.2.6 MPI Library Implementation Example
```

If the MPI library is implemented in C on a Unix system, then there are various options, including the two presented here, for supporting the name-shift requirement. The choice between these two options depends partly on whether the linker and compiler support weak symbols.

```
<sup>36</sup> Systems with Weak Symbols
```

³⁷If 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:

4 5

6

7

The effect of this **#pragma** is to define the external symbol MPI_Example as a weak definition. This means that the linker will not complain if there is another definition of the symbol (for instance in the profiling library); however if no other definition exists, then the linker will use the weak definition.

Systems without Weak Symbols

In the absence of weak symbols then one possible solution would be to use the C macro preprocessor as the following example shows:

Example 14.3

```
#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.

14.2.7 Complications

Multiple Counting

Since parts of the MPI library may themselves be implemented using more basic MPI functions (e.g., a portable implementation of the collective operations implemented using point

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $45 \\ 46$

1 to point communications), there is potential for profiling functions to be called from within $\mathbf{2}$ an MPI function that was called from a profiling function. This could lead to "double 3 counting" of the time spent in the inner routine. Since this effect could actually be useful 4 under some circumstances (e.g., it might allow one to answer the question "How much time $\mathbf{5}$ is spent in the point to point routines when they are called from collective functions?"), we 6 have decided not to enforce any restrictions on the author of the MPI library that would 7overcome this. Therefore the author of the profiling library should be aware of this problem, 8 and guard against it. In a single-threaded world this is easily achieved through use of a 9 static variable in the profiling code that remembers if you are already inside a profiling 10 routine. It becomes more complex in a multi-threaded environment (as does the meaning 11of the times recorded).

12

¹³ 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 19 achieved by using wrapper functions on top of the C implementation. The author of the 20profile library then assumes that it is reasonable only to provide profile functions for the C 21binding, since Fortran will eventually call these, and the cost of the wrappers is assumed 22 to be small. However, if the wrapper functions are not in the profiling library, then none 23of the profiled entry points will be undefined when the profiling library is called. Therefore 24none of the profiling code will be included in the image. When the standard MPI library 25is scanned, the Fortran wrappers will be resolved, and will also pull in the base versions of 26the MPI functions. The overall effect is that the code will link successfully, but will not be 27profiled. 28

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**.

33

34 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 18.1.5.

 $40 \\ 41$

14.2.8 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

 $46 \\ 47$

48

- assuming a particular implementation language,
- imposing a run time cost even when no profiling was taking place.

Unofficial Draft for Comment Only

Since one of the objectives of MPI is to permit efficient, low latency implementations, and it is not the business of a standard to require a particular implementation language, we decided to accept the scheme outlined above.

Note, however, that it is possible to use the scheme above to implement a multi-level system, since the function called by the user may call many different profiling functions before calling the underlying MPI function. This capability has been demonstrated in the P^N MPI tool infrastructure [54].

14.3 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 allows the implementation to specify which control variables, performance variables, and events exist. Additionally, the user of the MPI tool information interface can obtain metadata about each available variable or event, such as its datatype, and a textual description. The MPI tool information interface provides the necessary routines to find all variables and events that exist in a particular MPI implementation; to query their properties; to retrieve descriptions about their meaning; to access and, if appropriate, to alter their values; and (in case of events) set callbacks triggered by them.

Variables, events, and categories across connected MPI processes with equivalent names are required to have the same meaning (see the definition of "equivalent" as related to strings in Section 14.3.3). Furthermore, enumerations with equivalent names across connected MPI processes are required to have the same meaning, but are allowed to comprise different enumeration items. Enumeration items that have equivalent names across connected MPI processes in enumerations with the same meaning must also have the same meaning. In order for variables and categories to have the same meaning, routines in the tools information interface that return details for those variables and categories have requirements on what parameters must be identical. These requirements are specified in their respective sections.

Rationale. The intent of requiring the same meaning for entities with equivalent names is to enforce consistency across connected MPI processes. For example, variables describing the number of packets sent on different types of network devices should have different names to reflect their potentially different meanings. (*End of rationale.*)

The MPI tool information interface can be used independently from the MPI communication functionality. In particular, the routines of this interface can be called before MPI is

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

45

 $46 \\ 47$

¹ initialized and after MPI is finalized. In order to support this behavior cleanly, the MPI tool
 ² information interface uses separate initialization and finalization routines. All identifiers
 ³ used in the MPI tool information interface have the prefix MPI_T_.

⁴ On success, all MPI tool information interface routines return MPI_SUCCESS, otherwise ⁵ they return an appropriate and unique return code indicating the reason why the call was not ⁶ successfully completed. Details on return codes can be found in Section 14.3.10. However, ⁷ unsuccessful calls to the MPI tool information interface are not fatal and do not impact the ⁸ execution of subsequent MPI routines.

⁹ Since the MPI tool information interface primarily focuses on tools and support li-¹⁰ braries, MPI implementations are only required to provide C bindings for functions and ¹¹ constants introduced in this section. Except where otherwise noted, all conventions and ¹² principles governing the C bindings of the MPI API also apply to the MPI tool information ¹³ interface, which is available by including the mpi.h header file. All routines in this interface ¹⁴ 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.*)

14.3.1 Verbosity Levels

29The MPI tool information interface provides access to internal configuration and perfor-30 mance information through a set of control and performance variables defined by the MPI 31 implementation. Since some implementations may export a large number of variables, 32 variables are classified by a verbosity level that categorizes both their intended audience 33 (end users, performance tuners or MPI implementors) and a relative measure of level of 34 detail (basic, detailed or all). These verbosity levels are described by a single integer. 35 Table 14.1 lists the constants for all possible verbosity levels. The values of the con-36 stants are monotonic in the order listed in the table; i.e., MPI_T_VERBOSITY_USER_BASIC 37 < MPI_T_VERBOSITY_USER_DETAIL < ... < MPI_T_VERBOSITY_MPIDEV_ALL. 38

39

40

15

16

17

18

19

20

21

22

23

24

25

26 27

28

14.3.2 Binding MPI Tool Information Interface Variables to MPI Objects

 41 Each MPI tool information interface variable provides access to a particular control setting 42or performance property of the MPI implementation. A variable may refer to a specific 43 MPI object such as a communicator, datatype, or one-sided communication window, or the 44variable may refer more generally to the MPI environment of the process. Except for the 45last case, the variable must be bound to exactly one MPI object before it can be used. 46Table 14.2 lists all MPI object types to which an MPI tool information interface variable 47can be bound, together with the matching constant that MPI tool information interface 48routines return to identify the object type.

Unofficial Draft for Comment Only

MPI_T_VERBOSITY_USER_BASIC	Basic information of interest to users	1
MPI_T_VERBOSITY_USER_DETAIL	Detailed information of interest to users	2
MPI_T_VERBOSITY_USER_ALL	All remaining information of interest to users	3
MPI_T_VERBOSITY_TUNER_BASIC	Basic information required for tuning	4
MPI_T_VERBOSITY_TUNER_DETAIL	Detailed information required for tuning	5
MPI_T_VERBOSITY_TUNER_ALL	All remaining information required for tuning	6
MPI_T_VERBOSITY_MPIDEV_BASIC	Basic information for MPI implementors	7
MPI_T_VERBOSITY_MPIDEV_DETAIL	Detailed information for MPI implementors	8
MPI_T_VERBOSITY_MPIDEV_ALL	All remaining information for MPI implementors	9
		10

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 14.1: MPI tool information interface verbosity levels

Table 14.2: Constants to identify associations of variables

Some variables have meanings tied to a specific MPI object. Examples Rationale. 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.)

Convention for Returning Strings 14.3.3

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

Unofficial Draft for Comment Only

 31

¹ characters into the buffer, followed by a null terminator. If the returned string's length is ² greater than or equal to n, the string will be truncated to n-1 characters. In this case, the ³ length of the string plus one (for the terminating null character) is returned in the length ⁴ argument. If the user passes the null pointer as the buffer argument or passes 0 as the ⁵ length argument, the function does not return the string and only returns the length of the ⁶ string plus one in the length argument. If the user passes the null pointer as the length ⁷ argument, the buffer argument is ignored and nothing is returned.

⁸ MPI implementations behave as if they have an internal character array that is copied ⁹ to the output character array supplied by the user. Such output strings are only defined ¹⁰ to be equivalent if their notional source-internal character arrays are identical (up to and ¹¹ including the null terminator), even if the output string is truncated due to a small input ¹² length parameter n.

 $13 \\ 14$

15

18 19

25

26

14.3.4 Initialization and Finalization

The MPI tool information interface requires a separate set of initialization and finalization
 routines.

MPI_T_INIT_THREAD(required, provided)

1	IN	required	desired level of thread support (integer)
2	OUT	provided	provided level of thread support (integer)

C binding

int MPI_T_init_thread(int required, int *provided)

All programs or tools that use the MPI tool information interface must initialize the 27MPI tool information interface in the processes that will use the interface before calling 28any other of its routines. A user can initialize the MPI tool information interface by calling 29MPI_T_INIT_THREAD, which can be called multiple times. In addition, this routine initial-30 izes the thread environment for all routines in the MPI tool information interface. Calling 31 this routine when the MPI tool information interface is already initialized has no effect 32 beyond increasing the reference count of how often the interface has been initialized. The 33 argument required is used to specify the desired level of thread support. The possible values 34and their semantics are identical to the ones that can be used with MPI_INIT_THREAD 35 listed in Section 10.6. The call returns in provided information about the actual level of 36 thread support that will be provided by the MPI implementation for calls to MPI tool 37 information interface routines. It can be one of the four values listed in Section 10.6. 38

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.

47 48

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 10.3), the requested and provided thread level for MPI_T_INIT_THREAD may influence the behavior and return values of MPI_SESSION_INIT (see Section 10.3), 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.

14.3.5 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.

Unofficial Draft for Comment Only

1 2

3

4

5 6

7

8

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

652 CHAPTER 14. TOOL SUPPORT MPI_INT MPI_INT32_T MPI_INT64_T MPI_UNSIGNED MPI_UNSIGNED_LONG MPI_UNSIGNED_LONG_LONG MPI_UINT32_T MPI_UINT64_T MPI_COUNT MPI_CHAR MPI_DOUBLE Table 14.3: MPI datatypes that can be used by the MPI tool information interface This would cause unnecessary complexity in the implementation of tools based on the MPI tool information interface. (End of rationale.) The MPI tool information interface only relies on a subset of the basic MPI datatypes and does not use any derived MPI datatypes. Table 14.3 lists all MPI datatypes that can be returned by the MPI tool information interface to represent its variables. The use of the datatype MPI_CHAR in the MPI tool information interface implies a nullterminated character array, i.e., a string in the C language. If a variable has type MPI_CHAR, the value of the count parameter returned by MPI_T_CVAR_HANDLE_ALLOC and MPI_T_PVAR_HANDLE_ALLOC must be large enough to include any valid value, including its terminating null character. The contents of returned MPI_CHAR arrays are only defined from index 0 through the location of the first null character. Rationale. The MPI tool information interface requires a significantly simpler type system than MPI itself. Therefore, only its required subset must be present before MPI initialization and MPI implementations do not need to initialize the complete MPI datatype system. (End of rationale.) For variables of type MPI_INT, an MPI implementation can provide additional information by associating names with a fixed number of values. We refer to this information in the following as an enumeration. In this case, the respective calls that provide additional metadata for each control or performance variable, i.e., MPI_T_CVAR_GET_INFO (Section 14.3.6), MPI_T_PVAR_GET_INFO (Section 14.3.7), and MPI_T_EVENT_GET_INFO (Section 14.3.8), return a handle of type MPI_T_enum that can be passed to the following functions to extract additional information. Thus, the MPI implementation can describe variables with a fixed set of values that each represents a particular state. Each enumeration type can have N different values, with a fixed N that can be queried using MPI_T_ENUM_GET_INFO.

1

2

3

4

5

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

IV		ow_GET_INTO(enumrype, nu	in, name, name_ien)	
	IN	enumtype	enumeration to be queried (handle)	2
	IIN	enunitype	chandles be queried (nandle)	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
	INCOT	name_len	length of the string and/or buller 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 14.3.3.

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 name (integer)	34

C binding

The arguments name and name_len are used to return the name of the enumeration item as described in Section 14.3.3.

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
```

14.3.6 Control Variables

The routines described in this section of the MPI tool information interface specification focus on the ability to list, query, and possibly set control variables exposed by the MPI implementation. These variables can typically be used by the user to fine tune properties and configuration settings of the MPI implementation. On many systems, such variables can be set using environment variables, although other configuration mechanisms may be available, such as configuration files or central configuration registries. A typical example that is available in several existing MPI implementations is the ability to specify an "eager limit," i.e., an upper bound on the size of messages sent or received using an eager protocol.

10 11 12

1

2

3

4

5

6

7

8

9

Control Variable Query Functions

¹³ An MPI implementation exports a set of N control variables through the MPI tool infor-¹⁴ mation interface. If N is zero, then the MPI implementation does not export any control ¹⁵ variables, otherwise the provided control variables are indexed from 0 to N-1. This index ¹⁶ number is used in subsequent calls to identify the individual variables.

An MPI implementation is allowed to increase the number of control variables during the execution of an MPI application when new variables become available through dynamic loading. However, MPI implementations are not allowed to change the index of a control variable or to delete a variable once it has been added to the set. When a variable becomes inactive, e.g., through dynamic unloading, accessing its value should return a corresponding error code.

Advice to users. While the MPI tool information interface guarantees that indices or variable properties do not change during a particular run of an MPI program, it does not provide a similar guarantee between runs. (End of advice to users.)

The following function can be used to query the number of control variables, num_cvar:

MPI_T_CVAR_GET_NUM(num_cvar)

OUT num_cvar

returns number of control variables (integer)

C binding

int MPI_T_cvar_get_num(int *num_cvar)

The function $\mathsf{MPI_T_CVAR_GET_INFO}$ provides access to additional information for each variable.

 24

25

26

27

28 29 30

31 32

33 34

35

36

37

MPI_T_C	VAR_GET_INFO(cvar_index,	name, name_len, verbosity, datatype, enumtype, desc,	1
	desc_len, bind, scope)		2
IN	cvar_index	index of the control variable to be queried, value	3
		between 0 and $num_cvar - 1$ (integer)	4
OUT	name	buffer to return the string containing the name of the	5 6
		control variable (string)	7
INOUT	name_len	length of the string and/or buffer for name (integer)	8
OUT	verbosity	verbosity level of this variable (integer)	9
	-		10
OUT	datatype	MPI datatype of the information stored in the	11
		control variable (handle)	12
OUT	enumtype	optional descriptor for enumeration information	13
		(handle)	14
OUT	desc	buffer to return the string containing a description of	15
		the control variable (string)	16
INOUT	desc_len	length of the string and/or buffer for $desc$ (integer)	17 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
		(O)	23
C bindin	σ		24

int MPI_T_cvar_get_info(int cvar_index, char *name, int *name_len, int *verbosity, MPI_Datatype *datatype, MPI_T_enum *enumtype, char *desc, int *desc_len, int *bind, int *scope)

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 14.3.3.

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 14.3.1).

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 14.3.5. 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.

47The arguments desc and desc_len are used to return a description of the control variable 48 as described in Section 14.3.3.

Unofficial Draft for Comment Only

26

27

28

2930

31

32 33

34

35

36 37

38 39

40 41

42

43 44

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 14.3.2).

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 14.4.

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.

.9			
)	Scor	pe Constant	Description
1	MPI.	_T_SCOPE_CONSTANT	read-only, value is constant
2	MPI.	_T_SCOPE_READONLY	read-only, cannot be written, but can change
3	MPI.	_T_SCOPE_LOCAL	may be writeable, writing is a local operation
4	MPI.	_T_SCOPE_GROUP	may be writeable, must be set to consistent values
5			across a group of connected MPI processes
6	MPI.	_T_SCOPE_GROUP_EQ	may be writeable, must be set to the same value
7			across a group of connected MPI processes
8	MPI.	_T_SCOPE_ALL	may be writeable, must be set to consistent values
9			across all connected MPI processes
0	MPI.	_T_SCOPE_ALL_EQ	may be writeable, must be set to the same value
31			across all connected MPI processes
32			
33		Table 1	14. Seconds for control variables
34		Table 1	4.4: Scopes for control variables
5			
6	Adv	ice to users. The s	cope of a variable only indicates if a variable might
7			rantee that it can be changed at any time. (End of ad
8		sers.)	
9			
0			
1			
2	MPI_T_C	VAR_GET_INDEX(name	e, cvar_index)
3	IN	name	name of the control variable (string)
4	OUT	cvar_index	index of the control variable (integer)
5	001		mass of the control variable (mucger)
6	C him i'	_	
7	C bindin		
8	INT MPI_	<pre>1_cvar_get_index(co)</pre>	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: Printing All Control Variables

Example 14.4 The following example shows how the MPI tool information interface can be used to query and to print the names of all available control variables.

```
#include <stdio.h>
                                                                                       23
                                                                                       ^{24}
#include <stdlib.h>
                                                                                       25
#include <mpi.h>
                                                                                       26
int main(int argc, char *argv[]) {
                                                                                       27
                                                                                       28
  int i, err, num, namelen, bind, verbose, scope;
                                                                                       29
  int threadsupport;
  char name[100];
                                                                                       30
                                                                                       31
  MPI_Datatype datatype;
                                                                                       32
                                                                                       33
  err=MPI_T_init_thread(MPI_THREAD_SINGLE,&threadsupport);
                                                                                       34
  if (err!=MPI_SUCCESS)
                                                                                       35
    return err;
                                                                                       36
                                                                                       37
  err=MPI_T_cvar_get_num(&num);
                                                                                       38
  if (err!=MPI_SUCCESS)
                                                                                       39
    return err;
                                                                                       40
                                                                                       41
  for (i=0; i<num; i++) {</pre>
                                                                                       42
    namelen=100;
    err=MPI_T_cvar_get_info(i, name, &namelen,
                                                                                       43
                                                                                       44
             &verbose, &datatype, NULL,
             NULL, NULL, /*no description */
                                                                                       45
                                                                                       46
             &bind, &scope);
                                                                                       47
    if (err!=MPI_SUCCESS && err!=MPI_T_ERR_INVALID_INDEX) return err;
                                                                                       48
    printf("Var %i: %s\n", i, name);
```

1

 $\mathbf{2}$

3

4

5 6

7

8

9

10

11

12

13

14

15

16 17

18 19 20

21

```
1
        }
2
3
        err=MPI_T_finalize();
4
        if (err!=MPI_SUCCESS)
5
          return 1;
6
        else
7
          return 0;
8
      }
9
10
      Handle Allocation and Deallocation
11
      Before reading or writing the value of a variable, a user must first allocate a handle of type
12
      MPI_T_cvar_handle for the variable by binding it to an MPI object (see also Section 14.3.2).
13
14
            Rationale.
                         Handles used in the MPI tool information interface are distinct from
15
           handles used in the remaining parts of the MPI standard because they must be usable
16
           before MPI is initialized and after MPI is finalized. Further, accessing handles, in
17
           particular for performance variables, can be time critical and having a separate handle
18
           space enables optimizations. (End of rationale.)
19
20
21
      MPI_T_CVAR_HANDLE_ALLOC(cvar_index, obj_handle, handle, count)
22
23
        IN
                  cvar_index
                                                index of control variable for which handle is to be
24
                                                allocated (index)
25
        IN
                  obj_handle
                                                reference to a handle of the MPI object to which this
26
                                                variable is supposed to be bound (pointer)
27
        OUT
                                                allocated handle (handle)
                  handle
28
29
        OUT
                  count
                                                number of elements used to represent this variable
30
                                                (integer)
^{31}
32
      C binding
33
      int MPI_T_cvar_handle_alloc(int cvar_index, void *obj_handle,
34
                      MPI_T_cvar_handle *handle, int *count)
35
36
          This routine binds the control variable specified by the argument index to an MPI object.
37
      The object is passed in the argument obj_handle as an address to a local variable that stores
38
      the object's handle. The argument obj_handle is ignored if the MPI_T_CVAR_GET_INFO
39
      call for this control 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 success-
40
41
      ful return, count contains the number of elements (of the datatype returned by a previous
42
      MPI_T_CVAR_GET_INFO call) used to represent this variable.
43
           Advice to users. The count can be different based on the MPI object to which the
44
           control variable was bound. For example, variables bound to communicators could
45
           have a count that matches the size of the communicator.
46
47
           It is not portable to pass references to predefined MPI object handles, such as
48
           MPI_COMM_WORLD to this routine, since their implementation depends on the MPI
```

library. Instead, such object handles should be stored in a local variable and the address of this local variable should be passed into MPI_T_CVAR_HANDLE_ALLOC. (*End of advice to users.*)

The value of cvar_index should be in the range 0 to num_cvar -1, where num_cvar is the number of available control variables as determined from a prior call to MPI_T_CVAR_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_CVAR_GET_INFO.

with the	type returned in the bind argu	ument in a prior call to MPI_T_CVAR_GET_INFO.	8
			9
MPI_T_C	VAR_HANDLE_FREE(handle)		10 11
INOUT	handle	handle to be freed (handle)	12
	nanare	hundre to be need (hundre)	13
C bindir	זפ		14
	.T_cvar_handle_free(MPI_T	_cvar_handle *handle)	15
call $MPI_$	T_CVAR_HANDLE_FREE to	, a user of the MPI tool information interface should free the handle and the associated resources in the	16 17 18
-		l return, MPI sets the handle to	19
	/AR_HANDLE_NULL.		20
Control V	ariable Access Functions		21 22
Control v			22
			24
MPI_T_C	VAR_READ(handle, buf)		25
IN	handle	handle to the control variable to be read (handle)	26
			27
OUT	buf	initial address of storage location for variable value (choice)	28 29
			30
C bindin	3		31
int MPI_	T_cvar_read(MPI_T_cvar_ha	andle handle, void *buf)	32
	-	ontrol variable identified by the argument handle and	33 34
		by the parameter buf . The user must ensure that the	35
		the entire value of the control variable (based on the	36
		or corresponding calls to MPI_T_CVAR_GET_INFO	37
and MPI_	T_CVAR_HANDLE_ALLOC, 1	espectively).	38
			39
MPI_T_C	VAR_WRITE(handle, buf)		40
IN	handle	handle to the control variable to be written (handle)	41
IN	buf	initial address of storage location for variable value	42 43
11 1	bui	(choice)	43 44
			45
C bindir	ng		46
	•	nandle handle, const void *buf)	47

 $\mathbf{2}$

 $\mathbf{5}$

1	This routine sets the value of the control variable identified by the argument handle to
2	the data stored in the buffer identified by the parameter buf . The user must ensure that the
3	buffer is of the appropriate size to hold the entire value of the control variable (based on the
4	returned datatype and count from prior corresponding calls to MPI_T_CVAR_GET_INFO
5	and MPI_T_CVAR_HANDLE_ALLOC, respectively).
6	If the variable has a global scope (as returned by a prior corresponding
7	MPI_T_CVAR_GET_INFO call), any write call to this variable must be issued by the user
8	in all connected (as defined in Section 10.10.4) MPI processes. If the variable has group
9	scope, any write call to this variable must be issued by the user in all MPI processes in
10	the group, which must be described by the MPI implementation in the description by the
11	MPI_T_CVAR_GET_INFO.
12	In both cases, the user must ensure that the writes in all participating MPI processes
13	are consistent. If the scope is either MPI_T_SCOPE_ALL_EQ or MPI_T_SCOPE_GROUP_EQ
14	this means that the variable in all connected MPI processes or MPI processes of the group,
15	respectively, must be set to the same value.
16	If it is not possible to change the variable at the time the call is made, the function
17	returns either MPI_T_ERR_CVAR_SET_NOT_NOW, if there may be a later time at which the
18	variable could be set, or MPI_T_ERR_CVAR_SET_NEVER, if the variable cannot be set for the
19	remainder of the application's execution.
20	
21	Example: Reading the Value of a Control Variable
22	
23	Example 14.5 The following example shows a routine that can be used to query the
24	value with a control variable with a given index. The example assumes that the variable is
25	intended to be bound to an MPI communicator.
26	interficed to be bound to an with communicator.
27	<pre>int getValue_int_comm(int index, MPI_Comm comm, int *val) {</pre>
28	int err, count;
29	MPI_T_cvar_handle handle;
30	
31	/* This example assumes that the variable index */
32	/* can be bound to a communicator */
33	
34	err=MPI_T_cvar_handle_alloc(index, &comm, &handle, &count);
35	if (err!=MPI_SUCCESS) return err;
36	
37 38	/* The following assumes that the variable is $*/$
39	<pre>/* represented by a single integer */</pre>
40	
41	<pre>err=MPI_T_cvar_read(handle,val);</pre>
42	if (err!=MPI_SUCCESS) return err;
43	
44	err=MPI_T_cvar_handle_free(&handle);
45	return err;
46	}
47	
48	

14.3.7 Performance Variables

The following section focuses on the ability to list and to query performance variables provided by the MPI implementation. Performance variables provide insight into MPI implementation specific internals and can represent information such as the state of the MPI implementation (e.g., waiting blocked, receiving, not active), aggregated timing data for submodules, or queue sizes and lengths.

Rationale. The interface for performance variables is separate from the interface for control variables, since performance variables have different requirements and parameters. By keeping them separate, the interface provides cleaner semantics and allows for more performance optimization opportunities. (*End of rationale.*)

Some performance variables and classes refer to *events*. In general, such events describe state transitions within software or hardware related to the performance of an MPI application. The events offered through the callback-driven event-notification interface described in Section 14.3.8 also refer to such state transitions; however, the set of state transitions referred to by performance variables and events as described in Section 14.3.8 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 14.3.5. 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 ⁴⁵ level of a resource. The value of a variable of this class can change at any time to match ⁴⁶ the current utilization level of the resource. Values returned from variables in this class ⁴⁷ are non-negative and represented by one of the following datatypes: MPI_UNSIGNED, ⁴⁸

Unofficial Draft for Comment Only

 $\mathbf{2}$

 24

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.

12 13

1

2

3

4

5

6

7

8

9

10

11

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

48

• MPI_T_PVAR_CLASS_PERCENTAGE

The value of a performance variable in this class represents the percentage utilization of a finite resource. The value of a variable of this class can change at any time to match the current utilization level of the resource. It will be returned as an MPI_DOUBLE datatype. The value must always be between 0.0 (resource not used at all) and 1.0 (resource completely used). The starting value is the current percentage utilization level of the resource at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

MPI_T_PVAR_CLASS_HIGHWATERMARK

A performance variable in this class represents a value that describes the high watermark utilization of a resource. The value of a variable of this class is 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 implementations must ensure that variables of this class cannot overflow.

• MPI_T_PVAR_CLASS_LOWWATERMARK

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 implementations must ensure that variables of this class cannot overflow.

• MPI_T_PVAR_CLASS_COUNTER

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. The starting value for variables of this class is 0. Variables of this class can overflow.

• MPI_T_PVAR_CLASS_AGGREGATE

The value of a performance variable in this class is an an aggregated value that

Unofficial Draft for Comment Only

represents a sum of arguments processed during a specific event (e.g., the amount of memory allocated by all memory allocations). This class is similar to the counter class, but instead of counting individual events, the value can be incremented by arbitrary amounts. 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. Variables of this class can overflow.

• 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 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

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

The fr each varial	ole. /AR_GET_INFO(pvar_index, na	var) NFO provides access to additional information for ame, name_len, verbosity, var_class, datatype, , bind, readonly, continuous, atomic)
IN	pvar_index	index of the performance variable to be queried between 0 and num_pvar - 1 (integer)
OUT	name	buffer to return the string containing the name of the performance variable (string)
INOUT	name_len	length of the string and/or buffer for name (integer)
OUT	verbosity	verbosity level of this variable (integer)
OUT	var_class	class of performance variable (integer)
OUT	datatype	MPI datatype of the information stored in the performance variable (handle)
OUT	enumtype	optional descriptor for enumeration information (handle)
OUT	desc	buffer to return the string containing a description of the performance variable (string)
INOUT	desc_len	length of the string and/or buffer for $desc$ (integer)
OUT	bind	type of MPI object to which this variable must be bound (integer)
OUT	readonly	flag indicating whether the variable can be written/reset (integer)
OUT	continuous	flag indicating whether the variable can be started and stopped or is continuously active (integer)
Ουτ	atomic	flag indicating whether the variable can be atomically read and reset (integer)
C binding	y.	
	_pvar_get_info(int pvar_: int *verbosity, int MPI_T_enum *enumtype	index, char *name, int *name_len, *var_class, MPI_Datatype *datatype, , char *desc, int *desc_len, int *bind, continuous, int *atomic)
calls to thi	s routine that query informat n. An MPI implementation is	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.

CHAPTER 14. TOOL SUPPORT

 2

 $\mathbf{5}$

 $\overline{7}$

 24

Unofficial Draft for Comment Only

variable as described in Section 14.3.3. If completed successfully, the routine is required

If any OUT parameter to MPI_T_PVAR_GET_INFO is a NULL pointer, the implementa-

The arguments name and name_len are used to return the name of the performance

tion will ignore the parameter and not return a value for the parameter.

to return a name of at least length one.

The argument verbosity returns the verbosity level of the variable (see Section 14.3.1). The class of the performance variable is returned in the parameter var_class. The class must be one of the constants defined in Section 14.3.7.

The combination of the name and the class of the performance variable must be unique with respect to all other names for performance variables used by the MPI implementation.

Advice to implementors. Groups of variables that belong closely together, but have different classes, can have the same name. This choice is useful, e.g., to refer to multiple variables that describe a single resource (like the level, the total size, as well as high and low watermarks). (*End of advice to implementors.*)

The argument datatype returns the MPI datatype that is used to represent the performance variable.

If the variable is of type MPI_INT, MPI can optionally specify an enumeration for the values represented by this variable and return it in enumtype. In this case, MPI returns an enumeration identifier, which can then be used to gather more information as described in Section 14.3.5. 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 14.3.2).

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

Unofficial Draft for Comment Only

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

1

 $\mathbf{2}$

38 39 40

45

46 47

provided by the caller, and pvar_index is returned by the MPI implementation. The name
 parameter is a string terminated with a null character.

This routine returns MPI_SUCCESS on success and returns MPI_T_ERR_INVALID_NAME
 if name does not match the name of any performance variable of the specified var_class
 provided by the implementation at the time of the call.

Rationale. This routine is provided to enable fast retrieval of performance variables by a tool, assuming it knows the name of the variable for which it is looking. The number of variables exposed by the implementation can change over time, so it is not possible for the tool to simply iterate over the list of variables once at initialization. Although using MPI implementation specific variable names is not portable across MPI implementations, tool developers may choose to take this route for lower overhead at runtime because the tool will not have to iterate over the entire set of variables to find a specific one. (End of rationale.)

¹⁶ 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 session. Subsequent calls that access performance variables can then be made within the context of this session. Any call executed in a session must not influence the results in any other session.

23 24

27

30

 31

32 33 34

35

36 37

38

39 40

41

42

43

6 7

8

9

10

11

12

13

14 15

MPI_T_PVAR_SESSION_CREATE(session)

²⁵₂₆ OUT session identifier of performance session (handle)

28 C binding

29 int MPI_T_pvar_session_create(MPI_T_pvar_session *session)

This call creates a new session for accessing performance variables and returns a handle for this session in the argument session of type MPI_T_pvar_session.

MPI_T_PVAR_SESSION_FREE(session)

session

identifier of performance experiment session (handle)

C binding

INOUT

int MPI_T_pvar_session_free(MPI_T_pvar_session *session)

This call frees an existing session. Calls to the MPI tool information interface can no longer be made within the context of a session after it is freed. On a successful return, MPI sets the session identifier to MPI_T_PVAR_SESSION_NULL.

⁴⁴ 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 14.3.2).

1 1	····_· / / //		pval_index, obj_nandić, nandić, countj	
	IN	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)	1

MPI_T_PVAR_HANDLE_ALLOC(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 session identified by the parameter 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 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 session as IN or INOUT arguments, if the handle argument passed in is not associated with the session argument, MPI_T_ERR_INVALID_HANDLE is returned.

MPI_T_PVAR_HANDLE_FREE(session, handle)

IN	session	identifier of performance experiment session (handle)	
INOUT	handle	handle to be freed (handle)	

C binding

Unofficial Draft for Comment Only

	000			CHAFTER 14. TOOL SUFFORT
$\frac{1}{2}$	int MPI_	-	free(MPI_T_p wr_handle *ha	ovar_session session, andle)
3 4 5 6 7	call MPI_ rameter s	n a handle is no lo T_PVAR_HANDI ession and the as	onger needed, _E_FREE to fr ssociated resou	a user of the MPI tool information interface should ee the handle in the session identified by the pa- urces in the MPI implementation. On a successful /AR_HANDLE_NULL.
8 9	Starting a	nd Stopping of Pe	erformance Var	iables
10 11 12 13 14 15 16	continuou any time, stopped s	sly operating one but they cannot	te a handle has be started or andle has beer	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 a allocated; their values are not updated until they
17	MPI_T_P	VAR_START(sess	sion, handle)	
18	IN	session		identifier of performance experiment session (handle)
19 20	IN	handle		handle of a performance variable (handle)
21 22 23 24 25 26 27	This rameter h If the	T_pvar_start(M functions starts andle in the sessi e constant MPI_T	the performar on identified k _PVAR_ALL_H	ession session, MPI_T_pvar_handle handle) ace variable with the handle identified by the pa- by the parameter session. ANDLES is passed in handle, the MPI implementa- in the session identified by the parameter session for
28 29 30 31 32 33	which har ables are otherwise	ndles have been al started successfu MPI_T_ERR_PVA	located. In the lly (even if the R_NO_STARTS	is case, the routine returns MPI_SUCCESS if all vari- ere are no non-continuous variables to be started), STOP is returned. Continuous variables and vari- d when MPI_T_PVAR_ALL_HANDLES is specified.
34	MPI_T_P	VAR_STOP(sessi	on, handle)	
35 36	IN	session		identifier of performance experiment session (handle)
37	IN	handle		handle of a performance variable (handle)
38 39 40 41 42 43 44 45 46 47 48	This eter hand If the tion atter for which all variab	T_pvar_stop(MP functions stops the le in the session i e constant MPI_T npts to stop all handles have be les are stopped s	he performanc dentified by th _PVAR_ALL_H variables with pen allocated. uccessfully (ev	ssion session, MPI_T_pvar_handle handle) e variable with the handle identified by the param- ne parameter session. ANDLES is passed in handle, the MPI implementa- in the session identified by the parameter session In this case, the routine returns MPI_SUCCESS if yen if there are no non-continuous variables to be NO_STARTSTOP is returned. Continuous variables

CHAPTER 14. TOOL SUPPORT

and variables that are already stopped are ignored when $\mathsf{MPI_T_PVAR_ALL_HANDLES}$ is specified.

Performance Variable Access Functions

MPI_T_PVAR_READ(session, handle, buf)

IN	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 session identified by the parameter 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.

MPI_T_PVAR_WRITE(session, handle, buf)

IN	session	identifier of performance experiment session (handle)
IN	handle	handle of a performance variable (handle)
IN	buf	initial address of storage location for variable value (choice)

C binding

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 session identified by the parameter session. The value to be written is passed in the buffer identified by the parameter buf. The user must ensure that the buffer is of the appropriate size to hold the entire value of the performance variable (based on the datatype and count returned by the corresponding previous calls to MPI_T_PVAR_GET_INFO and MPI_T_PVAR_HANDLE_ALLOC, respectively).

If it is not possible to change the variable, the function returns MPI_T_ERR_PVAR_NO_WRITE.

The constant MPI_T_PVAR_ALL_HANDLES cannot be used as an argument for the function MPI_T_PVAR_WRITE.

Unofficial Draft for Comment Only

 $\mathbf{2}$

 24

1	MPI_T_	PVAR_RESET(sessi	ion, handle)
2 3	IN	session	identifier of performance experiment session (handle)
4	IN	handle	handle of a performance variable (handle)
5	<u> </u>		
6 7	C bindi	-	PI_T_pvar_session session, MPI_T_pvar_handle handle)
8	IIIC MFI	_1_pvar_reset(M	PI_1_pvar_session session, MF1_1_pvar_nandre nandre)
9			SET call sets the performance variable with the handle identified
10			its starting value specified in Section 14.3.7. If it is not possible
11	-	· · ·	function returns MPI_T_ERR_PVAR_NO_WRITE.
12			PVAR_ALL_HANDLES is passed in handle, the MPI implementation
13	-		bles within the session identified by the parameter session for allocated. In this case, the routine returns MPI_SUCCESS if all
14			fully (even if there are no valid handles or all are read-only),
15			R_NO_WRITE is returned. Read-only variables are ignored when
16		VAR_ALL_HANDLE	
17 18			
19			
20	MPI_I_		T(session, handle, buf)
21	IN	session	identifier of performance experiment session (handle)
22	IN	handle	handle of a performance variable (handle)
23	OUT	buf	initial address of storage location for variable value
24 25			(choice)
26	~		
27	C bindi	U	
28	int MPI		et(MPI_T_pvar_session session,
29			r_handle handle, void *buf)
30		v	ombines the functionality of $MPI_T_VAR_READ$ and
31			the same semantics as if these two calls were called separately.
32		_	is variable are not supported, this routine returns
33		RR_PVAR_NO_ATO	VAR_ALL_HANDLES cannot be used as an argument for the func-
34 35		I_T_PVAR_READR	
36			
37	Ad	lvice to implemento	<i>prs.</i> Sampling-based tools rely on the ability to call the MPI tool
38	inf	ormation interface.	, in particular routines to start, stop, read, write, and reset per-
39	for	mance variables, fr	com any program context, including asynchronous contexts such
40		-	IPI implementations should strive, if possible in their particular
41		,	le these usage scenarios for all or a subset of the routines men-
42		-	lementing only a subset, the read, write, and reset routines are
43		, e	itical for sampling based tools. An MPI implementation should
44			restrictions on the program contexts in which the MPI tool infor-
45			be used. Restrictions might include guaranteeing usage outside le a specific set of signals. Any restrictions could be documented,
46 47		-	the description returned by MPI_T_PVAR_GET_INFO. (<i>End of</i>
48		vice to implemento	
		*	/

Rationale. All routines to read, to write or to reset performance variables require the 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: Tool to Detect Receives with Long Unexpected Message Queues

Example 14.6 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 session and a handle for the variable with the found index, and starts the performance variable.

```
23
#include <stdio.h>
                                                                                       ^{24}
#include <stdlib.h>
                                                                                       25
#include <string.h>
                                                                                       26
#include <assert.h>
                                                                                       27
#include <mpi.h>
                                                                                       28
                                                                                       29
/* Global variables for the tool
                                                                                       30
static MPI_T_pvar_session session;
                                                                                       31
static MPI_T_pvar_handle handle;
                                                                                       32
                                                                                       33
int MPI_Init(int *argc, char ***argv ) {
                                                                                       34
      int err, num, i, index, namelen, verbosity;
                                                                                       35
      int var_class, bind, threadsup;
                                                                                       36
      int readonly, continuous, atomic, count;
                                                                                       37
      char name [18];
                                                                                       38
      MPI_Comm comm;
                                                                                       39
      MPI_Datatype datatype;
                                                                                       40
      MPI_T_enum enumtype;
                                                                                       41
                                                                                       42
      err=PMPI_Init(argc, argv);
                                                                                       43
      if (err!=MPI_SUCCESS) return err;
                                                                                       44
                                                                                       45
      err=PMPI_T_init_thread(MPI_THREAD_SINGLE, &threadsup);
                                                                                       46
      if (err!=MPI_SUCCESS) return err;
                                                                                       47
```

 $\mathbf{2}$

```
1
           err=PMPI_T_pvar_get_num(&num);
2
            if (err!=MPI_SUCCESS) return err;
3
            index=-1;
4
            i=0;
           while ((i<num) && (index<0) && (err==MPI_SUCCESS)) {</pre>
5
6
                  /* Pass a buffer that is at least one character longer than */
7
                  /* the name of the variable being searched for to avoid */
8
                  /* finding variables that have a name that has a prefix \ast/
9
                  /* equal to the name of the variable being searched. */
10
                  namelen=18;
11
                  err=PMPI_T_pvar_get_info(i, name, &namelen, &verbosity,
12
                           &var_class, &datatype, &enumtype, NULL, NULL, &bind,
13
                           &readonly, &continuous, &atomic);
14
                  if (strcmp(name, "MPI_T_UMQ_LENGTH")==0) index=i;
15
                  i++; }
            if (err!=MPI_SUCCESS) return err;
16
17
18
            /* this could be handled in a more flexible way for a generic tool */
19
            assert(index>=0);
20
            assert(var_class==MPI_T_PVAR_CLASS_LEVEL);
21
           assert(datatype==MPI_INT);
22
           assert(bind==MPI_T_BIND_MPI_COMM);
23
24
           /* Create a session */
25
            err=PMPI_T_pvar_session_create(&session);
26
            if (err!=MPI_SUCCESS) return err;
27
           /* Get a handle and bind to MPI_COMM_WORLD */
28
29
            comm=MPI_COMM_WORLD;
30
           err=PMPI_T_pvar_handle_alloc(session, index, &comm, &handle, &count);
31
            if (err!=MPI_SUCCESS) return err;
32
33
           /* this could be handled in a more flexible way for a generic tool */
34
           assert(count==1);
35
36
            /* Start variable */
37
           err=PMPI_T_pvar_start(session, handle);
38
            if (err!=MPI_SUCCESS) return err;
39
40
           return MPI_SUCCESS;
41
     }
42
43
     Part 2 — Testing the Queue Lengths During Receives: During every receive operation, the
44
     tool reads the unexpected queue length through the matching performance variable and
45
     compares it against a predefined threshold.
46
47
     #define THRESHOLD 5
48
```

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Status *status)
{
        int value, err;
        if (comm==MPI_COMM_WORLD) {
                err=PMPI_T_pvar_read(session, handle, &value);
                if ((err==MPI_SUCCESS) && (value>THRESHOLD))
                {
                         /* tool identified receive called with long UMQ */
                         /* execute tool functionality, */
                         /* e.g., gather and print call stack */
                }
        }
        return PMPI_Recv(buf, count, datatype, source, tag, comm, status);
}
Part 3 — Termination: In the wrapper for MPI_FINALIZE, the MPI tool information inter-
face is finalized.
```

```
int MPI_Finalize(void)
{
    int err;
    err=PMPI_T_pvar_handle_free(session, &handle);
    err=PMPI_T_pvar_session_free(&session);
    err=PMPI_T_finalize();
    return PMPI_Finalize();
}
```

14.3.8 Events

During the execution of an MPI application, the MPI implementation can raise *events* of a specific type to inform the user of a state change in the implementation. *Event types* describe specific state changes within the MPI implementation. In comparison to aggregate performance variables, events provide per-instance information on such state changes. The MPI implementation is said to *raise an event* when it invokes a callback function previously registered for the corresponding event type by the user. Each callback invocation for a specific event instance has a timestamp associated with it, which can be queried by the user, describing the time when the event was observed by the implementation. This decouples the observation of the state change from the communication of this information to the user. A timestamp in this context is a count of clock ticks elapsed since some time in the past and represented as a variable of type MPI_Count.

Event Sources

As a means to manage multiple state changes to be observed concurrently by different parts of the software and hardware system, the event interface of the MPI Tool Information

Unofficial Draft for Comment Only

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

37

38

39

40

41

42

43 44

 $45 \\ 46$

47

1		-	source in this context is a concept describing the
2			rce may or may not directly represent a concrete
3			. This concept is used primarily to describe partial
4	ordering of	f events across different comp	onents where total ordering cannot necessarily be
5	determined	l or is too costly to enforce.	
6	The fo	blowing function can be used t	to query the number of event sources, $num_sources:$
7			
8			
9	MPI_1_SC	OURCE_GET_NUM(num_sourc	ies)
10 11	OUT	num_sources	returns number of event sources (integer)
12	C binding	7	
13		S '_source_get_num(int *num	_sources)
14	TL		
15 16			rces can be queried with a call to nplementation is allowed to increase the number of
17			process. However, MPI implementations are not
		-	source or to delete an event source once it has been
18		_	
19			nt sources become available via dynamic loading of
20	additional	components in the MPI imple	ementation).
21			
22	MPI_T_SC	URCE_GET_INFO(source_ind	ex, name, name_len, desc, desc_len, ordering,
23		ticks_per_second, max_ti	
24	IN	source_index	
25 26		source_index	index of the source to be queried between 0 and num_sources -1 (integer)
27	OUT	name	buffer to return the string containing the name of the
28			source (string)
29 30	INOUT	name_len	length of the string and/or buffer for name (integer)
31	OUT	desc	buffer to return the string containing the description
32			of the source (string)
33 34	INOUT	desc_len	length of the string and/or buffer for $desc$ (integer)
35	OUT	ordering	flag indicating chronological ordering guarantees
36			given by the source (integer)
37	OUT	ticks_per_second	the number of ticks per second for the timer of this
38	001	tiend_per_second	source (integer)
39	OUT	max_ticks	the maximum count of ticks reported by this source
40			before overflow occurs (integer)
41	OUT	info	optional info object (handle)
42			
43 44	C binding	g	
45			rce_index, char *name, int *name_len,
45	· ··· - - -	-	c_len, MPI_T_source_order *ordering,
			_second, MPI_Count *max_ticks,
47 48		MPI_Info *info)	,,,,,,,,
-10		······································	

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 14.3.3.

The arguments desc and desc_len are used to return the description of the source as described in Section 14.3.3.

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 MPI is not yet initialized, already finalized, or the argument to info provided by the user is the NULL pointer, this argument is ignored. If an info object is returned, 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_SOURCE_GET_TIMESTAMP(source_index, timestamp)			
IN so	ource_index	index of the source	31
	urce_index	index of the bource	32
OUT tii	mestamp	Current timestamp from specified source	33
			34

C binding

int MPI_T_source_get_timestamp(int source_index, MPI_Count *timestamp)

To enable proper query of a reference timestamp for a specific source, a user can obtain a current timestamp using MPI_T_SOURCE_GET_TIMESTAMP. The argument source_index identifies the index of the source to query. The call returns MPI_SUCCESS and a current timestamp in the argument timestamp if the source supports ad-hoc generation of timestamps. The call returns MPI_T_ERR_INVALID_INDEX if the index does not identify a valid source. The call returns MPI_T_ERR_NOT_SUPPORTED if the source does not support the ad-hoc generation of timestamps.

Callback Safety Requirements

The actions a user is allowed to perform inside a callback function may vary with its execution context. As the user has no control over the execution context of specific callback

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

35

36 37

38

39

40

41

42

43

44 45

 $46 \\ 47$

function invocations, MPI provides a way to communicate this information using callback
 safety levels.

3	
4	Safety Requirement
5	MPI_T_CB_REQUIRE_NONE
6	MPI_T_CB_REQUIRE_MPI_RESTRICTED
7	MPI_T_CB_REQUIRE_THREAD_SAFE
8	MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE
9	
10	
11	Table 14.5: Hierarchy of safety requirement levels for event callback routines.
12	Table 14.5 provides the hierarchy of callback safety requirements levels within user-
13	defined callback functions. The MPI implementation provides the safety requirement as an
14	argument to the callback when it is invoked.
15	The level of MPI_T_CB_REQUIRE_NONE is the lowest level and does not impose any
16	restrictions on the callback function.
17	The level of MPI_T_CB_REQUIRE_MPI_RESTRICTED restricts the set of MPI functions
18	
19	that can be called from inside the callback to all functions with the prefix MPI_T as well as
20	MPI_WTICK and MPI_WTIME.
21	Advice to users. While some MPI functions are safe to be called inside a callback
22	function used in the MPI tool information interface—which may in some implemen-
23	tations be issued from asynchronous contexts such as signal handlers—this does not
24	imply that those MPI functions are generally safe to be called in asynchronous contexts
25	such as signal handlers. (<i>End of advice to users.</i>)
26	such as signal handlers. (End of duotee to users.)
27	The level of MPI_T_CB_REQUIRE_THREAD_SAFE includes all the limitations of
28	MPI_T_CB_REQUIRE_MPI_RESTRICTED and additionally requires the callback to be reen-
29	trant and thread-safe. This means the callback must allow its execution to be interrupted
30	by or happen concurrently with any other callback including itself.
31	The level of MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE includes all the limitations of
32	MPI_T_CB_REQUIRE_THREAD_SAFE and additionally requires the callback to meet the safety
33	requirements needed to support invocations from asynchronous contexts, such as signal
34	handlers.
35	
36	Advice to users. It is always safe to assume the highest restrictions for a callback
37	invocation (i.e., MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE). By evaluating the spe-
38	cific requirements at runtime, a tool may obtain more freedom of action within the
39	callback. (End of advice to users.)
40	
41	Advice to implementors. A high-quality implementation will strive to set callback
42	safety requirements to the most permissive level for a given callback invocation. (End
43	of advice to implementors.)
44	- /
45	All functions with the prefix MPI_T, except those listed in Table 14.6, may return
46	the error code MPI_T_ERR_NOT_ACCESSIBLE to indicate that the user may not access this
47	function at this time.
48	

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
		12

Table 14.6: List of MPI functions that when called from within a callback function may not return MPI_T_ERR_NOT_ACCESSIBLE.

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*:

MPI_T_EVENT_GET_NUM(num_events)			44
OUT	num_events	returns number of event types (integer)	45
			46
C binding			47
	5		10

13

141516

17

18

19

2021

22

23

2425

26

27

28

29 30

3132

33

34

35

36

37

38

39

40

41

4243

	678		CHAPTER 14. TOOL SUPPORT			
1	int MPI_T	int MPI_T_event_get_num(int *num_events)				
2 3 4 5	The function MPI_T_EVENT_GET_INFO provides access to additional information about a specific event type.					
6 7 8	MPI_T_EV	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)				
9 10 11	IN	event_index	index of the event type to be queried between 0 and $num_events - 1$ (integer)			
12 13	OUT	name	buffer to return the string containing the name of the event type (string)			
14	INOUT	name_len	length of the string and/or buffer for name (integer)			
15 16	OUT	verbosity	verbosity level of this event type (integer)			
17 18	OUT	array_of_datatypes	array of MPI basic datatypes used to encode the event data (array of handles)			
19 20	OUT	array_of_displacements	array of byte displacements of the elements in the event buffer (array of non-negative integers)			
21 22 23	INOUT	num_elements	<pre>length of array_of_datatypes and array_of_displacements arrays (integer)</pre>			
24 25	OUT	enumtype	optional descriptor for enumeration information (handle)			
26	OUT	info	optional info object (handle)			
27 28 29	OUT	desc	buffer to return the string containing a description of the event type (string)			
30	INOUT	desc_len	length of the string and/or buffer for desc (integer)			
31 32 33 34	OUT	bind	type of MPI object to which an event of this type must be bound (integer)			
35	C binding					
36	int MPI_I	<pre>int MPI_T_event_get_info(int event_index, char *name, int *name_len,</pre>				
37		MPI_Aint *array_of_displacements, int *num_elements,				
38 39		MPI_T_enum *enumtype, MPI_Info* info, char *desc,				
40		int *desc_len, int *bind)				
41	After	After a successful call to MPI_T_EVENT_GET_INFO for a particular event type, sub-				
42	-	sequent calls to this routine that query information about the same event type must return				
43	the same information. If any INOUT or OUT argument to MPI_T_EVENT_GET_INFO is a					
44 45	-	NULL pointer, the implementation will ignore the argument and not return a value for the specific argument.				
46		The arguments name and name_len are used to return the name of the event type as				
47 48	described in Section 14.3.3. 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 14.3.1).

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.

MPI can optionally return an info object containing the default hints set for a registration handle for this event type. If MPI is not yet initialized, already finalized, or the argument to info provided by the user is the NULL pointer, this argument is ignored. If an info object is returned, an MPI implementation is required to return all hints that are sup-2728ported 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 implementation; and 2930 any additional hints that were set by the implementation. If no such hints exist, a handle 31 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 14.3.3. 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 14.3.2).

If an event type has an equivalent name across connected MPI processes, the following OUT parameters must be identical: verbosity, array_of_datatypes, num_elements, enumtype, and bind. The returned description must be equivalent. As the argument array_of_displacements is process dependent, it may differ across connected MPI processes.

This routine returns MPI_SUCCESS on success and returns MPI_T_ERR_INVALID_INDEX if event_index does not match a valid event type index provided by the implementation at the time of the call.

454647

48

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6 $\overline{7}$

8 9

10

1112

13

14

15

16

17

18

1920

21

22

23 24

25

26

32 33

34

35

36 37

38 39

40 41

42

1 MPI_T_EVENT_GET_INDEX(name, event_index) 2 IN name of the event type (string) name 3 OUT event_index index of the event type (integer) 4 56 C binding $\overline{7}$ int MPI_T_event_get_index(const char *name, int *event_index) 8 MPI_T_EVENT_GET_INDEX returns the index of an event type identified by a known 9 event type name. The name parameter is provided by the caller, and event_index is re-10 turned by the MPI implementation. The name parameter is a string terminated with a null 11 character. 12This routine returns MPI_SUCCESS on success and returns MPI_T_ERR_INVALID_NAME 13 if **name** does not match the name of any event type provided by the implementation at the 14time of the call. 1516This routine is provided to enable fast retrieval of an event index by Rationale. 17 a tool, assuming it knows the name of the event type for which it is looking. The 18 number of event types exposed by the implementation can change over time, so it 19 is not possible for the tool to simply iterate over the list of event types once at 20initialization. Although using MPI implementation specific event type names is not 21portable across MPI implementations, tool developers may choose to take this route 22 for lower overhead at runtime because the tool will not have to iterate over the entire 23set of event types to find a specific one. (End of rationale.) 2425Handle Allocation and Deallocation 26Before the MPI implementation calls a callback function on the occurrence of a specific 2728event, 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. 2930 31 MPI_T_EVENT_HANDLE_ALLOC(event_index, obj_handle, info, 32 event_registration) 33 34 IN event_index index of the event type to be queried between 0 and 35 $num_events - 1$ (integer) 36 IN obj_handle pointer to a handle of the MPI object to which this 37 event is supposed to be bound (pointer) 38 IN info info argument (handle) 39 40 OUT event_registration event registration (handle) 41 42C binding 43 int MPI_T_event_handle_alloc(int event_index, void *obj_handle, 44 MPI_Info info, MPI_T_event_registration *event_registration) 45MPI_T_EVENT_HANDLE_ALLOC creates a registration handle for the event type iden-46 tified by event_index. Furthermore, if required by the event type, the registration handle 47is bound to the object referred to by the argument obj_handle. The argument obj_handle 48

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 argument (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 an 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.*)

MPI_T_EVENT_HANDLE_GET_INFO(event_registration, info_used)							
IN event_registration	event registration (handle)						
OUT info_used	info argument (handle)						
C binding							
<pre>int MPI_T_event_handle_get_info(</pre>							

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.

 $\mathbf{2}$

12	MPI_T_EV	MPI_T_EVENT_REGISTER_CALLBACK(event_registration, cb_safety, info, user_data, event_cb_function)				
$\frac{3}{4}$	IN	event_registration	event registration (handle)			
4 5	IN	cb_safety	Maximum callback safety level (integer)			
6	IN	info	info argument (handle)			
7	IN	user_data	pointer to a user-controlled buffer (pointer)			
8 9	IN	event_cb_function	pointer to user-defined callback function (pointer)			
10			pointer to aber defined consider function (pointer)			
11	C binding	Ş				
12 13 14 15	int MPI_T	<pre>int MPI_T_event_register_callback(</pre>				
16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34	MPI_T_EVENT_REGISTER_CALLBACK associates a user-defined function pointed to by event_cb_function with an allocated event-registration handle. The maximum callback safety level supported by the callback function is passed in the argument cb_safety. The safety levels are defined in Table 14.5. A user can register multiple callback functions for a given event-registration handle, potentially specifying one for each callback safety level. Registering a callback function for a specific callback safety level overwrites any previously registered callback function pointer and info object associated with the event registration for the specific callback safety level. If event_cb_function is the NULL pointer, the association of a callback function for that callback safety level is removed. When an event is triggered, the implementation will select from all registered callbacks is invoked. In situations where the required callback safety level exceeds the highest level for which a callback function is registered for a given registration handle, the event instance is dropped. At invocation time, the implementation passes the pointer to a user-defined memory region specified during callback registration with the argument user_data. The user can pass hints for the registration of the specified callback function to the MPI implementation via the info argument.					
35 36 37 38 39	is ass callba lower	Advice to users. As event instances can be raised as soon as the registration handle is associated with the first callback function, the callback function with the highest callback safety guarantees should be registered before any further registrations for lower callback safety guarantees, to avoid dropped events due to insufficient callback safety guarantees. (<i>End of advice to users.</i>)				
40 41		-	PI_T_EVENT_REGISTER_CALLBACK in the argu-			
42	ment event	_cb_function needs to have th	e ionowing type:			
43	typedef v	oid (*MPI_T_event_cb_fund	ction)(
44 45			nt_instance event_instance,			
45 46			nt_registration event_registration,			
47		<pre>MPI_T_cb_safety cb_safety, void *user_data);</pre>				
48		TOTA AUSEL	_uuuu, ,			

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 FV	ENT CALLBACK SET INFO(event_registration, cb_safety, info)
IN	event_registration	event registration (handle)
IN	cb_safety	
	2	Callback safety level (integer)
IN	info	info argument (handle)
C binding	_event_callback_set_info(tion event_registration,
istered for associated effect on pr on previous	the callback safety level spec with event_registration using eviously set or defaulted hints sly set or defaulted hints	NFO updates the hints of the callback function reg- ified by cb_safety of the event-registration handle the hints provided in info. This operation has no a that are not specified by info. It also has no effect are specified by info, but are ignored by the MPI /ENT_CALLBACK_SET_INFO.
MPI_T_EV	ENT_CALLBACK_GET_INFO	(event_registration, cb_safety, info_used)
IN	event_registration	event registration (handle)
IN	cb_safety	Callback safety level (integer)
Ουτ	info_used	info argument (handle)
C binding	_event_callback_get_info(

MPI_T_event_registration event_registration, MPI_T_cb_safety cb_safety, MPI_Info *info_used)

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 setting 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

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\overline{7}$

1 2 3	-	at contains no key/value pai	exist, a handle to a newly created info object is rs. The user is responsible for freeing info_used via
4 5 6	To stop		om raising events for a specific registration, a user gistration handle.
7 8	MPI_T_EVE	ENT_HANDLE_FREE(event_u	registration, user_data, free_cb_function)
9	IN	event_registration	event registration (handle)
10 11	IN	user_data	pointer to a user-controlled buffer (pointer)
12	IN	free_cb_function	pointer to user-defined callback function (pointer)
13 14	C binding		
15 16	int MPI_T_		_event_registration event_registration, function free_cb_function)
17 18 19 20 21 22 23 24 25 26	was initiate event_regist of the call. when it is a registration user function pointer to u	ed successfully and returns ration does not match a val The callback function free_ able to guarantee that no fu handle will be raised. If the on is invoked after successful	The second secon
27 28 29 30 31 32 33	alway safety <i>users.</i> The ca	s be prepared to postpone a requirements exceed those r) llback function passed to MI	unction associated with a registration handle should any pending actions, should the provided callback required by the pending actions. (<i>End of advice to</i> PI_T_EVENT_HANDLE_FREE in the argument
34	free_cb_fune	ction needs to have the follow	ning type:
35 36 37 38 39	typedef vo		nt_registration event_registration, safety cb_safety,
40	Handling Dr	opped Events	
41 42 43 44 45 46 47	correspondi events and c callback fur callback fur	ng to a matching event han delay the callback invocation. action cannot be called and action meeting the required ca	PI implementation cannot invoke the user function dle. An implementation is allowed to buffer such If an event occurs at times when the corresponding the corresponding data cannot be buffered, or no allback safety level is registered, the event data may the user can set a handler function for a specific

MPI_T_	EVENT_SET_DROPPED_HA	NDLER(event_registration, dropped_cb_function)	1
IN	event_registration	valid event registration (handle)	2
IN	dropped_cb_function	pointer to user-defined callback function (pointer)	3 4
			5
C bind	•		6
int MP	[_T_event_set_dropped_hand		7
	0	ration event_registration, d_cb_function dropped_cb_function)	8 9
	MF1_1_event_droppe	a_cb_runction aroppea_cb_runction)	10
		_HANDLER registers the function	11
		the MPI implementation when event information is	12
		pecified in event_registration. Subsequent calls to NDLER with the same registration handle will replace	13
		ns for that registration handle. If the pointer to	14
-		tter, no data loss is recorded or reported until a new	15
	llback function is registered.		16
			17 18
		on of the dropped handler callback function may not	18
oc	cur close to the time the even	t was actually lost. (End of advice to users.)	20
The	e callback function passed to	MPI_T_EVENT_SET_DROPPED_HANDLER in the	21
	nt dropped_cb_function needs		22
			23
typedef		ped_cb_function)(int count,	24
		vent_registration event_registration,	25
		rce_index, b_safety_cb_safety,	26
		ser_data);	27 28
			20
	3	corresponds to the event registration handle to which	30
		argument count provides a best effort estimation of	31
		red event callback corresponding to event_registration	32
		istration of the dropped-callback handler or the last	33
	<u> </u>	Ilback handler. The source_index provides the index sponding event information. The argument cb_safety	34
		callback function must fulfill in the current invocation.	35
		described in Table 14.5. The argument user_data is	36 27
-		that was passed to the MPI implementation during	37 38
callback	registration.		39

Advice to users. A callback function for dropped events associated with a registration handle should always be prepared to postpone any pending actions, should the provided callback safety requirements exceed those required by the pending actions. (*End of advice to users.*)

Advice to implementors. A high-quality implementation will strive to invoke a callback function for dropped events associated with a registration handle at times that provide as much freedom of action to the function as possible. (*End of advice to implementors.*)

Unofficial Draft for Comment Only

39 40

 41

42

43

44

45 46

47

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 14.5. 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 and 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	t_instance	event-instance handle provided to the callback function (handle)
IN eleme	ent_index	index into the array of datatypes of the item to be queried (integer)
OUT buffe	r	pointer to a memory location to store the item data (pointer)
C binding int MPI_T_even	t_read(MPI_T_event_i	nstance event_instance,

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.

4
MPI_T_EVENT_COPY(event_instance, buffer) 5
IN event_instance event instance provided to the callback function ⁶ (handle) ⁷
OUTbufferuser-allocated buffer for event data (pointer)89
10
C binding
<pre>int MPI_T_event_copy(MPI_T_event_instance event_instance, void *buffer) 12</pre>
MPI_T_EVENT_COPY copies the event 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 ¹⁴
type, which can be computed from the type and displacement information returned by ¹⁵
the corresponding call to MPI_T_EVENT_GET_INFO. The data may include padding bytes
between individual elements of the event data in the buffer. A user can reconstruct the
location and size of the data contained in the buffer through the information returned by
MPI_T_EVENT_GET_INFO.
Advice to implementary. An implementation should strive to use an appropriately 2^{1}
nucce to implementations. An implementation should shrive to use an appropriately
Compact representation when copying event instance data to a user buffer via MPI_T_EVENT_COPY to reduce the amount of memory required for the user buffer. ²²
(End of advice to implementors.)
Reading Event Meta Data
27
Additional to the specific event data encoded by each event type, supplemental information 28
available across all event types can be queried. 29
30
MPI_T_EVENT_GET_TIMESTAMP(event_instance, event_timestamp) 31
32
OUTevent_timestamptimestamp the event was observed (integer)3536
C binding
<pre>int MPI_T_event_get_timestamp(MPI_T_event_instance event_instance, MPT_Count_*event_timestamp)</pre>

MPI_Count *event_timestamp)

MPI_T_EVENT_GET_TIMESTAMP returns the timestamp of when the event was initially observed by the implementation. The event_instance argument identifies the event instance to query. It is erroneous to provide any other handle to the call than the one passed by the MPI implementation to the callback function in which the timestamp is read.

Advice to users. An MPI implementation may postpone the call to the user's callback function. In this case, the call to MPI_T_EVENT_GET_TIMESTAMP may yield a timestamp in the past that is closer to the time the event was initially observed, as 48

Unofficial Draft for Comment Only

1 2

40

 41

42

43

 $44 \\ 45$

1 2		osed to a timestamp capt sers.)	ured during callback function invocation. ($End \ of \ advice$
3 4 5	close	e as possible to the earlie	high-quality implementation will return a timestamp as st time the event was observed by the MPI implementa-
6 7	tion.	(End of advice to imple	mentors.)
8		-	different components acting as event sources in the MPI ontext is an abstract concept that helps to define partial
9 10	ordering o	f raised events, as each	source provides its own ordering guarantees. A source
11 12 13	To ide	entify the source of an ev	event, rather than the origin of the data. rent instance, the user can query the index of the source lback function invocation.
14 15 16	Advi	ice to implementors. An	excessive number of event sources may negatively impact per-source overhead in event handling. (<i>End of advice</i>
17	to in	nplementors.)	
18 19			
20	MPI_T_E\	/ENT_GET_SOURCE(eve	ent_instance, source_index)
21 22 23	IN	event_instance	event instance provided to the callback function (handle)
24	OUT	source_index	index identifying the source
25 26	C bindin	σ	
27 28		-	I_T_event_instance event_instance, ex)
29 30 31 32	to provide	e any other event-instan	identifies the event instance to query. It is erroneous ce handle to the call than the one passed by the MPI action in which the source is queried.
33 34			turns the index of the source of the event instance. It ation on the source using MPI_T_SOURCE_GET_INFO.
35 36 37 38	chro	nological processing of ev	nction invocations are associated with a source to enable rents on the tool side, when required, while retaining low API implementation. (<i>End of rationale.</i>)
39 40	14.3.9 V	ariable Categorization	
41	MPI imple	mentations can optionall	y group performance and control variables into categories
42	-		ween various variables. For example, an MPI implemen-
43 44			performance variables that refer to message transfers in aby distinguish them from variables that refer to local
45		-	eby distinguish them from variables that refer to local ons or other interactions with the operating system.
46		-	her categories to form a hierarchical grouping. Categories
47	-		r directly or transitively within other included categories.
48		,	his allows MPI to refine the grouping of variables referring

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_cat	current number of categories (integer)	
-----	---------	--	--

C binding

int MPI_T_category_get_num(int *num_cat)

Individual category information can then be queried by calling the following function:

	num_pvars, num_o	cat_index, name, name_len, desc, desc_len, num_cvars, categories)
IN	cat_index	index of the category to be queried (integer)
OUT	name	buffer to return the string containing the name of the category (string)
INOUT	name_len	length of the string and/or buffer for name (integer)
OUT	desc	buffer to return the string containing the description of the category (string)
INOUT	desc_len	length of the string and/or buffer for $desc\xspace$ (integer)
OUT	num_cvars	number of control variables in the category (integer)
OUT	num_pvars	number of performance variables in the category (integer)
OUT	num_categories	number of categories contained in the category (integer)
The a lescribed i The r inique wit If any mplement The a lescribed i Retur lescription be set to o The fit categories hum_categ If the	char *desc, int int *num_catego rguments name and na in Section 14.3.3. outine is required to re- h respect to all other n outine of the parameter to N ation will ignore the pa- rguments desc and desc in Section 14.3.3. ning a description is op n, the first character for one at the return of this inction returns the num contained in the querie ories, respectively.	turn a name of at least length one. This name must be ames for categories used by the MPI implementation. IPI_T_CATEGORY_GET_INFO is the NULL pointer, the rameter and not return a value for the parameter. E_len are used to return the description of the category as tional. If an MPI implementation decides not to return a desc must be set to the null character and desc_len must call. ber of control variables, performance variables and other ed category in the arguments num_cvars, num_pvars, and equivalent across connected MPI processes, then the re-
	ATEGORY GET NUM	EVENTS(cat_index, num_events)
	cat_index	index of the category to be queried (integer)
IN		

MPI_T_CATEGORY_GET_NUM_EVENTS returns the number of event types contained in the queried category.

MPI_T_	CATEGORY_G	ET_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)

 			33
IN	cat_index	index of the category to be queried, in the range 0	34
		and $num_cat - 1$ (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 0
3
                                                and 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 0
       IN
                  cat_index
18
                                                and 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 0
33
                                                and \mathsf{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 and MPI_T_CATEGORY_GET_CATEGORIES can be used as input to MPI_T_CVAR_GET_INFO, MPI_T_PVAR_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 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.

14.3.10 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 14.7) 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 8.4 and follow the same rules and restrictions. In particular, they must satisfy:

 $0 = MPI_SUCCESS < MPI_T_ERR_XXX \le MPI_ERR_LASTCODE.$

Rationale. All MPI tool information interface functions must return error classes, because applications cannot portably call MPI_ERROR_CLASS before MPI initialization to map an arbitrary error code to an error class. (*End of rationale.*)

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

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

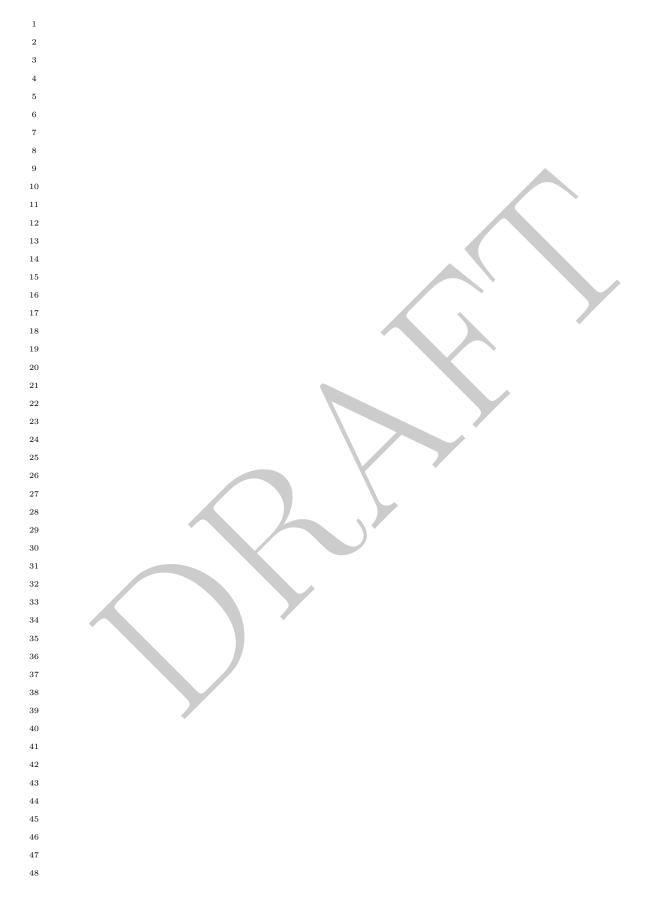
All requirements for the profiling interfaces, as described in Section 14.2, also apply to the MPI tool information interface. All rules, guidelines, and recommendations from Sec-tion 14.2 apply equally to calls defined as part of the MPI tool information interface. $\mathbf{5}$ $\overline{7}$

14.3.11 Profiling Interface

 $\mathbf{2}$

Return Code	Description	
Return Codes for All Functions in t	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	ns: MPI_T_ENUM_*	
MPI_T_ERR_INVALID_INDEX	The enumeration index is invalid	
MPI_T_ERR_INVALID_ITEM	The item index queried is out of range	
	(for MPI_T_ENUM_GET_ITEM only)	
Return Codes for Variable, Categor	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 Session Functions:		
MPI_T_ERR_OUT_OF_SESSIONS	No more sessions available	
MPI_T_ERR_INVALID_SESSION	Session argument is not a valid session	
Return Codes for Control Variable Access Functions: MPI_T_CVAR_{READ WRITE}		
MPI_T_ERR_CVAR_SET_NOT_NOW	Variable cannot be set at this moment	
MPI_T_ERR_CVAR_SET_NEVER	Variable cannot be set until end of execution	
MPI_T_ERR_INVALID_HANDLE	The handle is invalid	
Return Codes for Performance Variable Access and Control:		
MPI_T_PVAR_{START STOP READ	WRITE RESET READREST}	
MPI_T_ERR_INVALID_HANDLE	The handle is invalid	
MPI_T_ERR_INVALID_SESSION	Session argument is not a valid session	
MPI_T_ERR_PVAR_NO_STARTSTOP	Variable cannot be started or stopped (for	
	MPI_T_PVAR_START and MPI_T_PVAR_STOP)	
MPI_T_ERR_PVAR_NO_WRITE	Variable cannot be written or reset (for	
	MPI_T_PVAR_WRITE and MPI_T_PVAR_RESET)	
MPI_T_ERR_PVAR_NO_ATOMIC	Variable cannot be read and written atomically (for	
	MPI_T_PVAR_READRESET)	
Return Codes for Source Functions:		
MPI_T_ERR_INVALID_INDEX	The source index is invalid	
MPI_T_ERR_NOT_SUPPORTED	Requested functionality not supported	
Return Codes for Category Function	ns: MPI_T_CATEGORY_*	
MPI_T_ERR_INVALID_INDEX	The category index is invalid	

Table 14.7: Return codes used in functions of the MPI tool information interface



Deprecated Interfaces

15.1 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 18.2.7. 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	<u> </u>		30
int MPI		Copy_function *copy_fn,	31
		<pre>nction *delete_fn, int *keyval,</pre>	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	Ild 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	ontron 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

CHAPTER 15. DEPRECATED INTERFACES

1 2 3 4 5 6 7 8 9 10 11	<pre>SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,</pre>
12 13 14 15 16 17	<pre>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,</pre>
18 19 20	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 31	<pre>delete_fn may be specified as MPI_NULL_DELETE_FN from either C or Fortran; MPI_NULL_DELETE_FN is a function that does nothing, other than returning MPI_SUCCESS. Note that MPI_NULL_DELETE_FN is also deprecated. The following function is deprecated and is superseded by MPI_COMM_FREE_KEYVAL in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified. MPI_KEYVAL_FREE(keyval)</pre>
32 33	INOUT keyval Frees the integer key value (integer)
34 35 36	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_ATTF	R_PUT(comm, keyval, attribute	e_val)	1
INOUT	comm	communicator to which attribute will be attached	2
	comm	(handle)	3
IN	keyval	key value, as returned by MPI_KEYVAL_CREATE	4
		(integer)	5 6
IN	attribute_val	attribute value	7
	attribute_var		8
C binding	7		9
•		nt keyval, void *attribute_val)	10
	•		11
For this ro	utine, an interface within the	mpi_f08 module was never defined.	12
Fortran b	binding		13
MPI_ATTR_	PUT(COMM, KEYVAL, ATTRIBU	JTE_VAL, IERROR)	14 15
INTEG	ER COMM, KEYVAL, ATTRIBUT	CE_VAL, IERROR	15
The fo	ollowing function is deprecated	and is superseded by MPI_COMM_GET_ATTR in	17
		nition of the deprecated function is the same as of	18
the new fu	nction, except of the function	name. The language bindings are modified.	19
			20
ΜΡΙ ΔΤΤΕ	R_GET(comm, keyval, attribute	val flag)	21
			22
IN	comm	communicator to which attribute is attached (handle)	23
IN	keyval	key value (integer)	24 25
OUT	attribute_val	attribute value, unless $flag = false$	25 26
OUT	flag	true if an attribute value was extracted; false if no	27

attribute is associated with the key

```
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.

MPI_ATTR_DELETE(comm, keyval)

			44
INOUT	comm	communicator to which attribute is attached (handle)	45
IN	keyval	The key value of the deleted attribute (integer)	46
			47

```
C binding
```

2829

30

 31 32

33

34

35

36

37 38

39

40

414243

1 int MPI_Attr_delete(MPI_Comm comm, int keyval) $\mathbf{2}$ For this routine, an interface within the mpi_f08 module was never defined. 3 4 Fortran binding 5MPI_ATTR_DELETE(COMM, KEYVAL, IERROR) 6 INTEGER COMM, KEYVAL, IERROR 7 8 9 15.2Deprecated since MPI-2.2 10 11The entire set of C++ language bindings have been removed. See Chapter 16, Removed 12Interfaces for more information. The following function typedefs have been deprecated and are superseded by new 13 14names. Other than the typedef names, the function signatures are exactly the same; the 15names were updated to match conventions of other function typedef names. 16Deprecated Name New Name 17 MPI_Comm_errhandler_fn MPI_Comm_errhandler_function 18 MPI_File_errhandler_function MPI_File_errhandler_fn 19 MPI_Win_errhandler_fn MPI_Win_errhandler_function 20212215.3 Deprecated since MPI-3.2 23Cancelling a send request by calling MPI_CANCEL has been deprecated and may be removed 24 in a future version of the MPI specification. 252627Deprecated since MPI-4.0 15.4 2829The following function is deprecated and is superseded by the new MPI_INFO_GET_STRING 30 call in MPI-4.0. 31 32 MPI_INFO_GET(info, key, valuelen, value, flag) 33 34 IN info info object (handle) 35 IN key key (string) 36 37 IN valuelen length of value arg (integer) 38 OUT value value (string) 39 OUT flag true if key defined, false if not (boolean) 40 41 C binding 42int MPI_Info_get(MPI_Info info, const char *key, int valuelen, char *value, 43 int *flag) 44 45Fortran 2008 binding 46MPI_Info_get(info, key, valuelen, value, flag, ierror) 47TYPE(MPI_Info), INTENT(IN) :: info 48

	ACTER(LEN=*), INTENT(IN)	•	1
	GER, INTENT(IN) :: valuel		2 3
	ACTER(LEN=valuelen), INTE CAL, INTENT(OUT) :: flag	NI(UUI) :: Value	4
	GER, OPTIONAL, INTENT(OUT) :: ierror	5
Fortran	hinding		6
	_GET(INFO, KEY, VALUELEN,	VALUE, FLAG, IERROR)	7 8
	GER INFO, VALUELEN, IERRO	R	9
	ACTER*(*) KEY, VALUE		10
	CAL FLAG		11
	· ·	ed and is superseded by the new	12 13
MPI_INFC	GET_STRING call in MPI-4.	υ.	14
			15
MPI_INFC	GET_VALUELEN(info, key, v	aluelen, flag)	16
IN	info	info object (handle)	17 18
IN	key	key (string)	19
OUT	valuelen	length of value arg (integer)	20
OUT	flag	true if key defined, false if not (boolean)	21
			22 23
C bindin	0		23 24
int MPI_	<pre>Info_get_valuelen(MP1_Inf</pre>	o info, const char *key, int *valuelen,	25
			26
	2008 binding _get_valuelen(info, key,	valuelen flag jerror)	27 28
	(MPI_Info), INTENT(IN) ::	-	28
	ACTER(LEN=*), INTENT(IN)		30
	GER, INTENT(OUT) :: value	len	31
	CAL, INTENT(OUT) :: flag) ierrer	32
	GER, OPTIONAL, INTENT(OUT) lellor	$33 \\ 34$
Fortran	<u> </u>		35
	_GET_VALUELEN(INFO, KEY, GER INFO, VALUELEN, IERRO		36
	ACTER*(*) KEY		37
LOGI	CAL FLAG		$\frac{38}{39}$
The f	ollowing return class has been	deprecated and is superseded by a new name.	40
	Deprecated N		41
	MPI_T_ERR_INVALID_I	-	42
			43 44
	_	are deprecated because the Fortran language ctions provide similar functionality. Note that while	45
-	0	ize in bytes, storage_size() provides the size in bits.	46
_	v v		47

```
1
      MPI_SIZEOF(x, size)
\mathbf{2}
        IN
                   Х
                                                  a Fortran variable of numeric intrinsic type (choice)
3
        OUT
                                                  size of machine representation of that type (integer)
                   size
4
\mathbf{5}
      Fortran 2008 binding
6
     MPI_Sizeof(x, size, ierror)
7
           TYPE(*), DIMENSION(..) :: x
8
           INTEGER, INTENT(OUT) :: size
9
10
           INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
      Fortran binding
12
     MPI_SIZEOF(X, SIZE, IERROR)
13
           <type> X
14
           INTEGER SIZE, IERROR
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

Removed Interfaces

16.1 Removed MPI-1 Bindings

16.1.1 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.

16.1.2 Removed MPI-1 Functions

Table 16.1 shows the removed MPI-1 functions and their replacements.

	· · · · · · · · · · · · · · · · · · ·
Removed	MPI-2 Replacement
MPI_ADDRESS	MPI_GET_ADDRESS
MPI_ERRHANDLER_CREATE	MPI_COMM_CREATE_ERRHANDLER
MPI_ERRHANDLER_GET	MPI_COMM_GET_ERRHANDLER
MPI_ERRHANDLER_SET	MPI_COMM_SET_ERRHANDLER
MPI_TYPE_EXTENT	MPI_TYPE_GET_EXTENT
MPI_TYPE_HINDEXED	MPI_TYPE_CREATE_HINDEXED
MPI_TYPE_HVECTOR	MPI_TYPE_CREATE_HVECTOR
MPI_TYPE_LB	MPI_TYPE_GET_EXTENT
MPI_TYPE_STRUCT	MPI_TYPE_CREATE_STRUCT
MPI_TYPE_UB	MPI_TYPE_GET_EXTENT

Table 16.1: Removed MPI-1 functions and their replacements

16.1.3 Removed MPI-1 Datatypes

Table 16.2 shows the removed MPI-1 datatypes and their replacements.

16.1.4 Removed MPI-1 Constants

Table 16.3 shows the removed MPI-1 constants. There are no MPI-2 replacements.

	704 CHAPTER 16. REMOVED INTERFACES
1	Removed MPI-2 Replacement
2	MPI_LB MPI_TYPE_CREATE_RESIZED
3	MPI_UB MPI_TYPE_CREATE_RESIZED
4	
5	Table 16.2: Removed MPI-1 datatypes and their replacements
6	Table 10.2. Temoved wit P1 datatypes and then replacements
7	Removed MPI-1 Constants
8	C type: const int (or unnamed enum)
9	Fortran type: INTEGER
10 11	MPI_COMBINER_HINDEXED_INTEGER
12	MPI_COMBINER_HVECTOR_INTEGER
13	MPI_COMBINER_STRUCT_INTEGER
14	
15	Table 16.3: Removed MPI-1 constants
16	
17	16.1.5 Removed MPI-1 Callback Prototypes
18 19	Table 16.4 shows the removed MPI-1 callback prototypes and their MPI-2 replacements.
20	Removed MPI-2 Replacement
21	MPI_Handler_function MPI_Comm_errhandler_function
22	
23	
24	Table 16.4: Removed MPI-1 callback prototypes and their replacements
25	
26	
27 28	16.2 C++ Bindings
29	
30	The C++ bindings were deprecated as of MPI-2.2. The C++ bindings are removed in
31	MPI-3.0. The namespace is still reserved, however, and bindings may only be provided by
32	an implementation as described in the MPI-2.2 standard.
33	
34	
35	
36	
37	
38	
39	
40	
41	
42	
43 44	
44 45	
45	
40	
48	

Backward Incompatibilities

17.1 Backward Incompatible since MPI-3.2

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.



Language Bindings

18.1 Fortran Support

18.1.1 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 [42] + TS 29113 [43].

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 18.1.7. (*End of rationale.*)

MPI defines three methods of Fortran support:

- 1. USE mpi_f08: This method is described in Section 18.1.2. 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 18.1.3 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 18.1.4. 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 18.1.6 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 [43]; 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 18.1.2 through 18.1.4 define the Fortran support methods. The Fortran in-29terfaces of each MPI routine are shorthands. Section 18.1.5 defines the corresponding 30 full interface specification together with the specific procedure names and implications for 31 the profiling interface. Section 18.1.6 the implementation of the MPI routines for differ-32 ent versions of the Fortran standard. Section 18.1.7 summarizes major requirements for 33 valid MPI-3.0 implementations with Fortran support. Section 18.1.8 and Section 18.1.9 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 18.1.10 through 18.1.19 give an overview and details on known prob-41 lems when using Fortran together with MPI; Section 18.1.20 compares the Fortran problems 42with those in C. 43

43 44

18.1.2 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 18.1.6 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 18.1.3.

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 18.1.6 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 noncontiguous 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 18.1.6 for details.
- Declare each argument with an INTENT of IN, OUT, or INOUT as defined in this standard.

 $\frac{4}{5}$

 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 [42], 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 18.1.3. (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 [42] together with the Technical Specification "TS 29113 Further Interoperability with C" [43] 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 [43], "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.*)

18.1.3 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 18.1.6 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 18.1.12 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 [29], if the compiler supports this TS 29113 language feature. See Section 18.1.6 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 18.1.6 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

23

24

25

26

27 28

29

30

31

32

33 34

35

36

37 38

39

40

41

42

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.)

18.1.4 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

 $\overline{7}$

 $46 \\ 47$

	714	CHAPTER 1	8. LANGUAGE BINDINGS
$\frac{1}{2}$		f a status as an INTEGER variable _STATUS_SIZE.	instead of an INTEGER array
3 4	_	ument positions; e.g., interchang	ing the count and
5	datatype argu – Passing incorr	ect MPI handles; e.g., passing a d	latatype instead of a commu-
6 7	nicator.	/ 0/1 0	01
8	9	m mpif.h to the mpi module sho	
9 10		nting include 'mpif.h' after an plicit statement) as long as the	-
11		e and correctly written application	-
12 13	-	ficult. No compile or runtime pr file was always allowed to provid	
14 15	(End of advice to users	.)	
16 17		-3.0, the mpif.h include file wa	
18	8	compatibility. Internally, mpif.k sentialy the same library implem	-
19 20	can be used. (End of r	· · · · ·	ientation of the MFT fournes
20			
22		ons, Procedure Names, and the	-
23 24	_	ation of each MPI routine specifie	
25		program, and the names and typitutes. The Fortran standard all	
26 27	to be implemented with sever	al methods, e.g., within or outside	e of a module, with or without
28		or without TS 29113. Such imp ad different specific procedure n	
29	several implementation sche	nes together with the rules for t	the specific procedure names
30 31	and its implications for the p implementation details.	orofiling interface are specified with	thin this section, but not the
32	implementation details.		
33 34		n was introduced in MPI-3.0 on Searee Fortran support methods ha	
35 36 37	• Portable implement MPI routines in C	ntation of the wrappers from the	MPI Fortran interfaces to the
38		compatible implementation path	when switching
39		SUPPORTED from .FALSE. to .T	_
40 41		I interface need not be backwar	- /
42 43		that a tools layer can use to exa lure names and interfaces used.	amme the MPI library about
44	• No performance d	rawbacks.	
45	° °	een all three Fortran support met	hods.
$\frac{46}{47}$	• Consistent with F	ram 2008 + TS 29113.	
48			

No.	Specific pro- cedure name	Calling convention
1A	MPI_Isend_f08	Fortran interface and arguments, as in Annex A.3, except that in routines with a choice buffer dummy argument, this dummy argument is implemented with non-standard extensions like !\$PRAGMA IGNORE_TKR , which provides a call-by-reference argument without type, kind, and dimension checking.
1B	MPI_Isend_f08ts	Fortran interface and arguments, as in Annex A.3, but only for routines with one or more choice buffer dummy arguments; these dummy arguments are implemented with TYPE(*), DIMENSION().
2A	MPI_ISEND	Fortran interface and arguments, as in Annex A.4, except that in routines with a choice buffer dummy argument, this dummy argument is implemented with non-standard extensions like !\$PRAGMA IGNORE_TKR , which provides a call-by-reference argument without type, kind, and dimension checking.
2B	MPI_ISEND_FTS	C C

Table 18.1: 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 18.1. Case is not significant in the names.

Note that for the deprecated routines in Section 15.1, which are reported only in Annex A.4, scheme 2A is utilized in the mpi module and mpif.h, and also in the mpi_f08 module.

To set MPI_SUBARRAYS_SUPPORTED to .TRUE. within a Fortran support method, it is required that all nonblocking and split-collective routines with buffer arguments are

Unofficial Draft for Comment Only

implemented according to 1B and 2B, i.e., with MPI_Xxxx_f08ts in the mpi_f08 module,
 and with MPI_XXXX_FTS in the mpi module and the mpif.h include file.

The mpi and mpi_f08 modules and the mpif.h include file will each correspond to exactly one implementation scheme from Table 18.1. However, the MPI library may contain multiple implementation schemes from Table 18.1.

- Advice to implementors. This may be desirable for backwards binary compatibility in the scope of a single MPI implementation, for example. (*End of advice to implementors.*)
- 10 Rationale. After a compiler provides the facilities from TS 29113, i.e., TYPE(*), 11 DIMENSION(...), it is possible to change the bindings within a Fortran support method 12to support subarrays without recompiling the complete application provided that the 13 previous interfaces with their specific procedure names are still included in the li-14brary. Of course, only recompiled routines can benefit from the added facilities. 15There is no binary compatibility conflict because each interface uses its own spe-16cific procedure names and all interfaces use the same constants (except the value of 17 MPI_SUBARRAYS_SUPPORTED and MPI_ASYNC_PROTECTS_NONBLOCKING) and type 18 definitions. After a compiler also ensures that buffer arguments of nonblocking MPI 19 operations can be protected through the ASYNCHRONOUS attribute, and the proce-20dure declarations in the mpi_f08 and mpi module and the mpif.h include file declare 21choice buffers with the ASYNCHRONOUS attribute, then the value of 22
- MPI_ASYNC_PROTECTS_NONBLOCKING can be switched to .TRUE. in the module definition and include file. (*End of rationale.*)
 - Advice to users. Partial recompilation of user applications when upgrading MPI implementations is a highly complex and subtle topic. Users are strongly advised to consult their MPI implementation's documentation to see exactly what is and what is not supported. (*End of advice to users.*)
- 29 30

 31

32

33

34

25

26

27

28

Within the mpi_f08 and mpi modules and mpif.h, for all MPI procedures, a second procedure with the same calling conventions shall be supplied, except that the name is modified by prefixing with the letter "P", e.g., PMPI_Isend. The specific procedure names for these PMPI_Xxxx procedures must be different from the specific procedure names for the MPI_Xxxx procedures and are not specified by this standard.

³⁵ A user-written or middleware profiling routine should provide the same specific Fortran ³⁶ procedure names and calling conventions, and therefore can interpose itself as the MPI ³⁷ library routine. The profiling routine can internally call the matching

PMPI routine with any of its existing bindings, except for routines that have callback routine dummy arguments, choice buffer arguments, or that are attribute caching routines (

MPI_{COMM|WIN|TYPE}_{SET|GET}_ATTR). In this case, the profiling software should
 invoke the corresponding PMPI routine using the same Fortran support method as used in
 the calling application program, because the C, mpi_f08 and mpi callback prototypes are
 different or the meaning of the choice buffer or attribute_val arguments are different.

- ⁴⁵ Advice to users. Although for each support method and MPI routine (e.g.,
- ⁴⁶ MPI_ISEND in mpi_f08), multiple routines may need to be provided to intercept ⁴⁷ the specific procedures in the MPI library (e.g., MPI_Isend_f08 and MPI_Isend_f08ts), ⁴⁸ each profiling routine itself uses only one support method (e.g., mpi_f08) and calls

Unofficial Draft for Comment Only

7

8

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 two longest subroutine declaration statements are

SUBROUTINE PMPI_Dist_graph_create_adjacent(a,b,c,d,e,f,g,h,i,j,k) SUBROUTINE PMPI_Rget_accumulate(a,b,c,d,e,f,g,h,i,j,k,l,m,n)

with 71 and 66 characters. With buffers implemented with TS 29113, the specific procedure names have an additional postfix. The longest of such interface definitions is

INTERFACE PMPI_Rget_accumulate
SUBROUTINE PMPI_Rget_accumulate_fts(a,b,c,d,e,f,g,h,i,j,k,l,m,n)

with 70 characters. In principle, continuation lines would be possible in mpif.h (spaces in columns 73–131, & in column 132, and in column 6 of the continuation line) but this would not be valid if the source line length is extended with a compiler flag to 132 characters. Column 133 is also not available for the continuation character because lines longer than 132 characters are invalid with some compilers by default.

The longest specific procedure names are PMPI_Dist_graph_create_adjacent_f08 and PMPI_File_write_ordered_begin_f08ts both with 35 characters in the mpi_f08 module.

For example, the interface specifications together with the specific procedure names can be implemented with

Unofficial Draft for Comment Only

 31

```
1
           MODULE mpi_f08
2
             TYPE, BIND(C) :: MPI_Comm
3
               INTEGER :: MPI_VAL
             END TYPE MPI_Comm
4
             . . .
5
             INTERFACE MPI_Comm_rank ! (as defined in Chapter 6)
6
               SUBROUTINE MPI_Comm_rank_f08(comm, rank, ierror)
7
                 IMPORT :: MPI_Comm
8
                 TYPE(MPI_Comm),
                                       INTENT(IN) :: comm
9
                                       INTENT(OUT) :: rank
                 INTEGER,
                 INTEGER, OPTIONAL,
                                       INTENT(OUT) :: ierror
10
               END SUBROUTINE
11
             END INTERFACE
12
           END MODULE mpi_f08
13
14
           MODULE mpi
15
             INTERFACE MPI_Comm_rank ! (as defined in Chapter 6)
16
               SUBROUTINE MPI_Comm_rank(comm, rank, ierror)
17
                 INTEGER, INTENT(IN) :: comm
                                                  ! The INTENT may be added although
                 INTEGER, INTENT(OUT) :: rank
                                                  ! it is not defined in the
18
                 INTEGER, INTENT(OUT) :: ierror ! official routine definition.
19
               END SUBROUTINE
20
             END INTERFACE
21
           END MODULE mpi
22
23
           And if interfaces are provided in mpif.h, they might look like this (outside of any
24
           module and in fixed source format):
25
           !23456789012345678901234567890123456789012345678901234567890123456789012
26
                 INTERFACE MPI_Comm_rank ! (as defined in Chapter 6)
27
                  SUBROUTINE MPI_Comm_rank(comm, rank, ierror)
28
                   INTEGER, INTENT(IN) :: comm / The argument names may be
29
                   INTEGER, INTENT(OUT) :: rank
                                                  ! shortened so that the
30
                   INTEGER, INTENT(OUT) :: ierror ! subroutine line fits to the
31
                  END SUBROUTINE
                                                    ! maximum of 72 characters.
32
                 END INTERFACE
33
34
           (End of advice to implementors.)
35
           Advice to users. The following is an example of how a user-written or middleware
36
           profiling routine can be implemented:
37
38
           SUBROUTINE MPI_Isend_f08ts(buf,count,datatype,dest,tag,comm,request,ierror)
39
             USE :: mpi_f08, my_noname => MPI_Isend_f08ts
40
             TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
41
             INTEGER,
                                  INTENT(IN)
                                                    :: count, dest, tag
42
             TYPE(MPI_Datatype), INTENT(IN)
                                                    :: datatype
             TYPE(MPI_Comm),
                                  INTENT(IN)
                                                    :: comm
43
             TYPE(MPI_Request), INTENT(OUT)
                                                    :: request
44
             INTEGER, OPTIONAL,
                                  INTENT(OUT)
                                                    :: ierror
45
               ! ... some code for the begin of profiling
46
             call PMPI_Isend (buf, count, datatype, dest, tag, comm, request, ierror)
47
               ! ... some code for the end of profiling
48
           END SUBROUTINE MPI_Isend_f08ts
```

Unofficial Draft for Comment Only

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. 11 ple, an additional MPI_ISEND profiling wrapper can be provided that is called by the 12application and internally calls PMPI_ISEND. There are two typical use cases: a pro-13 filing layer that is developed independently from the application and the MPI library. 14and profiling routines that are part of the application and have access to the appli-15cation data. With MPI-3.0, new Fortran interfaces and implementation schemes were 16 introduced that have several implications on how Fortran MPI routines are internally 17 implemented and optimized. For profiling layers, these schemes imply that several in-18 ternal interfaces with different specific procedure names may need to be intercepted, 19 as shown in the example code above. Therefore, for wrapper routines that are part 20of a Fortran application, it may be more convenient to make the name shift within 21the application, i.e., to substitute the call to the MPI routine (e.g., MPI_ISEND) by a 22call to a user-written profiling wrapper with a new name (e.g., X_MPI_ISEND) and to 23call the Fortran MPI_ISEND from this wrapper, instead of using the PMPI interface. 24 (End of advice to users.) 25

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 18.1 so that the MPI Fortran routines are interceptable as described above. (*End of advice to implementors.*)

18.1.6 MPI for Different Fortran Standard Versions

This section describes which Fortran interface functionality can be provided for different versions of the Fortran standard.

- For Fortran 77 with some extensions:
 - MPI identifiers may be up to 30 characters (31 with the profiling interface).
 - MPI identifiers may contain underscores after the first character.
 - An MPI subroutine with a choice argument may be called with different argument types.
 - Although not required by the MPI standard, the INCLUDE statement should be available for including mpif.h into the user application source code.

Only MPI-1.1, MPI-1.2, and MPI-1.3 can be implemented. The use of absolute addresses from MPI_ADDRESS and MPI_BOTTOM may cause problems if an address does not fit into the memory space provided by an INTEGER. (In MPI-2.0 this problem is solved with MPI_GET_ADDRESS, but not for Fortran 77.)
45
46
47
48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6 7

8 9

10

26

27

28

29

30 31

32 33

34

35

36 37

38

39 40

41

42

43

1	• For Fortran 90:
2	The major additional features that are needed from Fortran 90 are:
$\frac{3}{4}$	- The MODULE and INTERFACE concept.
5	- The KIND= and SELECTEDKIND concept.
6	- Fortran derived TYPEs and the SEQUENCE attribute.
7 8	- The OPTIONAL attribute for dummy arguments.
9 10	 Cray pointers, which are a non-standard compiler extension, are needed for the use of MPI_ALLOC_MEM.
11 12 13 14 15 16	With these features, MPI-1.1 – MPI-2.2 can be implemented without restrictions. MPI-3.0 can be implemented with some restrictions. The Fortran support methods are abbreviated with $S1 = \text{the mpi_f08} \mod S2 = \text{the mpi module}$, and $S3 = \text{the mpif.f}$ include file. If not stated otherwise, restrictions exist for each method which prevent implementing the complete semantics of MPI-3.0.
17 18 19 20	 MPI_SUBARRAYS_SUPPORTED equals .FALSE., i.e., subscript triplets and non- contiguous subarrays cannot be used as buffers in nonblocking routines, RMA, or split-collective I/O.
21 22	- S1, S2, and S3 can be implemented, but for S1, only a preliminary implementation is possible.
23	- In this preliminary interface of $$1$, the following changes are necessary:
24	* TYPE(*), DIMENSION() is substituted by non-standardized extensions
25 26	like !\$PRAGMA IGNORE_TKR.
20	* The ASYNCHRONOUS attribute is omitted.
28	* PROCEDURE() callback declarations are substituted by EXTERNAL .
29	- The specific procedure names are specified in Section 18.1.5.
30 31 32	 Due to the rules specified in Section 18.1.5, choice buffer declarations should be implemented only with non-standardized extensions like !\$PRAGMA IGNORE_TKR (as large as E2008 + TS 20112 is not emilable)
33	(as long as F2008+TS 29113 is not available). In S2 and S3: Without such extensions, routines with choice buffers should be
34	provided with an implicit interface, instead of overloading with a different MPI
35	function for each possible buffer type (as mentioned in Section 18.1.11). Such
36 37	overloading would also imply restrictions for passing Fortran derived types as
38	choice buffer, see also Section $18.1.15$.
39	Only in S1: The implicit interfaces for routines with choice buffer arguments
40	imply that the ierror argument cannot be defined as OPTIONAL. For this reason, it is recommended not to provide the main fOP module if such an extension is not
41	it is recommended not to provide the mpi_f08 module if such an extension is not available.
42	- The ASYNCHRONOUS attribute can not be used in applications to protect buffers
43 44	in nonblocking MPI calls (S1–S3).
45	- The TYPE(C_PTR) binding of the MPI_ALLOC_MEM and MPI_WIN_ALLOCATE
46	routines is not available.
47	
48	

- 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 18.2.5) are also not available in the mpi module and mpif.h.
- For Fortran 95:

The quality of the MPI interface and the restrictions are the same as with Fortran 90.

• For Fortran 2003:

The major features that are needed from Fortran 2003 are:

- Interoperability with C, i.e.,
 - * BIND(C) derived types.
 - * The ISO_C_BINDING intrinsic type C_PTR and routine C_F_POINTER.
- The ability to define an ABSTRACT INTERFACE and to use it for PROCEDURE dummy arguments.
- The ability to overload the operators .EQ. and .NE. to allow the comparison of derived types (used in MPI-3.0 for MPI handles).
- The ASYNCHRONOUS attribute is available to protect Fortran asynchronous I/O. This feature is not yet used by MPI, but it is the basis for the enhancement for MPI communication in the TS 29113.

With these features (but still without the features of TS 29113), MPI-1.1 – MPI-2.2 can be implemented without restrictions, but with one enhancement:

 The user application can use TYPE(C_PTR) together with MPI_ALLOC_MEM as long as MPI_ALLOC_MEM is defined with an implicit interface because a C_PTR and an INTEGER(KIND=MPI_ADDRESS_KIND) argument must both map to a void * argument.

MPI-3.0 can be implemented with the following restrictions:

- MPI_SUBARRAYS_SUPPORTED equals .FALSE..
- For S1, only a preliminary implementation is possible. The following changes are necessary:
 - * TYPE(*), DIMENSION(..) is substituted by non-standardized extensions like !\$PRAGMA IGNORE_TKR.
- The specific procedure names are specified in Section 18.1.5.
- With S1, the ASYNCHRONOUS is required as specified in the second Fortran interfaces. With S2 and S3 the implementation can also add this attribute if explicit interfaces are used.

Unofficial Draft for Comment Only

 24

1 2	 The ASYNCHRONOUS Fortran attribute can be used in applications to try to protect buffers in nonblocking MPI calls, but the protection can work only if the compiler
3	is able to protect asynchronous Fortran I/O and makes no difference between such
4	asynchronous Fortran I/O and MPI communication.
5	- The TYPE(C_PTR) binding of the MPI_ALLOC_MEM, MPI_WIN_ALLOCATE,
6	MPI_WIN_ALLOCATE_SHARED, and MPI_WIN_SHARED_QUERY routines can
7	be used only for Fortran types that are C compatible.
8	- The same restriction as for Fortran 90 applies if non-standardized extensions like
9	!\$PRAGMA IGNORE_TKR are not available.
10 11	• For Fortran $2008 + TS$ 29113 and later and
11	• For Fortran $2003 + TS 29113$ and later and For Fortran $2003 + TS 29113$:
13	The major feature that are needed from TS 29113 are:
14	
15	- TYPE(*), DIMENSION() is available.
16	- The ASYNCHRONOUS attribute is extended to protect also nonblocking MPI com-
17	munication.
18	- The array dummy argument of the ISO_C_BINDING intrinsic C_F_POINTER is not
19	restricted to Fortran types for which a corresponding type in C exists.
20	Using these features, MPI-3.0 can be implemented without any restrictions.
21 22	- With S1, MPI_SUBARRAYS_SUPPORTED equals .TRUE The ASYNCHRONOUS at-
23	tribute can be used to protect buffers in nonblocking MPI calls. The TYPE(C_PTR)
24	binding of the MPI_ALLOC_MEM, MPI_WIN_ALLOCATE,
25	MPI_WIN_ALLOCATE_SHARED, and MPI_WIN_SHARED_QUERY routines can
26	be used for any Fortran type.
27	– With S2 and S3, the value of MPI_SUBARRAYS_SUPPORTED is implementation
28	dependent. A high quality implementation will also provide
29	$MPI_SUBARRAYS_SUPPORTED = = . TRUE. \text{ and will use the } ASYNCHRONOUS \text{ attribute}$
30	in the same way as in S1.
31 32	$-$ If non-standardized extensions like ! PRAGMA IGNORE_TKR are not available then
33	S2 must be implemented with TYPE(*), DIMENSION().
34	Advice to implementors. If MPI_SUBARRAYS_SUPPORTED == .FALSE., the choice
35	argument may be implemented with an explicit interface using compiler directives,
36	for example:
37	INTERFACE
38	SUBROUTINE MPI(buf,)
39	DEC\$ ATTRIBUTES NO_ARG_CHECK :: buf
40	!\$PRAGMA IGNORE_TKR buf
41	!DIR\$ IGNORE_TKR buf
42	!IBM* IGNORE_TKR buf
43 44	REAL, DIMENSION(*) :: buf
44	! declarations of the other arguments
46	END SUBROUTINE
47	END INTERFACE
48	(End of advice to implementors)

18.1.7 Requirements on Fortran Compilers

 $\mathsf{MPI-3.0}$ (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 18.1.11 through 18.1.19 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 [43] 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 18.1.12 for more details.
- SEQUENCE and BIND(C) derived types are valid as actual arguments passed to choice buffer dummy arguments, and, in the case of MPI_SUBARRAYS_SUPPORTED== .FALSE., they are passed with call by reference, and passed by descriptor in the case of .TRUE..
- All actual arguments that are allowed for a dummy argument in an implicitly defined and separately compiled Fortran routine with the given compiler (e.g., CHARACTER(LEN=*) strings and array of strings) must also be valid for choice buffer dummy arguments with all Fortran support methods.
- The array dummy argument of the ISO_C_BINDING intrinsic module procedure C_F_POINTER is not restricted to Fortran types for which a corresponding type in C exists.
- The Fortran compiler shall not provide TYPE(*) unless the ASYNCHRONOUS attribute protects MPI communication as described in TS 29113. Specifically, the TS 29113 must be implemented as a whole.

The following rules are required at least as long as the compiler does not provide the extension of the ASYNCHRONOUS attribute as part of TS 29113 and there still exists a Fortran support method with MPI_ASYNC_PROTECTS_NONBLOCKING==.FALSE.. Observation of these rules by the MPI application developer is especially recomended for backward compatibility of existing applications that use the mpi module or the mpif.h include file. The rules are as follows: $\overline{7}$

 24

- 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 746 and Section 18.1.8, and DD on page 747) solve the problems described in Section 18.1.17.
 - The problems with temporary data movement (described in detail in Section 18.1.18) are solved as long as the application uses different sets of variables for the nonblocking communication (or nonblocking or split collective I/O) and the computation when overlapping communication and computation.
 - Problems caused by automatic and permanent data movement (e.g., within a garbage collection, see Section 18.1.19) 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 [43], and some of these requirements for MPI-3.0 are beyond the scope of TS 29113. (End of advice to *implementors.*)

2122 23

25

2728

30

 31 32

34

35

39

40

41

42

43 44

45

46

Additional Support for Fortran Register-Memory-Synchronization 18.1.8

 24 As described in Section 18.1.17, a dummy call may be necessary to tell the compiler that registers are to be flushed for a given buffer or that accesses to a buffer may not be moved 26across a given point in the execution sequence. Only a Fortran binding exists for this call.

MPI_F_SYNC_REG(buf) 29

INOUT

initial address of buffer (choice)

```
Fortran 2008 binding
33
```

buf

MPI_F_sync_reg(buf)

TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf

36 Fortran binding 37 MPI_F_SYNC_REG(BUF) 38 <type> BUF(*)

This routine has no executable statements. It must be compiled in the MPI library in such a manner that a Fortran compiler cannot detect in the module that the routine has an empty body. It is used only to force the compiler to flush a cached register value of a variable or buffer back to memory (when necessary), or to invalidate the register value.

Rationale. This function is not available in other languages because it would not be useful. This routine has no ierror return argument because there is no operation that can fail. (End of rationale.)

47 48

1

 $\mathbf{2}$

3

4

5

6

7

8

9 10

11

12

13

1415

16

17

18

19

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.)

If only a part of an array (e.g., defined by a subscript triplet) is Advice to users. 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.)

Additional Support for Fortran Numeric Intrinsic Types 18.1.9

MPI provides a small number of named datatypes that correspond to named intrinsic types supported by C and Fortran. These include MPI_INTEGER, MPI_REAL, MPI_INT, MPI_DOUBLE, etc., as well as the optional types MPI_REAL4. MPI_REAL8, etc. There is a one-to-one correspondence between language declarations and MPI types.

Fortran (starting with Fortran 90) provides so-called KIND-parameterized types. These types are declared using an intrinsic type (one of INTEGER, REAL, COMPLEX, LOGICAL, and 27CHARACTER) with an optional integer KIND parameter that selects from among one or more 2829variants. The specific meaning of different KIND values themselves are implementation 30 dependent and not specified by the language. Fortran provides the KIND selection functions 31selected_real_kind for REAL and COMPLEX types, and selected_int_kind for INTEGER types that allow users to declare variables with a minimum precision or number of digits. 33 These functions provide a portable way to declare KIND-parameterized REAL, COMPLEX, and 34 INTEGER variables in Fortran. This scheme is backward compatible with Fortran 77. REAL and INTEGER Fortran variables have a default KIND if none is specified. Fortran DOUBLE 3536 PRECISION variables are of intrinsic type REAL with a non-default KIND. The following two 37 declarations are equivalent:

double precision x real(KIND(0.0d0)) x

41 MPI provides two orthogonal methods for handling communication buffers of numeric 42intrinsic types. The first method (see the following section) can be used when variables have been declared in a portable way — using default KIND or using KIND parameters obtained 4344with the selected_int_kind or selected_real_kind functions. With this method, MPI automatically selects the correct data size (e.g., 4 or 8 bytes) and provides representation 4546conversion in heterogeneous environments. The second method (see "Support for size-47specific MPI Datatypes" on page 730) gives the user complete control over communication 48 by exposing machine representations.

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6

 $\overline{7}$

8

9

10

11

1213

14

15

16

17

18

1920

21

22

23 24

25

26

32

38

39

1 2 726

Parameterized Datatypes with Specified Precision and Exponent Range

MPI provides named datatypes corresponding to standard Fortran 77 numeric types: 3

MPI_INTEGER, MPI_COMPLEX, MPI_REAL, MPI_DOUBLE_PRECISION and 4

MPI_DOUBLE_COMPLEX. MPI automatically selects the correct data size and provides rep-5resentation conversion in heterogeneous environments. The mechanism described in this 6 section extends this model to support portable parameterized numeric types. 7

The model for supporting portable parameterized types is as follows. Real variables 8 are declared (perhaps indirectly) using selected_real_kind(p, r) to determine the KIND 9 parameter, where p is decimal digits of precision and r is an exponent range. Implicitly 10 MPI maintains a two-dimensional array of predefined MPI datatypes D(p, r). D(p, r) is 11 defined for each value of (p, r) supported by the compiler, including pairs for which one 12value is unspecified. Attempting to access an element of the array with an index (p, r) not 13 supported by the compiler is erroneous. MPI implicitly maintains a similar array of COMPLEX 14 datatypes. For integers, there is a similar implicit array related to selected_int_kind and 15 indexed by the requested number of digits r. Note that the predefined datatypes contained 16 in these implicit arrays are not the same as the named MPI datatypes MPI_REAL, etc., but 17a new set. 18

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.)

3132 33

19

20

21

22 23

24

25

26

27

28

29

30

34

38 39 MPI_TYPE_CREATE_F90_REAL(p, r, newtype)

35	IN	р	precision, in decimal digits (integer)
36 37	IN	r	decimal exponent range (integer)
38	OUT	newtype	the requested MPI data type (handle)

40 C binding

```
41
     int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)
```

42Fortran 2008 binding 43

```
MPI_Type_create_f90_real(p, r, newtype, ierror)
44
```

```
INTEGER, INTENT(IN) :: p, r
45
```

```
TYPE(MPI_Datatype), INTENT(OUT) :: newtype
46
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
```

```
48
     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 \mathbf{p} and \mathbf{r} or \mathbf{B} is a duplicate of such a datatype. Restrictions on using the returned datatype with the "external32" data representation are given on page 729.

It is erroneous to supply values for p and r not supported by the compiler.

MPI_TYPE_CREATE_F90_COMPLEX(p, r, newtype)

IN	р	precision, in decimal digits (integer)	
IN	r	decimal exponent range (integer)	
OUT	newtype	the requested MPI data type (handle)	

C binding

int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype)

Fortran 2008 binding MPI_Type_create_f90_complex(p, r, newtype, ierror) INTEGER, INTENT(IN) :: p, r TYPE(MPI_Datatype), INTENT(OUT) :: newtype INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding

MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR) INTEGER P, R, NEWTYPE, IERROR

This function returns a predefined MPI datatype that matches a COMPLEX variable of KIND selected_real_kind(p, r). Either p or r may be omitted from calls to selected_real_kind(p, r) (but not both). Analogously, either p or r may be set to MPI_UNDEFINED. Matching rules for datatypes created by this function are analogous to the matching rules for datatypes created by MPI_TYPE_CREATE_F90_REAL. Restrictions on using the returned datatype with the "external32" data representation are given on page 729.

It is erroneous to supply values for p and r not supported by the compiler.

MPI_TYPE_CREATE_F90_INTEGER(r, newtype) 42IN decimal exponent range, i.e., number of decimal r digits (integer) OUT the requested MPI datatype (handle) newtype

C binding

1

 $\mathbf{2}$

3

4

5

6

7

8

9

10

11

1213 14

21

22

23

 24 25

26

2728

29

30

3132

33

34

35

36

37

38

39

40 41

43

44

45

4647

```
1
     int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)
2
     Fortran 2008 binding
3
     MPI_Type_create_f90_integer(r, newtype, ierror)
4
          INTEGER, INTENT(IN) :: r
5
          TYPE(MPI_Datatype), INTENT(OUT) :: newtype
6
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
     Fortran binding
9
     MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR)
10
          INTEGER R, NEWTYPE, IERROR
11
         This function returns a predefined MPI datatype that matches a INTEGER variable of
12
     KIND selected_int_kind(r). Matching rules for datatypes created by this function are
13
     analogous to the matching rules for datatypes created by MPI_TYPE_CREATE_F90_REAL.
14
     Restrictions on using the returned datatype with the "external 32" data representation are
15
     given on page 729.
16
         It is erroneous to supply a value for r that is not supported by the compiler.
17
         Example:
18
19
     integer
                     longtype, quadtype
20
     integer, parameter :: long = selected_int_kind(15)
21
     integer(long) ii(10)
22
     real(selected_real_kind(30)) x(10)
23
     call MPI_TYPE_CREATE_F90_INTEGER(15, longtype, ierror)
24
     call MPI_TYPE_CREATE_F90_REAL(30, MPI_UNDEFINED, quadtype, ierror)
25
      . . .
26
27
     call MPI_SEND(ii, 10, longtype, ...)
28
     call MPI_SEND(x, 10, quadtype, ...)
29
30
           Advice to users.
                              The datatypes returned by the above functions are predefined
31
           datatypes. They cannot be freed; they do not need to be committed; they can be
32
           used with predefined reduction operations. There are two situations in which they
33
           behave differently syntactically, but not semantically, from the MPI named predefined
34
          datatypes.
35
             1. MPI_TYPE_GET_ENVELOPE returns special combiners that allow a program to
36
               retrieve the values of \boldsymbol{p} and \boldsymbol{r}.
37
38
             2. Because the datatypes are not named, they cannot be used as compile-time
39
               initializers or otherwise accessed before a call to one of the
40
                MPI_TYPE_CREATE_F90_XXX routines.
41
           If a variable was declared specifying a non-default KIND value that was not obtained
42
           with selected_real_kind() or selected_int_kind(), the only way to obtain a
43
           matching MPI datatype is to use the size-based mechanism described in the next
44
           section.
45
46
           (End of advice to users.)
47
48
```

Advice to implementors. An application may often repeat a call to MPI_TYPE_CREATE_F90_XXX with the same combination of (XXX,p,r). The application is not allowed to free the returned predefined, unnamed datatype handles. To prevent the creation of a potentially huge amount of handles, a high quality MPI implementation should return the same datatype handle for the same (REAL/COMPLEX/ INTEGER,p,r) combination. Checking for the combination (p,r) in the preceding call to MPI_TYPE_CREATE_F90_XXX and using a hash table to find formerly generated handles should limit the overhead of finding a previously generated datatype with same combination of (XXX,p,r). (*End of advice to implementors.*)

Rationale. The MPI_TYPE_CREATE_F90_REAL/COMPLEX/INTEGER interface needs as input the original range and precision values to be able to define useful and compiler-independent external (Section 13.5.2) or user-defined (Section 13.5.3) data representations, and in order to be able to perform automatic and efficient data conversions in a heterogeneous environment. (*End of rationale.*)

We now specify how the datatypes described in this section behave when used with the "external32" external data representation described in Section 13.5.2.

The external32 representation specifies data formats for integer and floating point values. Integer values are represented in two's complement big-endian format. Floating point values are represented by one of three IEEE formats. These are the IEEE "Single," "Double," and "Double Extended" formats, requiring 4, 8, and 16 bytes of storage, respectively. For the IEEE "Double Extended" formats, MPI specifies a Format Width of 16 bytes, with 15 exponent bits, bias = +10383, 112 fraction bits, and an encoding analogous to the "Double" format.

The external32 representations of the datatypes returned by MPI_TYPE_CREATE_F90_REAL/COMPLEX/INTEGER are given by the following rules. For MPI_TYPE_CREATE_F90_REAL:

if	(p > 33) or (r	> 4931) then exte	ernal32 representation	29
			indefined	30
else if	(p > 15) or (r	> 307) then exte	ernal32_size = 16	31
	-	> 37) then exte		32
else			ernal32_size = 4	33
			-	34
For MPI_TY	PE_CREATE_F90_	COMPLEX: twice the	size as for	35
MPI_TYPE_	CREATE_F90_RE/	AL.		36
For MPI_TY	PE_CREATE_F90_	INTEGER:		37
				38
if		-	entation is undefined	39
else if	(r > 18) then	external32_size =	16	40
else if	(r > 9) then	external32_size =	8	41
else if	(r > 4) then	external32_size =	4	
else if	(r > 2) then	external32_size =	2	42
else	. ,	external32 size =		43
0100			-	44

If the external 32 representation of a datatype is undefined, the result of using the datatype ⁴⁵ directly or indirectly (i.e., as part of another datatype or through a duplicated datatype) ⁴⁶ in operations that require the external 32 representation is undefined. These operations include MPI_PACK_EXTERNAL, MPI_UNPACK_EXTERNAL, and many MPI_FILE functions, 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

when the "external32" data representation is used. The ranges for which the external32
 representation is undefined are reserved for future standardization.

3 4

5

Support for Size-specific MPI Datatypes

MPI provides named datatypes corresponding to optional Fortran 77 numeric types that
 contain explicit byte lengths — MPI_REAL4, MPI_INTEGER8, etc. This section describes a
 mechanism that generalizes this model to support all Fortran numeric intrinsic types.

⁹ We assume that for each **typeclass** (integer, real, complex) and each word size there is ¹⁰ a unique machine representation. For every pair (**typeclass**, **n**) supported by a compiler, ¹¹ MPI must provide a named size-specific datatype. The name of this datatype is of the form ¹² MPI_<TYPE>n in C and Fortran where <TYPE> is one of REAL, INTEGER and COMPLEX, and ¹³ **n** is the length in bytes of the machine representation. This datatype locally matches all ¹⁴ variables of type (**typeclass**, **n**) in Fortran. The list of names for such types includes:

- ¹⁵ MPI_REAL4
- ¹⁶ MPI_REAL8
- ¹⁷ MPI_REAL16
- ¹³ MPI_COMPLEX8
- MPI_COMPLEX16
- MPI_COMPLEX32
- MPI_INTEGER1
- ²² MPI_INTEGER2
- MPI_INTEGER4
- MPI_INTEGER8
- ²⁵ MPI_INTEGER16

²⁷ One datatype is required for each representation supported by the Fortran compiler.

Rationale. Particularly for the longer floating-point types, C and Fortran may use different representations. For example, a Fortran compiler may define a 16-byte REAL type with 33 decimal digits of precision while a C compiler may define a 16-byte long double type that implements an 80-bit (10 byte) extended precision floating point value. Both of these types are 16 bytes long, but they are not interoperable. Thus, these types are defined by Fortran, even though C may define types of the same length. (End of rationale.)

To be backward compatible with the interpretation of these types in MPI-1, we assume that the nonstandard declarations REAL*n, INTEGER*n, always create a variable whose representation is of size n. These datatypes may also be used for variables declared with KIND=INT8/16/32/64 or KIND=REAL32/64/128, which are defined in the ISO_FORTRAN_ENV intrinsic module. Note that the MPI datatypes and the REAL*n, INTEGER*n declarations count bytes whereas the Fortran KIND values count bits. All these datatypes are predefined.

43

28

29

30

31

32

33 34

- 44 45
- 46
- 47
- 48

MPI	_TYPE_MATCH_SIZE(typ	eclass, size, datatype)	1
IN	typeclass	generic type specifier (integer)	2
IN	size	size, in bytes, of representation (integer)	3 4
	JT datatype	datatype with correct type, size (handle)	5
0	datatype	datatype with correct type, size (handle)	6
C h	inding		7
		nt typeclass, int size, MPI_Datatype *datatype)	8
			9
	ran 2008 binding	lass, size, datatype, ierror)	10
ГШ <u>т</u> .	INTEGER, INTENT(IN) :		11 12
		NTENT(OUT) :: datatype	13
	INTEGER, OPTIONAL, IN	TENT(OUT) :: ierror	14
For	ran binding		15
	0	LASS, SIZE, DATATYPE, IERROR)	16
-	INTEGER TYPECLASS, SI		17
	typoclass is one of MPL T	YPECLASS_REAL, MPI_TYPECLASS_INTEGER and	18 19
MPI	51	presenting to the desired typeclass . The function returns	20
		ocal variable of type (typeclass , size).	21
		eference (handle) to one of the predefined named datatypes,	22
not	a duplicate. This type c	annot be freed. MPI_TYPE_MATCH_SIZE can be used to	23
		at matches a Fortran numeric intrinsic type by first calling	24
	storage_size() in order to compute the variable size in bits, dividing it by eight, and then		
		SIZE to find a suitable datatype. In C, one can use the C as the size in bytes) instead of storage_size() (which returns	26
		or variables of default kind the variable's size can be computed	27 28
		EXTENT, if the typeclass is known. It is erroneous to specify	20
	e not supported by the co		30
			31
		nvenience function. Without it, it can be tedious to find the	32
	correct named type. See	note to implementors below. (End of rationale.)	33
4	Advice to implementors	This function could be implemented as a series of tests.	34
	nuovee to implementations.	This function could be implemented as a series of tests.	35
	int MPI_Type_match_s:	ize(int typeclass, int size, MPI_Datatype *rtype)	36 37
	{		38
	<pre>switch(typeclass) </pre>		39
		LASS_REAL: switch(size) {	40
		pe = MPI_REAL4; return MPI_SUCCESS;	41
	case 8: *rty] default: erre	<pre>pe = MPI_REAL8; return MPI_SUCCESS; or():</pre>	42
	}	JI (),	43
		LASS_INTEGER: switch(size) {	44 45
		<pre>ype = MPI_INTEGER4; return MPI_SUCCESS;</pre>	45 46
	case 8: *rt	<pre>ype = MPI_INTEGER8; return MPI_SUCCESS;</pre>	47
	default: er:	ror();	48

```
1
                  }
2
                 ... etc. ...
3
              }
4
5
              return MPI_SUCCESS;
6
           }
7
8
           (End of advice to implementors.)
9
10
     Communication With Size-specific Types
11
     The usual type matching rules apply to size-specific datatypes: a value sent with datatype
12
     MPI_{TYPE>n} can be received with this same datatype on another process. Most modern
13
     computers use 2's complement for integers and IEEE format for floating point. Thus, com-
14
     munication using these size-specific datatypes will not entail loss of precision or truncation
15
     errors.
16
17
           Advice to users. Care is required when communicating in a heterogeneous environ-
18
           ment. Consider the following code:
19
20
           real(selected_real_kind(5)) x(100)
21
           size = storage_size(x) / 8
22
           call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
23
           if (myrank .eq. 0) then
24
                ... initialize x ...
25
                call MPI_SEND(x, xtype, 100, 1,
                                                     ...)
26
           else if (myrank .eq. 1) then
27
                call MPI_RECV(x, xtype, 100, 0,
28
           endif
29
30
           This may not work in a heterogeneous environment if the value of size is not the
31
           same on process 1 and process 0. There should be no problem in a homogeneous
32
           environment. To communicate in a heterogeneous environment, there are at least four
33
           options. The first is to declare variables of default type and use the MPI datatypes
34
           for these types, e.g., declare a variable of type REAL and use MPI_REAL. The second
35
           is to use selected_real_kind or selected_int_kind and with the functions of the
36
           previous section. The third is to declare a variable that is known to be the same
37
           size on all architectures (e.g., selected_real_kind(12) on almost all compilers will
38
           result in an 8-byte representation). The fourth is to carefully check representation
39
           size before communication. This may require explicit conversion to a variable of size
40
           that can be communicated and handshaking between sender and receiver to agree on
41
           a size.
42
           Note finally that using the "external32" representation for I/O requires explicit at-
43
           tention to the representation sizes. Consider the following code:
44
45
46
           real(selected_real_kind(5)) x(100)
47
           size = storage_size(x) / 8
48
           call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
```

```
if (myrank .eq. 0) then
   call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo',
                                                            &
                      MPI_MODE_CREATE+MPI_MODE_WRONLY,
                                                            &
                      MPI_INFO_NULL, fh, ierror)
   call MPI_FILE_SET_VIEW(fh, zero, xtype, xtype, 'external32',&
                          MPI_INFO_NULL, ierror)
   call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
   call MPI_FILE_CLOSE(fh, ierror)
endif
call MPI_BARRIER(MPI_COMM_WORLD, ierror)
if (myrank .eq. 1) then
   call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo', MPI_MODE_RDONLY,
                 MPI_INFO_NULL, fh, ierror)
   call MPI_FILE_SET_VIEW(fh, zero, xtype, xtype, 'external32',&
                          MPI_INFO_NULL, ierror)
   call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
   call MPI_FILE_CLOSE(fh, ierror)
endif
```

If processes 0 and 1 are on different machines, this code may not work as expected if the size is different on the two machines. (*End of advice to users.*)

18.1.10 Problems With Fortran Bindings for MPI

This section discusses a number of problems that may arise when using MPI in a Fortran program. It is intended as advice to users, and clarifies how MPI interacts with Fortran. It is intended to clarify, not add to, this standard.

As noted in the original MPI specification, the interface violates the Fortran standard in several ways. While these may cause few problems for Fortran 77 programs, they become more significant for Fortran 90 programs, so that users must exercise care when using new Fortran 90 features. With Fortran 2008 and the new semantics defined in TS 29113, most violations are resolved, and this is hinted at in an addendum to each item. The violations were originally adopted and have been retained because they are important for the usability of MPI. The rest of this section describes the potential problems in detail.

The following MPI features are inconsistent with Fortran 90 and Fortran 77.

- 1. An MPI subroutine with a choice argument may be called with different argument types. When using the mpi_f08 module together with a compiler that supports Fortran 2008 + TS 29113, this problem is resolved.
- 2. An MPI subroutine with an assumed-size dummy argument may be passed an actual scalar argument. This is only solved for choice buffers through the use of DIMENSION(...).
- 3. Nonblocking and split-collective MPI routines assume that actual arguments are passed by address or descriptor and that arguments and the associated data are not copied

Unofficial Draft for Comment Only

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

1	on entrance to or exit from the subroutine. This problem is solved with the use of the ASYNCHRONOUS attribute.
3	ASTNOIRONOOS attribute.
4	4. An MPI implementation may read or modify user data (e.g., communication buffers
5	used by nonblocking communications) concurrently with a user program that is ex-
6	ecuting outside of MPI calls. This problem is resolved by relying on the extended
7	semantics of the ASYNCHRONOUS attribute as specified in TS 29113.
8	5. Several named "constants," such as MPI_BOTTOM, MPI_IN_PLACE,
9	MPI_STATUS_IGNORE, MPI_STATUSES_IGNORE, MPI_ERRCODES_IGNORE,
10	MPI_UNWEIGHTED, MPI_WEIGHTS_EMPTY, MPI_ARGV_NULL, and MPI_ARGVS_NULL
11	are not ordinary Fortran constants and require a special implementation. See Sec-
12	tion 2.5.4 for more information.
13	
14	6. The memory allocation routine MPI_ALLOC_MEM cannot be used from
15	Fortran 77/90/95 without a language extension (for example, Cray pointers) that
16	allows the allocated memory to be associated with a Fortran variable. Therefore,
17	address sized integers were used in MPI-2.0 – MPI-2.2. In Fortran 2003,
18 19	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
20	an additional, overloaded interface to support this language feature. The use of Cray
21	pointers is deprecated. The mpi_f08 module only supports TYPE(C_PTR) pointers.
22	pointers is deprecated. The mpi_100 module only supports fiff(0_f fit) pointers.
23	Additionally, MPI is inconsistent with Fortran 77 in a number of ways, as noted below.
24 25	• MPI identifiers exceed 6 characters.
26	• MPI identifiers may contain underscores after the first character.
27 28	• MPI requires an include file, mpif.h. On systems that do not support include files,
29	the implementation should specify the values of named constants.
30	
31	• 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 sup-
32	port Fortran 90-style parameterized types, INTEGER*8 or INTEGER should be used
33	instead.
34	instead.
35	MPI-1 contained several routines that take address-sized information as input or return
36	address-sized information as output. In C such arguments were of type MPI_Aint and in
37	Fortran of type INTEGER. On machines where integers are smaller than addresses, these
38 20	routines can lose information. In MPI-2 the use of these functions has been deprecated and
39 40	they have been replaced by routines taking INTEGER arguments of KIND=MPI_ADDRESS_KIND.
40 41	A number of new MPI-2 functions also take INTEGER arguments of non-default KIND. See Section 2.6 and Section 4.1.1 for more information.
42	Section 2.6 and Section 4.1.1 for more information. Sections 18.1.11 through 18.1.19 describe several problems in detail which concern
43	the interaction of MPI and Fortran as well as their solutions. Some of these solutions
44	require special capabilities from the compilers. Major requirements are summarized in
45	Section 18.1.7.
46	
47	
48	

18.1.11 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 18.1.6). 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.

18.1.12 Problems Due to Data Copying and Sequence Association with Subscript Triplets

Arrays with subscript triplets describe Fortran subarrays with or without strides, e.g.,

```
REAL a(100,100,100)
CALL MPI_Send(a(11:17, 12:99:3, 1:100), 7*30*100, MPI_REAL, ...)
```

The handling of subscript triplets depends on the value of the constant MPI_SUBARRAYS_SUPPORTED:

• If MPI_SUBARRAYS_SUPPORTED equals .TRUE.:

Choice buffer arguments are declared as TYPE(*), DIMENSION(..). For example, consider the following code fragment:

Unofficial Draft for Comment Only

 24

1	REAL s(100), r(100)
2	CALL MPI_Isend(s(1:100:5), 3, MPI_REAL,, rq, ierror)
3	CALL MPI_Wait(rq, status, ierror)
4	CALL MPI_Irecv(r(1:100:5), 3, MPI_REAL,, rq, ierror)
5	CALL MPI_Wait(rq, status, ierror)
6	
7	In this case, the individual elements $s(1)$, $s(6)$, and $s(11)$ are sent between the start
8	of MPI_ISEND and the end of MPI_WAIT even though the compiled code will not copy
9	s(1:100:5) to a real contiguous temporary scratch buffer. Instead, the compiled code
10	will pass a descriptor to MPI_ISEND that allows MPI to operate directly on $s(1)$, $s(6)$,
11	$s(11), \ldots, s(96)$. The called MPI_ISEND routine will take only the first three of these
12	elements due to the type signature "3, MPI_REAL".
13	
14	All nonblocking MPI functions (e.g., MPI_ISEND, MPI_PUT,
15	MPI_FILE_WRITE_ALL_BEGIN) behave as if the user-specified elements of choice buf-
16	fers are copied to a contiguous scratch buffer in the MPI runtime environment. All
17	datatype descriptions (in the example above, "3, MPI_REAL") read and store data
18	from and to this virtual contiguous scratch buffer. Displacements in MPI derived
19	datatypes are relative to the beginning of this virtual contiguous scratch buffer. Upon
20	completion of a nonblocking receive operation (e.g., when MPI_WAIT on a correspond-
21	ing MPI_Request returns), it is as if the received data has been copied from the virtual
22	contiguous scratch buffer back to the non-contiguous application buffer. In the ex-
23	ample above, $r(1)$, $r(6)$, and $r(11)$ are guaranteed to be defined with the received
24	data when MPI_WAIT returns.
25	Note that the above definition does not supercede restrictions about buffers used with
26	nonblocking operations (e.g., those specified in Section $3.7.2$).
27	
28	Advice to implementors. The Fortran descriptor for TYPE(*), DIMENSION()
29	arguments contains enough information that, if desired, the MPI library can make
30	a real contiguous copy of non-contiguous user buffers when the nonblocking op-
31	eration is started, and release this buffer not before the nonblocking communi-
32	cation has completed (e.g., the MPI_WAIT routine). Efficient implementations
33	may avoid such additional memory-to-memory data copying. (End of advice to
34	implementors.)
35	
36	Rationale. If MPI_SUBARRAYS_SUPPORTED equals .TRUE., non-contiguous
37	buffers are handled inside the MPI library instead of by the compiler through
38	argument association conventions. Therefore, the scope of MPI library scratch
39	buffers can be from the beginning of a nonblocking operation until the completion
40	of the operation although beginning and completion are implemented in different
41	routines. (End of rationale.)
42	
43	• If MPI_SUBARRAYS_SUPPORTED equals .FALSE.:
44	In this case, the use of Fortran arrays with subscript triplets as actual choice buffer
45	arguments in any nonblocking MPI operation (which also includes persistent request,
46	and split collectives) may cause undefined behavior. They may, however, be used in
47	blocking MPI operations.
48	

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>]... )
```

That is, there are zero or more dimensions that are selected in full, then one dimension selected without a stride, then zero or more dimensions that are selected with a simple subscript. The compiler can detect from analyzing the source code that the array is contiguous. Examples are $\mathbf{2}$

 $\overline{7}$

 $41 \\ 42$

¹Technically, the Fortran standard is worded to allow non-contiguous storage of any array data, unless the dummy argument has the CONTIGUOUS attribute.

1 2	A(1:N), A(:,N), A(:,1:N,1), A(1:6,N), A(:,:,1:N)
3	Passuge of Fortran's column major ordering where the first index varies fastest
4	Because of Fortran's column-major ordering, where the first index varies fastest, a "simply contiguous" section of a contiguous array will also be contiguous.
5	
6	The same problem can occur with a scalar argument. A compiler may make a copy of
7	scalar dummy arguments within a called procedure when passed as an actual argument
8	to a choice buffer routine. That this can cause a problem is illustrated by the example
9	
10	
11	real :: a
12	call user1(a,rq)
13	call MPI_WAIT(rq,status,ierr)
14	write (*,*) a
15	subroutine user1(buf,request)
16	call MPI_IRECV(buf,,request,)
17	end
18	end
19	If a is copied, MPI_IRECV will alter the copy when it completes the communication
20	and will not alter a itself.
21	
22	Note that copying will almost certainly occur for an argument that is a non-trivial
23	expression (one with at least one operator or function call), a section that does not
24	select a contiguous part of its parent (e.g., A(1:n:2)), a pointer whose target is such
25	a section, or an assumed-shape array that is (directly or indirectly) associated with
26	such a section.
27	If a compiler option exists that inhibits copying of arguments, in either the calling or
28	called procedure, this must be employed.
29	If a compiler makes copies in the calling procedure of arguments that are explicit-
30	shape or assumed-size arrays, "simply contiguous" array sections of such arrays, or
31	scalars, and if no compiler option exists to inhibit such copying, then the compiler
32	cannot be used for applications that use MPI_GET_ADDRESS, or any nonblocking
33	MPI routine. If a compiler copies scalar arguments in the called procedure and there
34	is no compiler option to inhibit this, then this compiler cannot be used for applications
35	that use memory references across subroutine calls as in the example above.
36	
37	18.1.13 Problems Due to Data Copying and Sequence Association with Vector Subscripts
38 39	
40	Fortran arrays with vector subscripts describe subarrays containing a possibly irregular
40	set of elements
42	REAL a(100)
43	CALL MPI_Send(A((/7,9,23,81,82/)), 5, MPI_REAL,)
44	Only in 1_bond(n((/,;0;20;01;02/)/, 0; in 1_10h);)
45	Fortran arrays with a vector subscript must not be used as actual choice buffer argu-
46	ments in any nonblocking or split collective MPI operations. They may, however, be used
47	in blocking MPI operations.
48	

18.1.14 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.

18.1.15 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
disp(3) = disp(3) - base
disp(4) = disp(4) - base
blocklen(1) = 1
```

```
1
     blocklen(2) = 1
\mathbf{2}
     blocklen(3) = 1
3
     blocklen(4) = 1
4
5
     type(1) = MPI_INTEGER
6
     type(2) = MPI_REAL
\overline{7}
     type(3) = MPI_DOUBLE_PRECISION
8
     type(4) = MPI_LOGICAL
9
10
     call MPI_TYPE_CREATE_STRUCT(4, blocklen, disp, type, newtype, ierr)
11
     call MPI_TYPE_COMMIT(newtype, ierr)
12
13
     call MPI_SEND(foo%i, 1, newtype, dest, tag, comm, ierr)
14
     ! or
15
     call MPI_SEND(foo, 1, newtype, dest, tag, comm, ierr)
16
     ! expects that base == address(foo%i) == address(foo)
17
18
     call MPI_GET_ADDRESS(fooarr(1), disp(1), ierr)
19
     call MPI_GET_ADDRESS(fooarr(2), disp(2), ierr)
20
     extent = disp(2) - disp(1)
21
     1b = 0
22
     call MPI_TYPE_CREATE_RESIZED(newtype, lb, extent, newarrtype, ierr)
23
     call MPI_TYPE_COMMIT(newarrtype, ierr)
^{24}
25
     call MPI_SEND(fooarr, 5, newarrtype, dest, tag, comm, ierr)
```

Using the derived type variable foo instead of its first basic type element foo%i may be impossible if the MPI library implements choice buffer arguments through overloading instead of using TYPE(*), DIMENSION(..), or through a non-standardized extension such as !\$PRAGMA IGNORE_TKR; see Section 18.1.6.

To use a derived type in an array requires a correct extent of the datatype handle 31 to take care of the alignment rules applied by the compiler. These alignment rules may 32 imply that there are gaps between the components of a derived type, and also between the 33 subsuguent elements of an array of a derived type. The extent of an interoperable derived 34 type (i.e., defined with BIND(C)) and a SEQUENCE derived type with the same content may 35 be different because C and Fortran may apply different alignment rules. As recommended 36 in the advice to users in Section 4.1.6, one should add an additional fifth structure element 37 with one numerical storage unit at the end of this structure to force in most cases that 38 the array of structures is contiguous. Even with such an additional element, one should 39 keep this resizing due to the special alignment rules that can be used by the compiler for 40 structures, as also mentioned in this advice. 41

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.

Unofficial Draft for Comment Only

18.1.16 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 18.1.17.
- Temporary data movement and temporary memory modifications; see Section 18.1.18.
- Permanent data movement (e.g., through garbage collection); see Section 18.1.19.

Table 18.2 shows the only usage areas where these optimization problems may occur.

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 18.2: Occurrence of Fortran optimization problems in several usage areas

The solutions in the following sections are based on compromises:

- to minimize the burden for the application programmer, e.g., as shown in Sections "Solutions" through "The (Poorly Performing) Fortran VOLATILE Attribute" on pages 744–748,
- to minimize the drawbacks on compiler based optimization, and
- to minimize the requirements defined in Section 18.1.7.

 $45 \\ 46$

1 18.1.17 Problems with Code Movement and Register Optimization $\mathbf{2}$ Nonblocking Operations 3 4 If a variable is local to a Fortran subroutine (i.e., not in a module or a COMMON block), the $\mathbf{5}$ compiler will assume that it cannot be modified by a called subroutine unless it is an actual 6 argument of the call. In the most common linkage convention, the subroutine is expected 7to save and restore certain registers. Thus, the optimizer will assume that a register which 8 held a valid copy of such a variable before the call will still hold a valid copy on return. 9 10 **Example 18.1** Fortran 90 register optimization — extreme. 11 Source compiled as or compiled as 1213 REAL :: buf, b1 REAL :: buf, b1 REAL :: buf, b1 14call MPI_IRECV(buf,..req) call MPI_IRECV(buf,..req) call MPI_IRECV(buf,..req) 15b1 = bufregister = buf 16call MPI_WAIT(req,..) call MPI_WAIT(req,..) call MPI_WAIT(req,..) 17b1 = bufb1 = register 18 19Example 18.1 shows extreme, but allowed, possibilities. MPI_WAIT on a concurrent 20thread modifies **buf** between the invocation of MPI_IRECV and the completion of MPI_WAIT. 21But the compiler cannot see any possibility that buf can be changed after MPI_IRECV has 22returned, and may schedule the load of buf earlier than typed in the source. The compiler 23has no reason to avoid using a register to hold **buf** across the call to MPI_WAIT. It also may 24 reorder the instructions as illustrated in the rightmost column. 2526Example 18.2 Similar example with MPI_ISEND 2728Source compiled as with a possible MPI-internal execution sequence 2930 REAL :: buf, copy REAL :: buf, copy REAL :: buf, copy 31 buf = val buf = val buf = val 32 call MPI_ISEND(buf,..req) addr = &buf call MPI_ISEND(buf,..req) 33 copy = buf copy= buf copy = buf34buf = val_overwrite buf = val_overwrite call MPI_WAIT(req,..) call MPI_WAIT(req,..) call send(*addr) ! within 35 ! MPI_WAIT 36 buf = val_overwrite 37 38 39

³⁹ Due to valid compiler code movement optimizations in Example 18.2, the content of ⁴⁰ buf may already have been overwritten by the compiler when the content of buf is sent. ⁴¹ The code movement is permitted because the compiler cannot detect a possible access to ⁴² buf in MPI_WAIT (or in a second thread between the start of MPI_ISEND and the end of ⁴³ MPI_WAIT).

Such register optimization is based on moving code; here, the access to buf was moved
 from after MPI_WAIT to before MPI_WAIT. Note that code movement may also occur across
 subroutine boundaries when subroutines or functions are inlined.

This register optimization/code movement problem for nonblocking operations does not occur with MPI parallel file I/O split collective operations, because in the ..._BEGIN

Unofficial Draft for Comment Only

and ..._END calls, the same buffer has to be provided as an actual argument. The register optimization / code movement problem for MPI_BOTTOM and derived MPI datatypes may occur in each blocking and nonblocking communication call, as well as in each parallel file I/O operation.

Persistent Operations

With persistent requests, the buffer argument is hidden from the MPI_START and MPI_STARTALL calls, i.e., the Fortran compiler may move buffer accesses across the MPI_START or MPI_STARTALL call, similar to the MPI_WAIT call as described in the Nonblocking Operations subsection in Section 18.1.17.

One-sided Communication

An example with instruction reordering due to register optimization can be found in Section 11.7.4.

MPI_BOTTOM and Combining Independent Variables in Datatypes

This section is only relevant if the MPI program uses a buffer argument to an MPI_SEND, MPI_RECV, etc., that hides the actual variables involved in the communication. MPI_BOTTOM with an MPI_Datatype containing *absolute addresses* is one example. Creating a datatype which uses one variable as an anchor and brings along others by using MPI_GET_ADDRESS to determine their offsets from the anchor is another. The anchor variable would be the only one referenced in the call. Also attention must be paid if MPI operations are used that run in parallel with the user's application.

Example 18.3 shows what Fortran compilers are allowed to do.

Example 18.3 Fortran 90 register optimization.

This source	can be compiled as:
call MPI_GET_ADDRESS(buf,bufaddr, ierror)	call MPI_GET_ADDRESS(buf,)
<pre>call MPI_TYPE_CREATE_STRUCT(1,1,</pre>	<pre>call MPI_TYPE_CREATE_STRUCT()</pre>
<pre>call MPI_TYPE_COMMIT(type,ierror)</pre>	call MPI_TYPE_COMMIT()
val_old = buf	register = buf
<pre>call MPI_RECV(MPI_BOTTOM,1,type,) val_new = buf</pre>	<pre>val_old = register call MPI_RECV(MPI_BOTTOM,) val_new = register</pre>

In Example 18.3, 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 18.4, 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

Unofficial Draft for Comment Only

 $\mathbf{2}$

 24

1 Example 18.4 Similar example with MPI_SEND $\mathbf{2}$ 3 This source ... can be compiled as: 4 ! buf contains val_old ! buf contains val_old 5buf = val_new 6 call MPI_SEND(MPI_BOTTOM,1,type,...) call MPI_SEND(...) 7 ! with buf as a displacement in type ! i.e. val_old is sent 8 1 9 ! buf=val_new is moved to here 10 ! and detected as dead code 11 ! and therefore removed 12L 13 buf = val_overwrite buf = val_overwrite 14 1516cannot detect that the call to MPI_SEND statement is interfering because the load access 17to buf is hidden by the usage of MPI_BOTTOM. 18 19 Solutions 2021The following sections show in detail how the problems with code movement and register 22optimization can be portably solved. Application writers can partially or fully avoid these 23compiler optimization problems by using one or more of the special Fortran declarations 24with the send and receive buffers used in nonblocking operations, or in operations in which 25MPI_BOTTOM is used, or if datatype handles that combine several variables are used: 26• Use of the Fortran ASYNCHRONOUS attribute. 2728• Use of the helper routine MPI_F_SYNC_REG, or an equivalent user-written dummy 29 routine. 30 31• Declare the buffer as a Fortran module variable or within a Fortran common block. 32 • Use of the Fortran VOLATILE attribute. 33 34 Each of these methods solves the problems of code movement and register optimization, 35 but may incur various degrees of performance impact, and may not be usable in every 36 application context. These methods may not be guaranteed by the Fortran standard, but 37 they must be guaranteed by a MPI-3.0 (and later) compliant MPI library and associated 38 compiler suite according to the requirements listed in Section 18.1.7. The performance 39 impact of using MPI_F_SYNC_REG is expected to be low, that of using module variables 40 or the ASYNCHRONOUS attribute is expected to be low to medium, and that of using the 41 VOLATILE attribute is expected to be high or very high. Note that there is one attribute 42that cannot be used for this purpose: the Fortran TARGET attribute does not solve code 43 movement problems in MPI applications. 44

45 46

The Fortran ASYNCHRONOUS Attribute

⁴⁷ Declaring an actual buffer argument with the ASYNCHRONOUS Fortran attribute in a scoping
 ⁴⁸ unit (or BLOCK) informs the compiler that any statement in the scoping unit may be executed

Unofficial Draft for Comment Only

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 3 extensions specified in TS 29113, a Fortran compiler may totally ignore this attribute if the 4 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 6 7 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 18.1.6), then the compiler has to guarantee call by reference 10 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 1112the protection of the actual buffer argument through ASYNCHRONOUS according to the TS 13 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 1415.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 18.5 Case (a) on page 751, the read accesses to b within function(b(i-1), b(i), b(i+1)) cannot be moved by compiler optimizations to before the wait call because b was declared as ASYNCHRONOUS. Note that only the elements 0, 1, 100, and 101 of b are involved in asynchronous communication but by definition, the total variable **b** is the pending communication affector and is usable for input and output asynchronous communication between the MPI_I... routines and MPI_Waitall. Case (a) works fine because the read accesses to b occur after the communication has completed.

In Case (b), the read accesses to b(1:100) in the loop i=2,99 are read accesses to 42a pending communication affector while input communication (i.e., the two MPI_Irecv 43 calls) is pending. This is a contradiction to the rule that for input communication, a 44 pending communication affector shall not be referenced. The problem can be solved by using 45separate variables for the halos and the inner array, or by splitting a common array into 46 disjoint subarrays which are passed through different dummy arguments into a subroutine, 47as shown in Example 18.9. 48

Unofficial Draft for Comment Only

1 $\mathbf{2}$

 $\mathbf{5}$

8 9

1617

18

19

20

21

22

23

 24

25

26

2728

29

30 31

32 33

34

35

36

37

38

39

40

1 2 3 4 5 6 7	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 18.3 and Example 18.4, 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.
8	Calling MPI_F_SYNC_REG
9 10 11 12 13	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 18.1.8.
1415 16	• The problems illustrated by the Examples 18.1 and 18.2 can be solved by calling $MPI_F_SYNC_REG(buf)$ once immediately after MPI_WAIT .
17	Example 18.1 Example 18.2
18	can be solved with can be solved with
19	call MPI_IRECV(buf,req) buf = val
20	call MPI_ISEND(buf,req)
21	copy = buf
22 23	<pre>call MPI_WAIT(req,) call MPI_WAIT(req,)</pre>
23	call MPI_F_SYNC_REG(buf) call MPI_F_SYNC_REG(buf)
25	b1 = buf buf = val_overwrite
26	The call to MPI_F_SYNC_REG(buf) prevents moving the last line before the
27	MPI_WAIT call. Further calls to MPI_F_SYNC_REG(buf) are not needed because it
28	is still correct if the additional read access copy=buf is moved below MPI_WAIT and
29	before buf=val_overwrite.
30	
31	• The problems illustrated by the Examples 18.3 and 18.4 can be solved with two 18.4 can be solved wi
32	additional MPI_F_SYNC_REG(buf) statements; one directly before MPI_RECV/
33 34	MPI_SEND, and one directly after this communication operation.
35	Example 18.3 Example 18.4
36	can be solved with can be solved with
37	call MPI_F_SYNC_REG(buf) call MPI_F_SYNC_REG(buf)
38	call MPI_RECV(MPI_BOTTOM,) call MPI_SEND(MPI_BOTTOM,)
39	call MPI_F_SYNC_REG(buf) call MPI_F_SYNC_REG(buf)
40	The first call to MPI_F_SYNC_REG(buf) is needed to finish all load and store refer-
41	ences to buf prior to MPI_RECV/MPI_SEND; the second call is needed to assure that
42	any subsequent access to buf is not moved before MPI_RECV/SEND.
43	
44	• In the example in Section 11.7.4, two asynchronous accesses must be protected: in
45 46	Process 1, the access to bbbb must be protected similar to Example 18.1, i.e., a call to MPL E SYNC REC(bbbb) is precised after the second MPL WIN EENCE to guarantee
40	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
48	Process 2, both calls to MPI_WIN_FENCE together act as a communication call with
	1 100005 2, both cans to with _vvitv_t Ervel together act as a communication can with

Unofficial Draft for Comment Only

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.

Sour	ce of Process 1	Source of Process 2
bbbb	= 777	buff = 999
		call MPI_F_SYNC_REG(buff)
call	MPI_WIN_FENCE	call MPI_WIN_FENCE
call	MPI_PUT(bbbb	
into	buff of process 2)	
call	MPI_WIN_FENCE	call MPI_WIN_FENCE
call	MPI_F_SYNC_REG(bbbb)	call MPI_F_SYNC_REG(buff)
		ccc = buff

• The temporary memory modification problem, i.e., Example 18.6, 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	(MP	I_BOTTOM	,)			
call	DD(buf)						

Such a user-defined routine was introduced in MPI-2.0 and is still included here to document such usage in existing application programs although new applications should prefer MPI_F_SYNC_REG or one of the other possibilities. In an existing application, calls to such a user-written routine should be substituted by a call to MPI_F_SYNC_REG because the user-written routine may not be implemented in accordance with the rules specified in Section 18.1.7.

Module Variables and COMMON Blocks

An alternative to the previously mentioned methods is to put the buffer or variable into a ⁴⁶ module or a common block and access it through a USE or COMMON statement in each scope ⁴⁷ where it is referenced, defined or appears as an actual argument in a call to an MPI routine. ⁴⁸

Unofficial Draft for Comment Only

 $\mathbf{2}$

¹ The compiler will then have to assume that the MPI procedure may alter the buffer or ² variable, provided that the compiler cannot infer that the MPI procedure does not reference ³ 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.
- 11 12 13

4

5

6

7 8

9

10

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.

20 21

22

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.

- 28
- 29 30

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.*)

31 32 33

18.1.18 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 18.5, Case (b) on page 751. Example 18.6 also shows a possibility that could be problematic.

In the compiler-generated, possible optimization in Example 18.7, buf(100,100) from Example 18.6 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 18.8 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 8 storing back of local_buf is therefore likely to interfere with asynchronously received data in buf(1,1:100).

Note that this problem may also occur:

- With the local buffer at the origin process, between an RMA communication call and the ensuing synchronization call; see Chapter 11.
- With the window buffer at the target process between two ensuing RMA synchronization calls.
- With the local buffer in MPI parallel file I/O split collective operations between the ..._BEGIN and ..._END calls; see Section 13.4.5.

As already mentioned in subsection *The Fortran ASYNCHRONOUS attribute* on page 744 of Section 18.1.17, 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 18.9 and in Example 18.10.

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 18.6 and 18.7.

Note also that compiler optimization with temporary data movement should **not** be prevented by declaring **buf** as **VOLATILE** because the **VOLATILE** implies that all accesses to any storage unit (word) of **buf** must be directly done in the main memory exactly in the sequence defined by the application program. The **VOLATILE** attribute prevents all register and cache optimizations. Therefore, **VOLATILE** may cause a huge performance degradation.

Instead of solving the problem, it is better to **prevent** the problem: when overlapping communication and computation, the nonblocking communication (or nonblocking or split collective I/O) and the computation should be executed **on different variables**, and the communication should be *protected* with the **ASYNCHRONOUS** attribute. In this case, the temporary memory modifications are done only on the variables used in the computation and cannot have any side effect on the data used in the nonblocking MPI operations.

Rationale. This is a strong restriction for application programs. To weaken this restriction, a new or modified asynchronous feature in the Fortran language would be necessary: an asynchronous attribute that can be used on parts of an array and together with asynchronous operations outside the scope of Fortran. If such a feature becomes available in a future edition of the Fortran standard, then this restriction also may be weakened in a later version of the MPI standard. (*End of rationale.*)

In Example 18.9 (which is a solution for the problem shown in Example 18.5 and in Example 18.10 (which is a solution for the problem shown in Example 18.8), the array is split into inner and halo part and both disjoint parts are passed to a subroutine separated_sections. This routine overlaps the receiving of the halo data and the calculations on the inner part of the array. In a second step, the whole array is used to do the

Unofficial Draft for Comment Only

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

1 calculation on the elements where inner+halo is needed. Note that the halo and the inner $\mathbf{2}$ area are strided arrays. Those can be used in nonblocking communication only with a TS 3 29113 based MPI library.

4 5

6

7

18.1.19 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. 8 An implementation with automatic garbage collection is one use case. Such permanent data 9 movement is in conflict with MPI in several areas: 10

11 12

13

14

15

21

- 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.

16This problem can be also solved by using the ASYNCHRONOUS attribute for such buffers. 17This MPI standard requires that the problems with permanent data movement do not 18 occur by imposing suitable restrictions on the MPI library together with the compiler used; 19 see Section 18.1.7. 20

18.1.20 Comparison with C 22

23In C, subroutines which modify variables that are not in the argument list will not cause 24 register optimization problems. This is because taking pointers to storage objects by using 25the & operator and later referencing the objects by indirection on the pointer is an integral 26part of the language. A C compiler understands the implications, so that the problem should 27not occur, in general. However, some compilers do offer optional aggressive optimization 28levels which may not be safe. Problems due to temporary memory modifications can also 29 occur in C. As above, the best advice is to avoid the problem: use different variables for 30 buffers in nonblocking MPI operations and computation that is executed while a nonblocking 31 operation is pending. 32

33 34

Example 18.5 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 18.6 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 18.7 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 18.8 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 18.9 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 18.10 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
```

18.2 Language Interoperability

18.2.1 Introduction

It is not uncommon for library developers to use one language to develop an application library that may be called by an application program written in a different language. MPI currently supports ISO (previously ANSI) C and Fortran bindings. It should be possible for applications in any of the supported languages to call MPI-related functions in another language.

Moreover, MPI allows the development of client-server code, with MPI communication used between a parallel client and a parallel server. It should be possible to code the server in one language and the clients in another language. To do so, communications should be possible between applications written in different languages.

There are several issues that need to be addressed in order to achieve interoperability.

Initialization We need to specify how the MPI environment is initialized for all languages.

- Interlanguage passing of MPI opaque objects We need to specify how MPI object handles are passed between languages. We also need to specify what happens when an MPI object is accessed in one language, to retrieve information (e.g., attributes) set in another language.
- **Interlanguage communication** We need to specify how messages sent in one language can be received in another language.

It is highly desirable that the solution for interlanguage interoperability be extensible to new languages, should MPI bindings be defined for such languages.

18.2.2 Assumptions

We assume that conventions exist for programs written in one language to call routines written in another language. These conventions specify how to link routines in different languages into one program, how to call functions in a different language, how to pass arguments between languages, and the correspondence between basic data types in different languages. In general, these conventions will be implementation dependent. Furthermore, not every basic datatype may have a matching type in other languages. For example, C character strings may not be compatible with Fortran CHARACTER variables. However, we assume that a Fortran INTEGER, as well as a (sequence associated) Fortran array of INTEGERs, can be passed to a C program. We also assume that Fortran and C have addresssized 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.

18.2.3 Initialization

A call to MPI_INIT or MPI_INIT_THREAD, from any language, initializes MPI for execution in all languages.

Advice to users. Certain implementations use the (inout) argc, argv arguments of the C version of MPI_INIT in order to propagate values for argc and argv to all

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 executing processes. Use of the Fortran version of MPI_INIT to initialize MPI may 2 result in a loss of this ability. (End of advice to users.) 3 The function MPI_INITIALIZED returns the same answer in all languages. 4 The function MPI_FINALIZE finalizes the MPI environments for all languages. 5The function MPI_FINALIZED returns the same answer in all languages. 6 The function MPI_ABORT kills processes, irrespective of the language used by the 7 caller or by the processes killed. 8 9 The MPI environment is initialized in the same manner for all languages by 10 MPI_INIT. E.g., MPI_COMM_WORLD carries the same information regardless of language: 11 same processes, same environmental attributes, same error handlers. Information can be added to info objects in one language and retrieved in another. 1213 Advice to users. The use of several languages in one MPI program may require the 14use of special options at compile and/or link time. (End of advice to users.) 1516Advice to implementors. Implementations may selectively link language specific MPI 17 libraries only to codes that need them, so as not to increase the size of binaries for codes 18 that use only one language. The MPI initialization code need perform initialization for 19 a language only if that language library is loaded. (End of advice to implementors.) 202118.2.4 Transfer of Handles 22 23Handles are passed between Fortran and C by using an explicit C wrapper to convert Fortran 24 handles to C handles. There is no direct access to C handles in Fortran. 25The type definition MPI_Eint is provided in C for an integer of the size that matches a 26Fortran INTEGER; usually, MPI_Fint will be equivalent to int. With the Fortran mpi module 27or the mpif.h include file, a Fortran handle is a Fortran INTEGER value that can be used in 28the following conversion functions. With the Fortran mpi_f08 module, a Fortran handle is a 29 BIND(C) derived type that contains an INTEGER component named MPI_VAL. This INTEGER 30 value can be used in the following conversion functions. 31 The following functions are provided in C to convert from a Fortran communicator 32 handle (which is an integer) to a C communicator handle, and vice versa. See also Sec-33 tion 2.6.4. 34 C binding 35 MPI_Comm MPI_Comm_f2c(MPI_Fint comm) 36 If comm is a valid Fortran handle to a communicator, then MPI_Comm_f2c returns a 37 valid C handle to that same communicator; if $comm = MPI_COMM_NULL$ (Fortran value), 38 then MPI_Comm_f2c returns a null C handle; if comm is an invalid Fortran handle, then 39 MPI_Comm_f2c returns an invalid C handle. 40 MPI_Fint MPI_Comm_c2f(MPI_Comm comm) 41 42The function MPI_Comm_c2f translates a C communicator handle into a Fortran handle 43 to the same communicator; it maps a null handle into a null handle and an invalid handle 44 into an invalid handle. 45 Similar functions are provided for the other types of opaque objects. 46MPI_Datatype MPI_Type_f2c(MPI_Fint datatype) 4748 MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)

MPI_Group MPI_Group_f2c(MPI_Fint group)	1
MPI_Fint MPI_Group_c2f(MPI_Group group)	2 3
MPI_Request MPI_Request_f2c(MPI_Fint request)	4
MPI_Fint MPI_Request_c2f(MPI_Request request)	5
MPI_File MPI_File_f2c(MPI_Fint file)	6 7
	8
MPI_Fint MPI_File_c2f(MPI_File file)	9
MPI_Win MPI_Win_f2c(MPI_Fint win)	10 11
MPI_Fint MPI_Win_c2f(MPI_Win win)	12
MPI_Op MPI_Op_f2c(MPI_Fint op)	13
MPI_Fint MPI_Op_c2f(MPI_Op op)	14 15
MPI_Info MPI_Info_f2c(MPI_Fint info)	16
MPI_Fint MPI_Info_c2f(MPI_Info info)	17 18
	19
MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)	20
MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)	21
MPI_Message MPI_Message_f2c(MPI_Fint message)	22 23
MPI_Fint MPI_Message_c2f(MPI_Message message)	24
MPI_Session MPI_Session_f2c(MPI_Fint session)	25
MPI_Fint MPI_Session_c2f(MPI_Session session)	26 27
	28
Example 18.11 The example below illustrates how the Fortran MPI function	29
MPI_TYPE_COMMIT can be implemented by wrapping the C MPI function	30 31
MPI_Type_commit with a C wrapper to do handle conversions. In this example a Fortran-C	32
interface is assumed where a Fortran function is all upper case when referred to from C and	33
arguments are passed by addresses.	34
! FORTRAN PROCEDURE	35
SUBROUTINE MPI_TYPE_COMMIT(DATATYPE, IERR)	36
INTEGER :: DATATYPE, IERR	37
CALL MPI_X_TYPE_COMMIT(DATATYPE, IERR)	38
RETURN	39 40
END	41
/* C wrapper */	42
,	43
<pre>void MPI_X_TYPE_COMMIT(MPI_Fint *f_handle, MPI_Fint *ierr)</pre>	44
{	45
MPI_Datatype datatype;	46 47
	41

datatype = MPI_Type_f2c(*f_handle);

47

```
*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.
                  The design here provides a convenient solution for the prevalent case,
     Rationale.
     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.)
18.2.5 Status
The following two procedures are provided in C to convert from a Fortran (with the mpi
module or mpif.h) status (which is an array of integers) to a C status (which is a structure),
and vice versa. The conversion occurs on all the information in status, including that which
is hidden. That is, no status information is lost in the conversion.
int MPI_Status_f2c(const MPI_Fint *f_status, MPI_Status *c_status)
    If f_status is a valid Fortran status, but not the Fortran value of MPI_STATUS_IGNORE
or MPI_STATUSES_IGNORE, then MPI_Status_f2c returns in c_status a valid C status with
the same content. If f_status is the Fortran value of MPI_STATUS_IGNORE or
MPI_STATUSES_IGNORE, or if f_status is not a valid Fortran status, then the call is erroneous.
    The C status has the same source, tag and error code values as the Fortran status,
and returns the same answers when queried for count, elements, and cancellation. The
conversion function may be called with a Fortran status argument that has an undefined
error field, in which case the value of the error field in the C status argument is undefined.
    Two global variables of type MPI_Fint*, MPI_F_STATUS_IGNORE and
MPI_F_STATUSES_IGNORE are declared in mpi.h. They can be used to test, in C, whether
f_status is the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE defined in
the mpi module or mpif.h. These are global variables, not C constant expressions and
cannot be used in places where C requires constant expressions. Their value is defined only
between the calls to MPI_INIT and MPI_FINALIZE and should not be changed by user code.
    To do the conversion in the other direction, we have the following:
int MPI_Status_c2f(const MPI_Status *c_status, MPI_Fint *f_status)
    This call converts a C status into a Fortran status, and has a behavior similar to
MPI_Status_f2c. That is, the value of c_status must not be either MPI_STATUS_IGNORE or
MPI_STATUSES_IGNORE.
```

1

 $\mathbf{2}$

3

4

5

6

7

8 9

10

11

12

13

1415

16

17

18

19

2021

22

23

 24

25

2627

28

29

30

 31

32

33

34

35

36

37

38

39

40

41

42

43 44

45

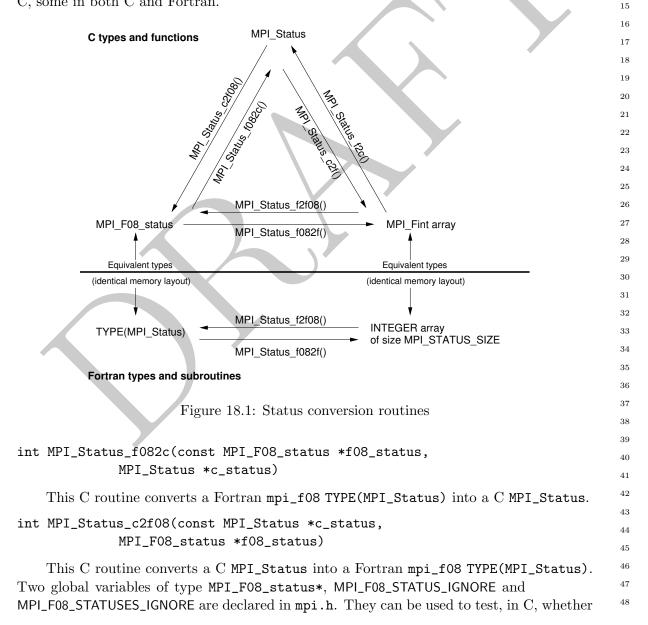
46

47

Advice to users. There exists no separate conversion function for arrays of statuses, since one can simply loop through the array, converting each status with the routines in Figure 18.1. (*End of advice to users.*)

Rationale. The handling of MPI_STATUS_IGNORE is required in order to layer libraries with only a C wrapper: if the Fortran call has passed MPI_STATUS_IGNORE, then the C wrapper must handle this correctly. Note that this constant need not have the same value in Fortran and C. If MPI_Status_f2c were to handle MPI_STATUS_IGNORE, then the type of its result would have to be MPI_Status**, which was considered an inferior solution. (*End of rationale.*)

Using the mpi_f08 Fortran module, a status is declared as TYPE(MPI_Status). The C type MPI_F08_status can be used to pass a Fortran TYPE(MPI_Status) argument into a C routine. Figure 18.1 illustrates all status conversion routines. Some are only available in C, some in both C and Fortran.



Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\overline{7}$

1 f_status is the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE defined in $\mathbf{2}$ the mpi_f08 module. These are global variables, not C constant expressions and cannot be 3 used in places where C requires constant expressions. Their value is defined only between 4 the calls to MPI_INIT and MPI_FINALIZE and should not be changed by user code. 5Conversion between the two Fortran versions of a status can be done with: 6 7 MPI_STATUS_F2F08(f_status, f08_status) 8 9 IN f_status status object declared as array 10 OUT f08_status status object declared as named type 11 12C binding 13 int MPI_Status_f2f08(MPI_Fint *f_status, MPI_F08_status *f08_status) 1415Fortran 2008 binding 16MPI_Status_f2f08(f_status, f08_status, ierror) 17 INTEGER, INTENT(IN) :: f_status(MPI_STATUS_SIZE) 18 TYPE(MPI_Status), INTENT(OUT) :: f08_status 19INTEGER, OPTIONAL, INTENT(OUT) :: ierror 20Fortran binding 21MPI_STATUS_F2F08(F_STATUS, F08_STATUS, IERROR) 22 INTEGER :: F_STATUS(MPI_STATUS_SIZE), IERROR 23TYPE(MPI_Status) :: F08_STATUS 2425This routine converts a Fortran INTEGER, DIMENSION (MPI_STATUS_SIZE) status array 26into a Fortran mpi_f08 TYPE(MPI_Status). 2728MPI_STATUS_F082F(f08_status, f_status) 2930 IN f08_status status object declared as named type 31OUT status object declared as array f_status 32 33 C binding 34 int MPI_Status_f082f(MPI_F08_status *f08_status, MPI_Fint *f_status) 35 36 Fortran 2008 binding 37 MPI_Status_f082f(f08_status, f_status, ierror) 38 TYPE(MPI_Status), INTENT(IN) :: f08_status 39 INTEGER, INTENT(OUT) :: f_status(MPI_STATUS_SIZE) 40 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 41 Fortran binding 42MPI_STATUS_F082F(F08_STATUS, F_STATUS, IERROR) 43 TYPE(MPI_Status) :: F08_STATUS 44 INTEGER :: F_STATUS(MPI_STATUS_SIZE), IERROR 4546This routine converts a Fortran mpi_f08 TYPE(MPI_Status) into a Fortran INTEGER, 47DIMENSION (MPI_STATUS_SIZE) status array. 48

18.2.6 MPI Opaque Objects

Unless said otherwise, opaque objects are "the same" in all languages: they carry the same information, and have the same meaning in both languages. The mechanism described in the previous section can be used to pass references to MPI objects from language to language. An object created in one language can be accessed, modified or freed in another language.

We examine below in more detail issues that arise for each type of MPI object.

Datatypes

Datatypes encode the same information in all languages. E.g., a datatype accessor like MPI_TYPE_GET_EXTENT will return the same information in all languages. If a datatype defined in one language is used for a communication call in another language, then the message sent will be identical to the message that would be sent from the first language: the same communication buffer is accessed, and the same representation conversion is performed, if needed. All predefined datatypes can be used in datatype constructors in any language. If a datatype is committed, it can be used for communication in any language.

The function MPI_GET_ADDRESS returns the same value in all languages. Note that we do not require that the constant MPI_BOTTOM have the same value in all languages (see Section 18.2.9).

Example 18.12

```
! FORTRAN CODE
REAL :: R(5)
INTEGER :: TYPE, IERR, AOBLEN(1), AOTYPE(1)
INTEGER (KIND=MPI_ADDRESS_KIND) :: AODISP(1)
! create an absolute datatype for array R
AOBLEN(1) = 5
CALL MPI_GET_ADDRESS(R, AODISP(1), IERR)
AOTYPE(1) = MPI_REAL
CALL MPI_TYPE_CREATE_STRUCT(1, AOBLEN, AODISP, AOTYPE, TYPE, IERR)
CALL C_ROUTINE(TYPE)
/* C code */
void C_ROUTINE(MPI_Fint *ftype)
Ł
   int count = 5;
   int lens[2] = {1,1};
  MPI_Aint displs[2];
  MPI_Datatype types[2], newtype;
   /* create an absolute datatype for buffer that consists
                                                              */
     of count, followed by R(5)
   /*
                                                               */
  MPI_Get_address(&count, &displs[0]);
```

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

1	displs[1] = 0;
2	<pre>types[0] = MPI_INT;</pre>
3	<pre>types[1] = MPI_Type_f2c(*ftype);</pre>
4	<pre>MPI_Type_create_struct(2, lens, displs, types, &newtype);</pre>
5	<pre>MPI_Type_commit(&newtype);</pre>
6	
7	<pre>MPI_Send(MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);</pre>
8	/* the message sent contains an int count of 5, followed $*/$
9	<pre>/* by the 5 REAL entries of the Fortran array R. */</pre>
10	}
11	
12	Advice to implementors. The following implementation can be used: MPI addresses,
13	as returned by MPI_GET_ADDRESS, will have the same value in all languages. One
14	obvious choice is that MPI addresses be identical to regular addresses. The address
15	is stored in the datatype, when datatypes with absolute addresses are constructed.
16	When a send or receive operation is performed, then addresses stored in a datatype
17	are interpreted as displacements that are all augmented by a base address. This base
18	address is (the address of) buf, or zero, if $buf = MPI_BOTTOM$. Thus, if MPI_BOTTOM
19	is zero then a send or receive call with $buf = MPI_BOTTOM$ is implemented exactly as
20	a call with a regular buffer argument: in both cases the base address is buf . On the
21	other hand, if MPI_BOTTOM is not zero, then the implementation has to be slightly
22	different. A test is performed to check whether $buf = MPI_BOTTOM$. If true, then the
23	base address is zero, otherwise it is buf. In particular, if MPI_BOTTOM does not have
24	the same value in Fortran and C, then an additional test for $buf = MPI_BOTTOM$ is
25	needed in at least one of the languages.
26	It may be desirable to use a value other than zero for MPI_BOTTOM even in C, so as
27	to distinguish it from a NULL pointer. If $MPI_BOTTOM = c$ then one can still avoid
28	the test $buf = MPI_BOTTOM$, by using the displacement from MPI_BOTTOM, i.e., the
29	regular address - c, as the MPI address returned by MPI_GET_ADDRESS and stored
30	in absolute datatypes. (End of advice to implementors.)
31	
32	Callback Functions
33	
34	MPI calls may associate callback functions with MPI objects: error handlers are associated
35	with communicators, files, windows, and sessions; attribute copy and delete functions are
36	associated with attribute keys; reduce operations are associated with operation objects, etc.
37	In a multilanguage environment, a function passed in an MPI call in one language may be
38	invoked by an MPI call in another language. MPI implementations must make sure that
39	such invocation will use the calling convention of the language the function is bound to.
40	
41	Advice to implementors. Callback functions need to have a language tag. This
42	tag is set when the callback function is passed in by the library function (which is
43	presumably different for each language and language support method), and is used to generate the right colling accurace when the college's function is involved. (End of
44	to generate the right calling sequence when the callback function is invoked. (End of advise to implementance)
45	advice to implementors.)
46	Advice to users. If a subroutine written in one language or Fortran support method

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.,

Unofficial Draft for Comment Only

47

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

Error Handlers

Advice to implementors. Error handlers, have, in C, a variable length argument list. It might be useful to provide to the handler information on the language environment where the error occurred. (*End of advice to implementors.*)

Reduce Operations

All predefined named and unnamed datatypes as listed in Section 5.9.2 can be used in the listed predefined operations independent of the programming language from which the MPI routine is called.

Advice to users. Reduce operations receive as one of their arguments the datatype of the operands. Thus, one can define "polymorphic" reduce operations that work for C and Fortran datatypes. (*End of advice to users.*)

18.2.7 Attributes

Attribute keys can be allocated in one language and freed in another. Similarly, attribute values can be set in one language and accessed in another. To achieve this, attribute keys will be allocated in an integer range that is valid all languages. The same holds true for system-defined attribute values (such as MPI_TAG_UB, MPI_WTIME_IS_GLOBAL, etc.).

Attribute keys declared in one language are associated with copy and delete functions in that language (the functions provided by the MPI_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 6.7 defines attributes arguments to be of type void* in C, and of type INTEGER, in Fortran. On some systems, INTEGERs will have 32 bits, while C pointers will have 64 bits. This is a problem if communicator attributes are used to move information from a Fortran caller to a C callee, or vice-versa.

MPI behaves as if it stores, internally, address sized attributes. If Fortran INTEGERs are smaller, then the (deprecated) Fortran function MPI_ATTR_GET will return the least significant part of the attribute word; the (deprecated) Fortran function MPI_ATTR_PUT will set the least significant part of the attribute word, which will be sign extended to the entire word. (These two functions may be invoked explicitly by user code, or implicitly, by attribute copying callback functions.)

Unofficial Draft for Comment Only

 24

¹ As for addresses, new functions are provided that manipulate Fortran address sized ² attributes, and have the same functionality as the old functions in C. These functions are ³ described in Section 6.7. Users are encouraged to use these new functions.

4 MPI supports two types of attributes: address-valued (pointer) attributes, and integer- $\mathbf{5}$ valued attributes. C attribute functions put and get address-valued attributes. Fortran 6 attribute functions put and get integer-valued attributes. When an integer-valued attribute 7is accessed from C, then MPI_XXX_get_attr will return the address of (a pointer to) the 8 integer-valued attribute, which is a pointer to MPI_Aint if the attribute was stored with 9 Fortran MPI_XXX_SET_ATTR, and a pointer to int if it was stored with the deprecated 10 Fortran MPI_ATTR_PUT. When an address-valued attribute is accessed from Fortran, then 11MPI_XXX_GET_ATTR will convert the address into an integer and return the result of this 12conversion. This conversion is lossless if new style attribute functions are used, and an 13 integer of kind MPI_ADDRESS_KIND is returned. The conversion may cause truncation if 14deprecated attribute functions are used. In C, the deprecated routines MPI_Attr_put and 15MPI_Attr_get behave identical to MPI_Comm_set_attr and MPI_Comm_get_attr.

```
17 Example 18.13
```

16

30

 31

A. Setting an attribute value in C int set_val = 3; struct foo set_struct;

/* Set a value that is a pointer to an int */ 24

```
MPI_Comm_set_attr(MPI_COMM_WORLD, keyval1, &set_val);
    /* Set a value that is a pointer to a struct */
```

```
MPI_Comm_set_attr(MPI_COMM_WORLD, keyval2, &set_struct);
```

```
/* Set an integer value */
```

```
MPI_Comm_set_attr(MPI_COMM_WORLD, keyval3, (void *) 17);
```

B. Reading the attribute value in C

```
<sup>32</sup> int flag, *get_val;
<sup>33</sup> struct foo *get_struct;
```

34

```
/* Upon successful return, get_val == &set_val
(and therefore *get_val == 3) */
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval1, &get_val, &flag);
/* Upon successful return, get_struct == &set_struct */
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval2, &get_struct, &flag);
/* Upon successful return, get_val == (void*) 17 */
/* i.e., (MPI_Aint) get_val == 17 */
```

MPI_Comm_get_attr(MPI_COMM_WORLD, keyval3, &get_val, &flag);

C. Reading the attribute value with (deprecated) Fortran MPI-1 calls

45 46

42

43

44

46

```
1
LOGICAL FLAG
                                                                                      \mathbf{2}
INTEGER IERR, GET_VAL, GET_STRUCT
                                                                                      3
! Upon successful return, GET_VAL == &set_val, possibly truncated
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL1, GET_VAL, FLAG, IERR)
! Upon successful return, GET_STRUCT == &set_struct, possibly truncated
                                                                                      6
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL2, GET_STRUCT, FLAG, IERR)
! Upon successful return, GET_VAL == 17
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL3, GET_VAL, FLAG, IERR)
                                                                                      10
    D. Reading the attribute value with Fortran MPI-2 calls
                                                                                      11
                                                                                      12
LOGICAL FLAG
                                                                                      13
INTEGER IERR
                                                                                      14
INTEGER (KIND=MPI_ADDRESS_KIND) GET_VAL, GET_STRUCT
                                                                                      15
                                                                                      16
! Upon successful return, GET_VAL == &set_val
                                                                                      17
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL1, GET_VAL, FLAG, IERR)
                                                                                      18
! Upon successful return, GET_STRUCT == &set_struct
                                                                                      19
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL2, GET_STRUCT, FLAG, IERR)
                                                                                      20
! Upon successful return, GET_VAL == 17
                                                                                      21
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL3, GET_VAL, FLAG, IERR)
                                                                                      22
                                                                                      23
                                                                                      ^{24}
Example 18.14 A. Setting an attribute value with the (deprecated) Fortran MPI-1 call
                                                                                      25
                                                                                      26
INTEGER IERR, VAL
                                                                                      27
VAL = 7
                                                                                      28
CALL MPI_ATTR_PUT(MPI_COMM_WORLD, KEYVAL, VAL, IERR)
                                                                                      29
                                                                                      30
    B. Reading the attribute value in C
                                                                                      31
                                                                                      32
int flag;
                                                                                      33
int *value;
                                                                                      34
                                                                                      35
/* Upon successful return, value points to internal MPI storage and
                                                                                      36
   *value == (int) 7 */
                                                                                      37
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval, &value, &flag);
                                                                                      38
    C. Reading the attribute value with (deprecated) Fortran MPI-1 calls
                                                                                      39
                                                                                      40
LOGICAL FLAG
                                                                                      41
INTEGER IERR, VALUE
                                                                                      42
                                                                                      43
! Upon successful return, VALUE == 7
                                                                                      44
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR)
                                                                                      45
                                                                                      46
    D. Reading the attribute value with Fortran MPI-2 calls
                                                                                      47
                                                                                      48
```

```
1
     LOGICAL FLAG
\mathbf{2}
     INTEGER IERR
3
     INTEGER (KIND=MPI_ADDRESS_KIND) VALUE
4
\mathbf{5}
     ! Upon successful return, VALUE == 7 (sign extended)
6
     CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR)
7
8
     Example 18.15 A. Setting an attribute value via a Fortran MPI-2 call
9
10
     INTEGER IERR
     INTEGER(KIND=MPI_ADDRESS_KIND) VALUE1
11
     INTEGER(KIND=MPI_ADDRESS_KIND) VALUE2
12
     VALUE1 = 42
13
     VALUE2 = INT(2, KIND=MPI_ADDRESS_KIND) ** 40
14
15
     CALL MPI_COMM_SET_ATTR(MPI_COMM_WORLD, KEYVAL1, VALUE1, IERR)
16
17
     CALL MPI_COMM_SET_ATTR(MPI_COMM_WORLD, KEYVAL2, VALUE2, IERR)
18
         B. Reading the attribute value in C
19
20
     int flag;
     MPI_Aint *value1, *value2;
21
22
23
     /* Upon successful return, value1 points to internal MPI storage and
^{24}
        *value1 == 42 */
     MPI_Comm_get_attr(MPI_COMM_WORLD, keyval1, &value1, &flag);
25
26
     /* Upon successful return, value2 points to internal MPI storage and
        *value2 == 2^40 */
27
     MPI_Comm_get_attr(MPI_COMM_WORLD, keyval2, &value2, &flag);
28
29
         C. Reading the attribute value with (deprecated) Fortran MPI-1 calls
30
^{31}
     LOGICAL FLAG
32
     INTEGER IERR, VALUE1, VALUE2
33
34
     ! Upon successful return, VALUE1 == 42
35
     CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL1, VALUE1, FLAG, IERR)
36
     ! Upon successful return, VALUE2 == 2<sup>40</sup>, or 0 if truncation
37
     ! needed (i.e., the least significant part of the attribute word)
38
     CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL2, VALUE2, FLAG, IERR)
39
         D. Reading the attribute value with Fortran MPI-2 calls
40
41
     LOGICAL FLAG
42
     INTEGER IERR
43
     INTEGER (KIND=MPI_ADDRESS_KIND) VALUE1, VALUE2
44
45
     ! Upon successful return, VALUE1 == 42
46
     CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL1, VALUE1, FLAG, IERR)
47
     ! Upon successful return, VALUE2 == 2^40
48
     CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL2, VALUE2, FLAG, IERR)
```

MPI_TAG_UB, &p, &flag) will return in p a pointer to an int containing the upper bound for tag value.

Address-valued predefined attributes, such as MPI_WIN_BASE behave as if they were put by a C call, i.e., in Fortran, MPI_WIN_GET_ATTR(win, MPI_WIN_BASE, val, flag, ierror) will return in val the base address of the window, converted to an integer. In C, MPI_Win_get_attr(win, MPI_WIN_BASE, &p, &flag) will return in p a pointer to the window base, cast to (void *).

Rationale. The design is consistent with the behavior specified for predefined attributes, and ensures that no information is lost when attributes are passed from language to language. Because the language interoperability for predefined attributes was defined based on MPI_ATTR_PUT, this definition is kept for compatibility reasons although the routine itself is now deprecated. (*End of rationale.*)

Advice to implementors. Implementations should tag attributes either as (1) address attributes, (2) as INTEGER(KIND=MPI_ADDRESS_KIND) attributes or (3) as INTEGER attributes, according to whether they were set in (1) C (with MPI_Attr_put or MPI_XXX_set_attr), (2) in Fortran with MPI_XXX_SET_ATTR or (3) with the deprecated Fortran routine MPI_ATTR_PUT. Thus, the right choice can be made when the attribute is retrieved. (End of advice to implementors.)

18.2.8 Extra-State

Extra-state should not be modified by the copy or delete callback functions. (This is obvious from the C binding, but not obvious from the Fortran binding). However, these functions may update state that is indirectly accessed via extra-state. E.g., in C, extra-state can be a pointer to a data structure that is modified by the copy or callback functions; in Fortran, extra-state can be an index into an entry in a COMMON array that is modified by the copy or callback functions. In a multithreaded environment, users should be aware that distinct threads may invoke the same callback function concurrently: if this function modifies state associated with extra-state, then mutual exclusion code must be used to protect updates and accesses to the shared state.

18.2.9 Constants

MPI constants have the same value in all languages, unless specified otherwise. This does not apply to constant handles (MPI_INT, MPI_COMM_WORLD, MPI_ERRORS_RETURN, MPI_SUM, etc.) These handles need to be converted, as explained in Section 18.2.4. Constants that specify maximum lengths of strings (see Section A.1.1 for a listing) have a value one less in Fortran than C since in C the length includes the null terminating character. Thus, these constants represent the amount of space which must be allocated to hold the largest possible such string, rather than the maximum number of printable characters the string could contain.

$47 \\ 48$

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\mathbf{5}$

 $\mathbf{6}$

 $\overline{7}$

 24

Advice to users. This definition means that it is safe in C to allocate a buffer to receive a string using a declaration like

char name [MPI_MAX_OBJECT_NAME];

(End of advice to users.)

Also constant "addresses," i.e., special values for reference arguments that are not handles, such as MPI_BOTTOM or MPI_STATUS_IGNORE may have different values in different languages.

Rationale. The current MPI standard specifies that MPI_BOTTOM can be used in initialization expressions in C, but not in Fortran. Since Fortran does not normally support call by value, then MPI_BOTTOM in Fortran must be the name of a predefined static variable, e.g., a variable in an MPI declared COMMON block. On the other hand, in C, it is natural to take MPI_BOTTOM = 0 (Caveat: Defining MPI_BOTTOM = 0 implies that NULL pointer cannot be distinguished from MPI_BOTTOM; it may be that MPI_BOTTOM = 1 is better. See the advice to implementors in the Datatypes subsection in Section 18.2.6) Requiring that the Fortran and C values be the same will complicate the initialization process. (End of rationale.)

²² 18.2.10 Interlanguage Communication

The type matching rules for communication in MPI are not changed: the datatype specifitraction for each item sent should match, in type signature, the datatype specification used to receive this item (unless one of the types is MPI_PACKED). Also, the type of a message item should match the type declaration for the corresponding communication buffer location, unless the type is MPI_BYTE or MPI_PACKED. Interlanguage communication is allowed if it complies with these rules.

Example 18.16 In the example below, a Fortran array is sent from Fortran and received in C.

```
33
     ! FORTRAN CODE
34
     SUBROUTINE MYEXAMPLE()
35
     USE mpi_f08
36
     REAL :: R(5)
37
     INTEGER :: IERR, MYRANK, AOBLEN(1)
38
     TYPE(MPI_Datatype) :: TYPE, AOTYPE(1)
39
     INTEGER (KIND=MPI_ADDRESS_KIND) :: AODISP(1)
40
^{41}
     ! create an absolute datatype for array R
42
     AOBLEN(1) = 5
43
     CALL MPI_GET_ADDRESS(R, AODISP(1), IERR)
^{44}
     AOTYPE(1) = MPI_REAL
45
     CALL MPI_TYPE_CREATE_STRUCT(1, AOBLEN, AODISP, AOTYPE, TYPE, IERR)
46
     CALL MPI_TYPE_COMMIT(TYPE, IERR)
47
48
     CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYRANK, IERR)
```

1

2

3 4

5 6

7

8

9

10 11

12

13

14

15

16

17

18

19

```
IF (MYRANK.EQ.0) THEN
CALL MPI_SEND(MPI_BOTTOM, 1, TYPE, 1, 0, MPI_COMM_WORLD, IERR)
ELSE
CALL C_ROUTINE(TYPE%MPI_VAL)
END IF
END SUBROUTINE
/* C code */
void C_ROUTINE(MPI_Fint *fhandle)
{
    MPI_Datatype type;
    MPI_Status status;
    type = MPI_Type_f2c(*fhandle);
    MPI_Recv(MPI_BOTTOM, 1, type, 0, 0, MPI_COMM_WORLD, &status);
}
```

MPI implementors may weaken these type matching rules, and allow messages to be sent with Fortran types and received with C types, and vice versa, when those types match. I.e., if the Fortran type INTEGER is identical to the C type int, then an MPI implementation may allow data to be sent with datatype MPI_INTEGER and be received with datatype MPI_INT. However, such code is not portable.

 $\mathbf{2}$

 24



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
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

1		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		MPI_ERR_CONVERSION
11		MPI_ERR_DISP
12		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_SHARED
36		MPI_ERR_RMA_SYNC
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		(P~6°)
46		
47		

	Error classes (continued)	1
-	C type: const int (or unnamed enum)	2
	Fortran type: INTEGER	3
-	MPI_T_ERR_CANNOT_INIT	4
	MPI_T_ERR_NOT_ACCESSIBLE	5
	MPI_T_ERR_NOT_INITIALIZED	6
	MPI_T_ERR_NOT_SUPPORTED	7
	MPI_T_ERR_MEMORY	8
	MPI_T_ERR_INVALID	9
	MPI_T_ERR_INVALID_INDEX	10
	MPI_T_ERR_INVALID_ITEM	11
	MPI_T_ERR_INVALID_SESSION	12
	MPI_T_ERR_INVALID_HANDLE	13
	MPI_T_ERR_INVALID_NAME	14
	MPI_T_ERR_OUT_OF_HANDLES	15
	MPI_T_ERR_OUT_OF_SESSIONS	16
	MPI_T_ERR_CVAR_SET_NOT_NOW	17
	MPI_T_ERR_CVAR_SET_NEVER	18
	MPI_T_ERR_PVAR_NO_WRITE	19
	MPI_T_ERR_PVAR_NO_STARTSTOP	20
	MPI_T_ERR_PVAR_NO_ATOMIC	21
	MPI_ERR_LASTCODE	22
-		23
	Buffer Address Constants	24
C type: void * c		25
	addingd momory location) ¹	
	edefined memory location) ¹	26
MPI_BOTTOM		27
MPI_BOTTOM MPI_IN_PLACE		27 28
MPI_BOTTOM MPI_IN_PLACE	ortran these constants are not usable for initialization	27 28 29
MPI_BOTTOM MPI_IN_PLACE		27 28 29 30
MPI_BOTTOM MPI_IN_PLACE	ortran these constants are not usable for initialization assignment. See Section 2.5.4.	27 28 29 30 31
MPI_BOTTOM MPI_IN_PLACE	ortran these constants are not usable for initialization assignment. See Section 2.5.4. Assorted Constants	27 28 29 30 31 32
MPI_BOTTOM MPI_IN_PLACE	ortran these constants are not usable for initialization assignment. See Section 2.5.4. Assorted Constants C type: const int (or unnamed enum)	27 28 29 30 31 32 33
MPI_BOTTOM MPI_IN_PLACE	ortran these constants are not usable for initialization assignment. See Section 2.5.4. Assorted Constants C type: const int (or unnamed enum) Fortran type: INTEGER	27 28 29 30 31 32 33 34
MPI_BOTTOM MPI_IN_PLACE	ortran these constants are not usable for initialization assignment. See Section 2.5.4. Assorted Constants C type: const int (or unnamed enum) Fortran type: INTEGER MPI_PROC_NULL	27 28 29 30 31 32 33 34 35
MPI_BOTTOM MPI_IN_PLACE	ortran these constants are not usable for initialization assignment. See Section 2.5.4. Assorted Constants C type: const int (or unnamed enum) Fortran type: INTEGER MPI_PROC_NULL MPI_ANY_SOURCE	27 28 29 30 31 32 33 34 35 36
MPI_BOTTOM MPI_IN_PLACE	ortran these constants are not usable for initialization assignment. See Section 2.5.4. Assorted Constants C type: const int (or unnamed enum) Fortran type: INTEGER MPI_PROC_NULL MPI_ANY_SOURCE MPI_ANY_TAG	27 28 29 30 31 32 33 34 35
MPI_BOTTOM MPI_IN_PLACE	ortran these constants are not usable for initialization assignment. See Section 2.5.4. Assorted Constants C type: const int (or unnamed enum) Fortran type: INTEGER MPI_PROC_NULL MPI_ANY_SOURCE MPI_ANY_TAG MPI_UNDEFINED	27 28 29 30 31 32 33 34 35 36 37
MPI_BOTTOM MPI_IN_PLACE	ortran these constants are not usable for initialization assignment. See Section 2.5.4. Assorted Constants C type: const int (or unnamed enum) Fortran type: INTEGER MPI_PROC_NULL MPI_ANY_SOURCE MPI_ANY_TAG MPI_UNDEFINED MPI_BSEND_OVERHEAD	27 28 29 30 31 32 33 34 35 36 37 38
MPI_BOTTOM MPI_IN_PLACE	ortran these constants are not usable for initialization assignment. See Section 2.5.4. <u>Assorted Constants</u> C type: const int (or unnamed enum) Fortran type: INTEGER MPI_PROC_NULL MPI_ANY_SOURCE MPI_ANY_TAG MPI_UNDEFINED MPI_BSEND_OVERHEAD MPI_KEYVAL_INVALID	27 28 29 30 31 32 33 34 35 36 37 38 39
MPI_BOTTOM MPI_IN_PLACE	ortran these constants are not usable for initialization assignment. See Section 2.5.4. <u>Assorted Constants</u> C type: const int (or unnamed enum) Fortran type: INTEGER MPI_PROC_NULL MPI_ANY_SOURCE MPI_ANY_TAG MPI_UNDEFINED MPI_BSEND_OVERHEAD MPI_KEYVAL_INVALID MPI_LOCK_EXCLUSIVE	27 28 29 30 31 32 33 34 35 36 37 38 39 40
MPI_BOTTOM MPI_IN_PLACE	ortran these constants are not usable for initialization assignment. See Section 2.5.4. Assorted Constants C type: const int (or unnamed enum) Fortran type: INTEGER MPI_PROC_NULL MPI_ANY_SOURCE MPI_ANY_TAG MPI_UNDEFINED MPI_BSEND_OVERHEAD MPI_BSEND_OVERHEAD MPI_LOCK_EXCLUSIVE MPI_LOCK_SHARED	27 28 29 30 31 32 33 34 35 36 37 38 39 40 41
MPI_BOTTOM MPI_IN_PLACE	ortran these constants are not usable for initialization assignment. See Section 2.5.4. <u>Assorted Constants</u> C type: const int (or unnamed enum) Fortran type: INTEGER MPI_PROC_NULL MPI_ANY_SOURCE MPI_ANY_TAG MPI_UNDEFINED MPI_BSEND_OVERHEAD MPI_KEYVAL_INVALID MPI_LOCK_EXCLUSIVE	27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
MPI_BOTTOM MPI_IN_PLACE	ortran these constants are not usable for initialization assignment. See Section 2.5.4. <u>Assorted Constants</u> C type: const int (or unnamed enum) Fortran type: INTEGER MPI_PROC_NULL MPI_ANY_SOURCE MPI_ANY_TAG MPI_UNDEFINED MPI_BSEND_OVERHEAD MPI_KEYVAL_INVALID MPI_LOCK_EXCLUSIVE MPI_LOCK_SHARED MPI_ROOT	27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
MPI_BOTTOM MPI_IN_PLACE ¹ Note that in Fe expressions or	ortran these constants are not usable for initialization assignment. See Section 2.5.4. Assorted Constants C type: const int (or unnamed enum) Fortran type: INTEGER MPI_PROC_NULL MPI_ANY_SOURCE MPI_ANY_SOURCE MPI_ANY_TAG MPI_UNDEFINED MPI_BSEND_OVERHEAD MPI_KEYVAL_INVALID MPI_LOCK_EXCLUSIVE MPI_LOCK_SHARED MPI_ROOT No Process Message Handle	27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
MPI_BOTTOM MPI_IN_PLACE ¹ Note that in Fe expressions or	ortran these constants are not usable for initialization assignment. See Section 2.5.4. <u>Assorted Constants</u> C type: const int (or unnamed enum) Fortran type: INTEGER MPI_PROC_NULL MPI_ANY_SOURCE MPI_ANY_TAG MPI_UNDEFINED MPI_BSEND_OVERHEAD MPI_KEYVAL_INVALID MPI_LOCK_EXCLUSIVE MPI_LOCK_SHARED MPI_ROOT <u>No Process Message Handle</u> type: MPI_Message	27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
MPI_BOTTOM MPI_IN_PLACE ¹ Note that in F expressions or C t For	ortran these constants are not usable for initialization assignment. See Section 2.5.4. Assorted Constants C type: const int (or unnamed enum) Fortran type: INTEGER MPI_PROC_NULL MPI_ANY_SOURCE MPI_ANY_SOURCE MPI_ANY_TAG MPI_UNDEFINED MPI_BSEND_OVERHEAD MPI_KEYVAL_INVALID MPI_LOCK_EXCLUSIVE MPI_LOCK_SHARED MPI_ROOT No Process Message Handle	27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45

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 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	Variable Address Size (Fortran only)
14	Fortran type: INTEGER
15	MPI_ADDRESS_KIND
16	MPI_COUNT_KIND
17	MPI_INTEGER_KIND
18	MPI_OFFSET_KIND
19	
20	Error-handling specifiers
21	C type: MPI_Errhandler
22	Fortran type: INTEGER or TYPE(MPI_Errhandler)
23	MPI_ERRORS_ARE_FATAL
24	MPI_ERRORS_RETURN
25	
26	Maximum Sizes for Strings
27	C type: const int (or unnamed enum)
28	Fortran type: INTEGER
29	MPI_MAX_DATAREP_STRING
30	MPI_MAX_ERROR_STRING
31	MPI_MAX_INFO_KEY
32	MPI_MAX_INFO_VAL
33	MPI_MAX_LIBRARY_VERSION_STRING
34	MPI_MAX_OBJECT_NAME
35	MPI_MAX_PORT_NAME
36	MPI_MAX_PROCESSOR_NAME
37	MPI_MAX_FROM_GROUP_TAG
38	MPI_MAX_PSET_NAME_LEN
39	
40	
41	
42	
43	
44	
45	
46	
47	
48	

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	
3	Fortran type: INTEGER	
4	or TYPE(MPI_Datatype)	
5	MPI_INTEGER	INTEGER
6	MPI_REAL	REAL
7	MPI_DOUBLE_PRECISION	DOUBLE PRECISION
8	MPI_COMPLEX	COMPLEX
9	MPI_LOGICAL	LOGICAL
10	MPI_CHARACTER	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	$s^1 \mid C++ types$
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	iler is missing, then the
26	MPI datatypes in this table are	not defined.
27		
28	Optional datatypes (F	ortran) Fortran types
29	$\mathrm{C} \; \mathrm{type}$: MPI_Datatype	
30	Fortran type: INTEGER	
31	or TYPE(MPI_Datatype)	
32	MPI_DOUBLE_COMPLEX	DOUBLE COMPLEX
33	MPI_INTEGER1	INTEGER*1
34	MPI_INTEGER2	INTEGER*2
35	MPI_INTEGER4	INTEGER*4
36	MPI_INTEGER8	INTEGER*8
37	MFT_INTEGERTO INTEGERT*10	
38	MPI_REAL2 REAL*2	
39	MPI_REAL4 REAL*4	
40	MPI_REAL8	REAL*8
41	MPI_REAL16	REAL*16
42	MPI_COMPLEX4	COMPLEX*4
43	MPI_COMPLEX8	COMPLEX*8
44	MPI_COMPLEX16	COMPLEX*16
45	MPI_COMPLEX32	COMPLEX*32
46		
47		
48		

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
	8
MPI_LONG_DOUBLE_INT	9
	10
Datatypes for reduction functions (Fortran)	11
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
	22
Communicator split type constants	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
MPI_HOST	47
MPI_WTIME_IS_GLOBAL	48

1	Collective Operations
2	C type: MPI_Op
3	Fortran type: INTEGER or TYPE(MPI_Op)
4	MPI_MAX
5	MPI_MIN
6	MPI_SUM
7	MPI_PROD
8	
9	MPI_MAXLOC
10	MPI_MINLOC
10	MPI_BAND
	MPI_BOR
12	MPI_BXOR
13	MPI_LAND
14	MPI_LOR
15	MPI_LXOR
16	MPI_REPLACE
17	MPI_NO_OP
18	
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	<pre>MPI_Datatype / INTEGER or TYPE(MPI_Datatype)</pre>
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	<pre>MPI_File / INTEGER or TYPE(MPI_File)</pre>
36	MPI_INFO_NULL
37	<pre>MPI_Info / INTEGER or TYPE(MPI_Info)</pre>
38	MPI_SESSION_NULL
39	MPI_Session / INTEGER MPI_WIN_NULL
40	MPI_Win / INTEGER or TYPE(MPI_Win)
41	MPI_MESSAGE_NULL
42	MPI_Message / INTEGER or TYPE(MPI_Message)
43	
44	Empty group
45	C type: MPI_Group
46	Fortran type: INTEGER or TYPE(MPI_Group)
47	MPI_GROUP_EMPTY
48	

	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 mp	i module / Fortran type with mpi_f08 module
MPI_COMM_NULL_CO	
MPI_Comm_copy_attr_:	function
/ COMM_COPY_ATTR_FUN	
MPI_COMM_DUP_FN	
MPI_Comm_copy_attr_:	
/ COMM_COPY_ATTR_FUN	
MPI_COMM_NULL_DEI	LETE_FN
MPI_Comm_delete_att:	-
/ COMM_DELETE_ATTR_F	· · · · · · · · · · · · · · · · · · ·
MPI_WIN_NULL_COPY	
MPI_Win_copy_attr_f	
/ WIN_COPY_ATTR_FUNC	CTION / PROCEDURE(MPI_Win_copy_attr_function) ¹)
MPI_WIN_DUP_FN	
MPI_Win_copy_attr_f	
/ WIN_COPY_ATTR_FUNC	· · · · · · · · · · · · · · · · · · ·
MPI_WIN_NULL_DELE	
MPI_Win_delete_attr	
/ WIN_DELETE_ATTR_FU MPI_TYPE_NULL_COP	
MPI_Type_copy_attr_:	_
/ TYPE_COPY_ATTR_FUN	
MPI_TYPE_DUP_FN	(it is a second of the second
MPI_Type_copy_attr_:	function
/ TYPE_COPY_ATTR_FUN	
MPI_TYPE_NULL_DELE	
MPI_Type_delete_att:	
/ TYPE_DELETE_ATTR_F	FUNCTION / PROCEDURE(MPI_Type_delete_attr_function) 1)
MPI_CONVERSION_FN	
MPI_Datarep_convers	ion_function
	$M_{FUNCTION} $ / PROCEDURE(MPI_Datarep_conversion_function) 1)
/ DATAREP_CONVERSION	
	ementors (on page 310) and advice to users (on page 310)
^{1} See the advice to impl	ementors (on page 310) and advice to users (on page 310) rtran functions MPI_COMM_NULL_COPY_FN, in

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
2	C type: const int (or unnamed enum)
3	Fortran type: INTEGER
4	MPI_APPNUM
5	MPI_LASTUSEDCODE
6	_
7	MPI_UNIVERSE_SIZE
8	MPI_WIN_BASE
9	MPI_WIN_SIZE
0	MPI_WIN_CREATE_FLAVOR
1	MPI_WIN_MODEL
2	
3	MPI Window Create Flavors
4	C type: const int (or unnamed enum)
5	Fortran type: INTEGER
6	MPI_WIN_FLAVOR_CREATE
7	MPI_WIN_FLAVOR_ALLOCATE
8	MPI_WIN_FLAVOR_DYNAMIC
9	MPI_WIN_FLAVOR_SHARED
0	
1	MPI Window Models
2	C type: const int (or unnamed enum)
3	Fortran type: INTEGER
4	MPI_WIN_SEPARATE
5	MPI_WIN_UNIFIED
3	
7	
3	
9	7
0	
1	
2	
3	
4	
4 5	
13 14 15 16 17	

-	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_NAMED	
	MPI_COMBINER_RESIZED	
	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	
-		

	Elle Or enstien Constants Dest 1
1	File Operation Constants, Part 1
2	C type: const MPI_Offset (or unnamed enum)
3	Fortran type: INTEGER (KIND=MPI_OFFSET_KIND)
4 5	MPI_DISPLACEMENT_CURRENT
6	File Or metion Constants Dest 2
6 7	File Operation Constants, Part 2
8	C type: const int (or unnamed enum)
9	Fortran type: INTEGER
10	
10	
12	MPI_DISTRIBUTE_DFLT_DARG
13	MPI_DISTRIBUTE_NONE
14	MPI_ORDER_C
15	MPI_ORDER_FORTRAN
16	MPI_SEEK_CUR
17	MPI_SEEK_END
18	MPI_SEEK_SET
19	E00 Detetune Metching Constants
20	F90 Datatype Matching Constants
21	C type: const int (or unnamed enum) Fortran type: INTEGER
22	MPI_TYPECLASS_COMPLEX
23	MPI_TYPECLASS_COMPLEX 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	char** / array of CHARACTER*(*)
33	MPI_ERRCODES_IGNORE
34	int* / INTEGER array
35	MPI_STATUSES_IGNORE
36	MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)
37	or TYPE(MPI_Status), DIMENSION(*)
38	MPI_STATUS_IGNORE
39	MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)
40	or TYPE(MPI_Status)
41	MPI_UNWEIGHTED
42	int* / INTEGER array
43	MPI_WEIGHTS_EMPTY
44	int* / INTEGER array
45	$\frac{1}{1}$ Note that in Fortran these constants are not usable for initialization
46	expressions or assignment. See Section 2.5.4.
47	T
48	

C type: MPI_Fint*	ing Ignored Input (no Fortran) equivalent to Fortran
/PI_F_STATUSES_IGNORE	MPI_STATUSES_IGNORE in mpi / mpif.h
IPI_F_STATUS_IGNORE	MPI_STATUS_IGNORE in mpi / mpif.h
type: MPI_F08_status*	equivalent to Fortran
IPI_F08_STATUSES_IGNORE	MPI_STATUSES_IGNORE in mpi_f08
1PI_F08_STATUS_IGNORE	MPI_STATUS_IGNORE in mpi_f08
	stants and Fortran Parameters hacro that expands to an int value
Null handles used in th	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	LL
MPI_T_pvar_session	
	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 MPI_T_VERBOSITY_MPID	
MPI_T_VERBOSITY_MPID	
MPI_T_VERBOSITY_MPID	
	EV_ALL
*	
•	
*	
•	
•	
•	

Unofficial Draft for Comment Only

784	ANNEX A. LANGUAGE BINDINGS SUMMARY
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	MPI_T_SCOPE_GROUP
23	MPI_T_SCOPE_GROUP_EQ
24	MPI_T_SCOPE_ALL
25	MPI_T_SCOPE_ALL_EQ
26	
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	
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 37	MPI_T_PVAR_CLASS_LEVEL
38	MPI_T_PVAR_CLASS_SIZE
39	MPI_T_PVAR_CLASS_PERCENTAGE
40	MPI_T_PVAR_CLASS_HIGHWATERMARK
41	MPI_T_PVAR_CLASS_LOWWATERMARK
42	MPI_T_PVAR_CLASS_COUNTER
43	MPI_T_PVAR_CLASS_AGGREGATE
44	MPI_T_PVAR_CLASS_TIMER
45	MPI_T_PVAR_CLASS_GENERIC
46	
47	
48	

Source event ordering guarantees in the	1
MPI 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 and defined O tone definitions included in the file and h	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
/* C handles to assorted structures */	26
MPI_Comm	27
MPI_Datatype	28
MPI_Errhandler	29
MPI_File	30 31
MPI_Group	31
MPI_Info	33
MPI_Message	34
MPI_Op	35
MPI_Request	36
MPI_Session	37
MPI_Win	38
/ Trues for the NDT T interface t/	39
<pre>/* Types for the MPI_T interface */</pre>	40
MPI_T_enum	41
MPI_T_cvar_handle	42
MPI_T_pvar_handle	43
MPI_T_pvar_session	44
MPI_T_event_instance	45
MPI_T_event_registration	46
MPI_T_source_order	47
MPI_T_cb_safety	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)
12
     TYPE(MPI_File)
13
     TYPE(MPI_Group)
14
     TYPE(MPI_Info)
15
     TYPE(MPI_Message)
16
     TYPE(MPI_Op)
17
     TYPE(MPI_Request)
18
     TYPE(MPI_Session)
19
     TYPE(MPI_Win)
20
21
     A.1.3 Prototype Definitions
22
23
     C Bindings
24
     The following are defined C typedefs for user-defined functions, also included in the file
25
     mpi.h.
26
27
     /* prototypes for user-defined functions */
28
     typedef void MPI_User_function(void *invec, void *inoutvec, int *len,
29
                   MPI_Datatype *datatype);
30
^{31}
     typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm, int comm_keyval,
32
                   void *extra_state, void *attribute_val_in,
33
                   void *attribute_val_out, int *flag);
34
     typedef int MPI_Comm_delete_attr_function(MPI_Comm comm, int comm_keyval,
35
                   void *attribute_val, void *extra_state);
36
37
     typedef int MPI_Win_copy_attr_function(MPI_Win oldwin, int win_keyval,
38
                   void *extra_state, void *attribute_val_in,
39
                   void *attribute_val_out, int *flag);
40
41
     typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval,
42
                   void *attribute_val, void *extra_state);
43
     typedef int MPI_Type_copy_attr_function(MPI_Datatype oldtype,
44
                   int type_keyval, void *extra_state, void *attribute_val_in,
45
                   void *attribute_val_out, int *flag);
46
47
     typedef int MPI_Type_delete_attr_function(MPI_Datatype datatype,
48
                   int type_keyval, void *attribute_val, void *extra_state);
```

typedef	<pre>void MPI_Comm_errhandler_function(MPI_Comm *comm, int *error_code,</pre>	1 2
typedef	<pre>void MPI_Win_errhandler_function(MPI_Win *win, int *error_code,);</pre>	3 4 5
typedef	<pre>void MPI_File_errhandler_function(MPI_File *file, int *error_code,);</pre>	6 7 8
typedef	<pre>void MPI_Session_errhandler_function(MPI_Session *session,</pre>	9 10
typedef	<pre>int MPI_Grequest_query_function(void *extra_state,</pre>	11 12 13
typedef	<pre>int MPI_Grequest_free_function(void *extra_state);</pre>	14
typedef	<pre>int MPI_Grequest_cancel_function(void *extra_state, int complete);</pre>	15 16
typedef	<pre>int MPI_Datarep_extent_function(MPI_Datatype datatype,</pre>	17 18
typedef	<pre>int MPI_Datarep_conversion_function(void *userbuf, MPI_Datatype datatype, int count, void *filebuf, MPI_Offset position, void *extra_state);</pre>	19 20 21 22
typedef	<pre>void MPI_T_event_cb_function(MPI_T_event_instance event_instance, MPI_T_event_registration event_registration,</pre>	23 24 25 26
	<pre>MPI_T_cb_safety cb_safety, void *user_data);</pre>	27 28 29 30
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>	31 32 33 34 35
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>	36 37 38 39 40
Fortran 2	008 Bindings with the mpi_f08 Module	41 42
The	back prototypes when using the Fortran mpi_f08 module are shown below: user-function argument to MPI_Op_create should be declared according to: INTERFACE	43 44 45
USE	JTINE MPI_User_function(invec, inoutvec, len, datatype) , INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR E(C_PTR), VALUE :: invec, inoutvec	46 47 48
1111	$\sum_{i=1}^{n} \sum_{i=1}^{n} \sum_{i$	-10

```
1
         INTEGER :: len
2
         TYPE(MPI_Datatype) :: datatype
3
         The copy and delete function arguments to MPI_Comm_create_keyval should be de-
4
     clared according to:
5
     ABSTRACT INTERFACE
6
       SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,
7
                    attribute_val_in, attribute_val_out, flag, ierror)
8
         TYPE(MPI_Comm) :: oldcomm
9
         INTEGER :: comm_keyval, ierror
10
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
11
                    attribute_val_out
12
         LOGICAL :: flag
13
14
     ABSTRACT INTERFACE
15
       SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval,
16
                    attribute_val, extra_state, ierror)
17
         TYPE(MPI_Comm) :: comm
18
         INTEGER :: comm_keyval, ierror
19
         INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
20
         The copy and delete function arguments to MPI_Win_create_keyval should be declared
21
     according to:
22
     ABSTRACT INTERFACE
23
       SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state,
24
                    attribute_val_in, attribute_val_out, flag, ierror)
25
         TYPE(MPI_Win) :: oldwin
26
         INTEGER :: win_keyval, ierror
27
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
28
                    attribute_val_out
29
         LOGICAL :: flag
30
31
     ABSTRACT INTERFACE
32
       SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,
33
                    extra_state, ierror)
34
        TYPE(MPI_Win) :: win
35
         INTEGER :: win_keyval, ierror
36
         INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
37
         The copy and delete function arguments to MPI_Type_create_keyval should be declared
38
     according to:
39
     ABSTRACT INTERFACE
40
       SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state,
41
                    attribute_val_in, attribute_val_out, flag, ierror)
42
         TYPE(MPI_Datatype) :: oldtype
43
         INTEGER :: type_keyval, ierror
44
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
45
                    attribute_val_out
46
         LOGICAL :: flag
47
48
```

```
ABSTRACT INTERFACE
                                                                                      1
                                                                                      2
  SUBROUTINE MPI_Type_delete_attr_function(datatype, type_keyval,
               attribute_val, extra_state, ierror)
    TYPE(MPI_Datatype) :: datatype
    INTEGER :: type_keyval, ierror
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
    The handler-function argument to MPI_Comm_create_errhandler should be declared
like this:
                                                                                      a
ABSTRACT INTERFACE
                                                                                      10
  SUBROUTINE MPI_Comm_errhandler_function(comm, error_code)
                                                                                      11
    TYPE(MPI_Comm) :: comm
                                                                                      12
    INTEGER :: error_code
                                                                                      13
                                                                                      14
    The handler-function argument to MPI_Win_create_errhandler should be declared like
                                                                                      15
this:
                                                                                      16
ABSTRACT INTERFACE
                                                                                      17
  SUBROUTINE MPI_Win_errhandler_function(win, error_code)
                                                                                      18
    TYPE(MPI_Win) :: win
                                                                                      19
    INTEGER :: error_code
                                                                                      20
    The handler-function argument to MPI_File_create_errhandler should be declared like
                                                                                      21
this:
                                                                                      22
ABSTRACT INTERFACE
                                                                                      23
  SUBROUTINE MPI_File_errhandler_function(file, error_code)
                                                                                      24
    TYPE(MPI_File) :: file
                                                                                      25
    INTEGER :: error_code
                                                                                      26
                                                                                      27
ABSTRACT INTERFACE
                                                                                      28
  SUBROUTINE MPI_File_errhandler_function(file, error_code)
                                                                                      29
    TYPE(MPI_File) :: file
                                                                                      30
    INTEGER :: error_code
                                                                                      31
    The handler-function argument to MPI_Session_create_errhandler should be declared
                                                                                      32
like this:
                                                                                      33
ABSTRACT INTERFACE
                                                                                      34
  SUBROUTINE MPI_Session_errhandler_function(session, error_code)
                                                                                      35
    TYPE(MPI_Session) :: session
                                                                                      36
    INTEGER :: error_code
                                                                                      37
                                                                                      38
ABSTRACT INTERFACE
                                                                                      39
  SUBROUTINE MPI_Session_errhandler_function(session, error_code)
                                                                                      40
    TYPE(MPI_Session) :: session
                                                                                      41
    INTEGER :: error_code
                                                                                      42
    The query, free, and cancel function arguments to MPI_Grequest_start should be de-
                                                                                      43
clared according to:
                                                                                      44
ABSTRACT INTERFACE
                                                                                      45
  SUBROUTINE MPI_Grequest_query_function(extra_state, status, ierror)
                                                                                      46
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
                                                                                      47
    TYPE(MPI_Status) :: status
                                                                                      48
```

```
1
         INTEGER :: ierror
\mathbf{2}
     ABSTRACT INTERFACE
3
       SUBROUTINE MPI_Grequest_free_function(extra_state, ierror)
4
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
5
         INTEGER :: ierror
6
7
     ABSTRACT INTERFACE
8
       SUBROUTINE MPI_Grequest_cancel_function(extra_state, complete, ierror)
9
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
10
         LOGICAL :: complete
11
         INTEGER :: ierror
12
         The extent and conversion function arguments to MPI_Register_datarep should be de-
13
     clared according to:
14
     ABSTRACT INTERFACE
15
       SUBROUTINE MPI_Datarep_extent_function(datatype, extent, extra_state,
16
                    ierror)
17
         TYPE(MPI_Datatype) :: datatype
18
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extent, extra_state
19
         INTEGER :: ierror
20
21
     ABSTRACT INTERFACE
22
       SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count,
23
                    filebuf, position, extra_state, ierror)
24
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
25
         TYPE(C_PTR), VALUE :: userbuf, filebuf
26
         TYPE(MPI_Datatype) :: datatype
27
         INTEGER :: count, ierror
28
         INTEGER(KIND=MPI_OFFSET_KIND) :: position
29
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
30
^{31}
     Fortran Bindings with mpif.h or the mpi Module
32
33
     With the Fortran mpi module or mpif.h, here are examples of how each of the user-defined
34
     subroutines should be declared.
35
         The user-function argument to MPI_OP_CREATE should be declared like this:
36
                    SUBROUTINE USER_FUNCTION (INVEC, INOUTVEC, LEN, DATATYPE)
37
         <type> INVEC(LEN), INOUTVEC(LEN)
38
         INTEGER LEN, DATATYPE
39
         The copy and delete function arguments to MPI_COMM_CREATE_KEYVAL should be
40
     declared like these:
41
     SUBROUTINE COMM_COPY_ATTR_FUNCTION(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
42
                   ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
43
         INTEGER OLDCOMM, COMM_KEYVAL, IERROR
44
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
45
                    ATTRIBUTE_VAL_OUT
46
47
         LOGICAL FLAG
48
```

SUBROUTINE COMM_DELETE_ATTR_FUNCTION(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,	1
EXTRA_STATE, IERROR) INTEGER COMM, COMM_KEYVAL, IERROR	2 3
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	4
	5
The copy and delete function arguments to MPI_WIN_CREATE_KEYVAL should be declared like these:	6
SUBROUTINE WIN_COPY_ATTR_FUNCTION(OLDWIN, WIN_KEYVAL, EXTRA_STATE,	7
ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)	8 9
INTEGER OLDWIN, WIN_KEYVAL, IERROR	9 10
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	11
ATTRIBUTE_VAL_OUT	12
LOGICAL FLAG	13
SUBROUTINE WIN_DELETE_ATTR_FUNCTION(WIN, WIN_KEYVAL, ATTRIBUTE_VAL,	14
EXTRA_STATE, IERROR)	15
INTEGER WIN, WIN_KEYVAL, IERROR	16 17
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	18
The delete function argument to $MPI_SESSION_CREATE_KEYVAL$ should be declared	19
like this:	20
SUBROUTINE SESSION_DELETE_ATTR_FUNCTION(SESSION, SESSION_KEYVAL, ATTRIBUTE_VAL	21
EXTRA_STATE, IERROR)	
INTEGER SESSION, SESSION_KEYVAL, IERROR	23 24
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	25
The converse delete function enguments to MDL TVDE CREATE KEV/AL should be	26
The copy and delete function arguments to MPI_TYPE_CREATE_KEYVAL should be declared like these:	27
SUBROUTINE TYPE_COPY_ATTR_FUNCTION(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE,	28
ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)	29
INTEGER OLDTYPE, TYPE_KEYVAL, IERROR	$30 \\ 31$
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	32
ATTRIBUTE_VAL_OUT	33
LOGICAL FLAG	34
SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,	35
EXTRA_STATE, IERROR)	36
INTEGER DATATYPE, TYPE_KEYVAL, IERROR	37 38
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	39
The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be de-	40
clared like this:	41
SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE) INTEGER COMM, ERROR_CODE	42
, _	43
The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be de-	44
clared like this: SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)	45 46
INTEGER WIN, ERROR_CODE	47
	48

1	The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de-
2 3	clared like this:
4	SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)
5	INTEGER FILE, ERROR_CODE
6	The handler-function argument to MPI_SESSION_CREATE_ERRHANDLER should be
7	declared like this:
8	SUBROUTINE SESSION_ERRHANDLER_FUNCTION(SESSION, ERROR_CODE)
9	INTEGER SESSION, ERROR_CODE
9 10	
11	The query, free, and cancel function arguments to MPI_GREQUEST_START should be
12	declared like these:
13	SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)
14	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
15	INTEGER STATUS(MPI_STATUS_SIZE), IERROR
16	SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)
17	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
18	INTEGER IERROR
19	SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)
20	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
21	LOGICAL COMPLETE
22	INTEGER IERROR
23	
24	The extent and conversion function arguments to MPI_REGISTER_DATAREP should
25	be declared like these:
26	SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)
27	INTEGER DATATYPE, IERROR
28	INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE
29	SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF,
30	POSITION, EXTRA_STATE, IERROR)
31 32	<type> USERBUF(*), FILEBUF(*)</type>
33	INTEGER DATATYPE, COUNT, IERROR
34	INTEGER(KIND=MPI_OFFSET_KIND) POSITION
35	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
36	
37	A.1.4 Deprecated Prototype Definitions
38	
39	The following are defined C typedefs for deprecated user-defined functions, also included in
40	the file mpi.h.
41	<pre>/* prototypes for user-defined functions */</pre>
42	/ prototypes for user defined functions /
43	typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval,
44	void *extra_state, void *attribute_val_in,
45	<pre>void *attribute_val_out, int *flag);</pre>
46	typedef int MPI Delete function (MPI Comm comm int koyya)
47	<pre>typedef int MPI_Delete_function(MPI_Comm comm, int keyval,</pre>
48	VOIU "AUGIIDAUG_VAI, VOIU "EXUIA_BUAUE/,

1 The following are deprecated Fortran user-defined callback subroutine prototypes. The $\mathbf{2}$ deprecated copy and delete function arguments to MPI_KEYVAL_CREATE should be de-3 clared like these: 4 SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR) 5INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, 6 ATTRIBUTE_VAL_OUT, IERR 7 LOGICAL FLAG 9 SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR) 10 INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR 11 1213 A.1.5 Info Keys 14The following info keys are reserved. They are strings. 1516access_style 17 accumulate_ops 18 accumulate_ordering 19alloc_shared_noncontig 20appnum arch 21cb_block_size 22 cb_buffer_size 23 24 cb_nodes 25chunked_item 26chunked_size chunked 27collective_buffering 28 29file 30 file_perm 31 filename host 32 33 io_node_list 34 ip_address ip_port 35mpi_assert_allow_overtaking 36 37 mpi_assert_exact_length mpi_assert_no_any_source 38 39 mpi_assert_no_any_tag mpi_assert_strict_start_ordering 40 41 mpi_hw_resource_type 42mpi_initial_errhandler mpi_optimization_goal 43 44mpi_reuse_count mpi_minimum_memory_alignment 4546nb_proc 47no_locks 48 num_io_nodes

```
1
       path
\mathbf{2}
       same_disp_unit
3
       same_size
4
       soft
\mathbf{5}
       striping_factor
6
       striping_unit
\overline{7}
       wdir
8
9
10
       A.1.6 Info Values
11
       The following info values are reserved. They are strings.
12
13
       false
       mpi_errors_abort
14
15
       mpi_errors_are_fatal
16
       mpi_errors_return
17
       mpi_shared_memory
18
       random
19
       rar
20
       raw
21
       read_mostly
22
       read_once
^{23}
       reverse_sequential
^{24}
       same_op
25
       same_op_no_op
26
       sequential
27
       true
28
       war
29
       waw
30
       write_mostly
       write_once
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

A.2 C Dinangs	1
	2
C binding	4
<pre>int MPI_Bsend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>	5 6
<pre>int MPI_Bsend_init(const void *buf, int count, MPI_Datatype datatype,</pre>	7 8 9
int MPI Buffer attach(void *buffer, int size)	10 11
	12
int MPI Cancel(MPI Request *request)	$13 \\ 14$
<pre>int MPI_Get_count(const MPI_Status *status, MPI_Datatype datatype,</pre>	14 15 16 17
<pre>int MPI_Ibsend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>	18 19
MPI_Message *message, MPI_Status *status)	20 21
<pre>int MPI_Imrecv(void *buf, int count, MPI_Datatype datatype,</pre>	22 23 24
MPI_Status *status)	25 26 27
<pre>int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source,</pre>	27 28 29
<pre>int MPI_Irsend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>	30 31 32
<pre>int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>	33 34
<pre>int MP1_Isendrecv(const void *sendbuf, int sendcount,</pre>	35 36 37 38 39
<pre>int MPI_Isendrecv_replace(void *buf, int count, MPI_Datatype datatype,</pre>	40 41 42
<pre>int MPI_Issend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>	43 44 45
int MPI_Mprobe(int source, int tag, MPI_Comm comm, MPI_Message *message, MPI_Status *status)	46 47 48

1 int MPI_Mrecv(void *buf, int count, MPI_Datatype datatype, $\mathbf{2}$ MPI_Message *message, MPI_Status *status) 3 int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status) 4 $\mathbf{5}$ int MPI_Recv_init(void *buf, int count, MPI_Datatype datatype, int source, 6 int tag, MPI_Comm comm, MPI_Request *request) 7 int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, 8 int tag, MPI_Comm comm, MPI_Status *status) 9 10int MPI_Request_free(MPI_Request *request) 11 int MPI_Request_get_status(MPI_Request request, int *flag, 12MPI_Status *status) 13 14int MPI_Rsend(const void *buf, int count, MPI_Datatype datatype, int dest, 15int tag, MPI_Comm comm) 16int MPI_Rsend_init(const void *buf, int count, MPI_Datatype datatype, 17int dest, int tag, MPI_Comm comm, MPI_Request *request) 18 19int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, 20int tag, MPI_Comm comm) 21int MPI_Send_init(const void *buf, int count, MPI_Datatype datatype, 22int dest, int tag, MPI_Comm comm, MPI_Request *request) 23 24 int MPI_Sendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, 25int dest, int sendtag, void *recvbuf, int recvcount, 26MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, 27MPI_Status *status) 28 int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype, 29 int dest, int sendtag, int source, int recvtag, MPI_Comm comm, 30 MPI_Status *status) 31 32 int MPI_Ssend(const void *buf, int count, MPI_Datatype datatype, int dest, 33 int tag, MPI_Comm comm) 34int MPI_Ssend_init(const void *buf, int count, MPI_Datatype datatype, 35int dest, int tag, MPI_Comm comm, MPI_Request *request) 36 37 int MPI_Startall(int count, MPI_Request array_of_requests[]) 38 39int MPI_Start(MPI_Request *request) 40int MPI_Testall(int count, MPI_Request array_of_requests[], int *flag, 41 MPI_Status array_of_statuses[]) 4243int MPI_Testany(int count, MPI_Request array_of_requests[], int *index, 44int *flag, MPI_Status *status) 45int MPI_Test_cancelled(const MPI_Status *status, int *flag) 4647int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status) 48

<pre>int MPI_Testsome(int incount, MPI_Request array_of_requests[],</pre>	1
MPI_Status array_of_statuses[])	3
<pre>int MPI_Waitall(int count, MPI_Request array_of_requests[],</pre>	4
MPI_Status array_of_statuses[])	5
<pre>int MPI_Waitany(int count, MPI_Request array_of_requests[], int ></pre>	*index, ⁷
MPI_Status *status)	8
<pre>int MPI_Wait(MPI_Request *request, MPI_Status *status)</pre>	9 10
<pre>int MPI_Waitsome(int incount, MPI_Request array_of_requests[],</pre>	11
<pre>int *outcount, int array_of_indices[],</pre>	12
<pre>MPI_Status array_of_statuses[])</pre>	14
	15
A.2.2 Datatypes C Bindings	16
<pre>int MPI_Get_address(const void *location, MPI_Aint *address)</pre>	17 18
<pre>int MPI_Pack(const void *inbuf, int incount, MPI_Datatype datatyp</pre>	
void *outbuf, int outsize, int *position, MPI_Comm	comm) 20
<pre>int MPI_Pack_external(const char datarep[], const void *inbuf, in</pre>	nt incount.
MPI_Datatype datatype, void *outbuf, MPI_Aint outsi	- 22
MPI_Aint *position)	23
<pre>int MPI_Pack_external_size(const char datarep[], int incount,</pre>	25
MPI_Datatype datatype, MPI_Aint *size)	26
int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm co	27 Omm,
int *size)	28 29
<pre>int MPI_Type_commit(MPI_Datatype *datatype)</pre>	30
	31
<pre>int MPI_Type_contiguous(int count, MPI_Datatype oldtype,</pre>	32
	33 34
<pre>int MPI_Type_create_darray(int size, int rank, int ndims,</pre>	
const int array_of_dargs[], const int array_of_psiz	
int order, MPI_Datatype oldtype, MPI_Datatype *newt	ype) 37
<pre>int MPI_Type_create_hindexed_block(int count, int blocklength,</pre>	38
const MPI_Aint array_of_displacements[], MPI_Dataty	pe oldtype, 40
MPI_Datatype *newtype)	41
<pre>int MPI_Type_create_hindexed(int count, const int array_of_block)</pre>	lengths[]. 42
const MPI_Aint array_of_displacements[], MPI_Dataty	-
MPI_Datatype *newtype)	44
<pre>int MPI_Type_create_hvector(int count, int blocklength, MPI_Aint</pre>	45 stride, 46
MPI_Datatype oldtype, MPI_Datatype *newtype)	40 47
	48

1 2 3	int	<pre>MPI_Type_create_indexed_block(int count, int blocklength,</pre>
4 5 6	int	<pre>MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint extent, MPI_Datatype *newtype)</pre>
7 8 9 10	int	<pre>MPI_Type_create_struct(int count, const int array_of_blocklengths[],</pre>
11 12 13	int	<pre>MPI_Type_create_subarray(int ndims, const int array_of_sizes[],</pre>
14 15	int	MPI_Type_dup(MPI_Datatype oldtype, MPI_Datatype *newtype)
16	int	MPI_Type_free(MPI_Datatype *datatype)
17 18 19 20 21	int	<pre>MPI_Type_get_contents(MPI_Datatype datatype, int max_integers,</pre>
22 23	int	<pre>MPI_Type_get_elements(MPI_Status *status, MPI_Datatype datatype,</pre>
24 25 26	int	<pre>MPI_Type_get_elements_x(MPI_Status *status, MPI_Datatype datatype,</pre>
27 28	int	<pre>MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers,</pre>
29 30 31	int	<pre>MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb,</pre>
32 33	int	<pre>MPI_Type_get_extent_x(MPI_Datatype datatype, MPI_Count *lb,</pre>
34 35 36	int	<pre>MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb, MPI_Aint *true_extent)</pre>
37 38	int	<pre>MPI_Type_get_true_extent_x(MPI_Datatype datatype, MPI_Count *true_lb,</pre>
 39 40 41 42 	int	<pre>MPI_Type_indexed(int count, const int array_of_blocklengths[],</pre>
43	int	MPI_Type_size(MPI_Datatype datatype, int *size)
44 45	int	<pre>MPI_Type_size_x(MPI_Datatype datatype, MPI_Count *size)</pre>
46 47 48	int	<pre>MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)</pre>

int	<pre>MPI_Unpack(const void *inbuf, int insize, int *position, void *outbuf,</pre>	1 2
int	<pre>MPI_Unpack_external(const char datarep[], const void *inbuf,</pre>	3
	int outcount, MPI_Datatype datatype)	5 6
MPI_	_Aint MPI_Aint_add(MPI_Aint base, MPI_Aint disp)	7
MPI_	_Aint MPI_Aint_diff(MPI_Aint addr1, MPI_Aint addr2)	8 9
		10
A.2.	3 Collective Communication C Bindings	11
int	MPI_Allgather(const void *sendbuf, int sendcount,	12 13
	MPI_Datatype sendtype, void *recvbuf, int recvcount,	14
	MPI_Datatype recvtype, MPI_Comm comm)	15
int	MPI_Allgather_init(const void *sendbuf, int sendcount,	16
	MPI_Datatype sendtype, void *recvbuf, int recvcount,	17
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,	18 19
	MPI_Request *request)	20
int	MPI_Allgatherv(const void *sendbuf, int sendcount,	21
	MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],	22
	<pre>const int displs[], MPI_Datatype recvtype, MPI_Comm comm)</pre>	23
int	MPI_Allgatherv_init(const void *sendbuf, int sendcount,	24
1110	MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],	25
	const int displs[], MPI_Datatype recvtype, MPI_Comm comm,	26
	MPI_Info info, MPI_Request *request)	27 28
int	MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,	29
	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)	30
int	MPI_Allreduce_init(const void *sendbuf, void *recvbuf, int count,	31
1110	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,	32
	MPI_Info info, MPI_Request *request)	33
		34 35
int	<pre>MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype,</pre>	36
	MPI_Comm comm)	37
		38
int	MPI_Alltoall_init(const void *sendbuf, int sendcount,	39
	MPI_Datatype sendtype, void *recvbuf, int recvcount,	40
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, MPI_Request *request)	41
	Mri_Mequest *request)	42
int	<pre>MPI_Alltoallv(const void *sendbuf, const int sendcounts[],</pre>	43
	<pre>const int sdispls[], MPI_Datatype sendtype, void *recvbuf,</pre>	44 45
	<pre>const int recvcounts[], const int rdispls[], MDL Deteture recording MDL Comm comm)</pre>	45 46
	MPI_Datatype recvtype, MPI_Comm comm)	40
int	<pre>MPI_Alltoallv_init(const void *sendbuf, const int sendcounts[],</pre>	48

```
1
                   const int sdispls[], MPI_Datatype sendtype, void *recvbuf,
\mathbf{2}
                   const int recvcounts[], const int rdispls[],
3
                   MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
4
                   MPI_Request *request)
5
     int MPI_Alltoallw(const void *sendbuf, const int sendcounts[],
6
                   const int sdispls[], const MPI_Datatype sendtypes[],
7
                   void *recvbuf, const int recvcounts[], const int rdispls[],
8
                   const MPI_Datatype recvtypes[], MPI_Comm comm)
9
10
     int MPI_Alltoallw_init(const void *sendbuf, const int sendcounts[],
11
                   const int sdispls[], const MPI_Datatype sendtypes[],
12
                   void *recvbuf, const int recvcounts[], const int rdispls[],
13
                   const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info,
14
                   MPI_Request *request)
15
     int MPI_Barrier_init(MPI_Comm comm, MPI_Info info, MPI_Request *request)
16
17
     int MPI_Barrier(MPI_Comm comm)
18
     int MPI_Bcast_init(void *buffer, int count, MPI_Datatype datatype,
19
                   int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)
20
21
     int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root,
22
                   MPI_Comm comm)
23
     int MPI_Exscan(const void *sendbuf, void *recvbuf, int count,
^{24}
                   MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
25
26
     int MPI_Exscan_init(const void *sendbuf, void *recvbuf, int count,
27
                   MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
28
                   MPI_Info info, MPI_Request *request)
29
     int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
30
                  void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
^{31}
                   MPI_Comm comm)
32
33
     int MPI_Gather_init(const void *sendbuf, int sendcount,
34
                   MPI_Datatype sendtype, void *recvbuf, int recvcount,
35
                   MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,
36
                   MPI_Request *request)
37
     int MPI_Gatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
38
                   void *recvbuf, const int recvcounts[], const int displs[],
39
                   MPI_Datatype recvtype, int root, MPI_Comm comm)
40
41
     int MPI_Gatherv_init(const void *sendbuf, int sendcount,
42
                   MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
43
                   const int displs[], MPI_Datatype recvtype, int root,
44
                   MPI_Comm comm, MPI_Info info, MPI_Request *request)
45
     int MPI_Iallgather(const void *sendbuf, int sendcount,
46
47
                   MPI_Datatype sendtype, void *recvbuf, int recvcount,
48
                   MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
```

ANNEX A. LANGUAGE BINDINGS SUMMARY

800

int	MPI_Iallgatherv(const void *sendbuf, int sendcount,	1
	<pre>MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],</pre>	2
	<pre>const int displs[], MPI_Datatype recvtype, MPI_Comm comm,</pre>	3
	MPI_Request *request)	4 5
int	MPI_Iallreduce(const void *sendbuf, void *recvbuf, int count,	6
	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,	7
	MPI_Request *request)	8
int	MPI_Ialltoall(const void *sendbuf, int sendcount,	9
	MPI_Datatype sendtype, void *recvbuf, int recvcount,	10
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)	11
÷	MDT Talltaalla (senst weid weendbuf senst int sendount []	12
int	<pre>MPI_Ialltoallv(const void *sendbuf, const int sendcounts[],</pre>	13
	const int recvcounts[], const int rdispls[],	14
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)	15
		16 17
int	MPI_Ialltoallw(const void *sendbuf, const int sendcounts[],	18
	<pre>const int sdispls[], const MPI_Datatype sendtypes[], weid meanwhif exact int meanwhif[]</pre>	19
	<pre>void *recvbuf, const int recvcounts[], const int rdispls[], const MPI_Datatype recvtypes[], MPI_Comm comm,</pre>	20
	MPI_Request *request)	21
		22
int	<pre>MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)</pre>	23
int	MPI_Ibcast(void *buffer, int count, MPI_Datatype datatype, int root,	24
	MPI_Comm comm, MPI_Request *request)	25
int	MPI_Iexscan(const void *sendbuf, void *recvbuf, int count,	26 27
	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,	28
	MPI_Request *request)	29
÷		30
int	<pre>MPI_Igather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,</pre>	31
	MPI_Comm comm, MPI_Request *request)	32
		33
int	MPI_Igatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,	34
	<pre>void *recvbuf, const int recvcounts[], const int displs[], </pre>	35
	MPI_Datatype recvtype, int root, MPI_Comm comm,	36
	MPI_Request *request)	37
int	MPI_Ireduce(const void *sendbuf, void *recvbuf, int count,	38 39
	MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,	40
	MPI_Request *request)	41
int	MPI_Ireduce_scatter_block(const void *sendbuf, void *recvbuf,	42
	int recvcount, MPI_Datatype datatype, MPI_Op op,	43
	MPI_Comm comm, MPI_Request *request)	44
int	MPI_Ireduce_scatter(const void *sendbuf, void *recvbuf,	45
1110	const int recvcounts[], MPI_Datatype datatype, MPI_Op op,	46
	MPI_Comm comm, MPI_Request *request)	47
		48

1 2 3	int	<pre>MPI_Iscan(const void *sendbuf, void *recvbuf, int count,</pre>
4 5 6 7	int	<pre>MPI_Iscatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)</pre>
8 9 10 11	int	<pre>MPI_Iscatterv(const void *sendbuf, const int sendcounts[],</pre>
12 13	int	MPI_Op_commutative(MPI_Op op, int *commute)
14 15	int	<pre>MPI_Op_create(MPI_User_function *user_fn, int commute, MPI_Op *op)</pre>
16	int	MPI_Op_free(MPI_Op *op)
17 18 19	int	<pre>MPI_Reduce(const void *sendbuf, void *recvbuf, int count,</pre>
20 21 22	int	<pre>MPI_Reduce_init(const void *sendbuf, void *recvbuf, int count,</pre>
23 24 25	int	<pre>MPI_Reduce_local(const void *inbuf, void *inoutbuf, int count, MPI_Datatype datatype, MPI_Op op)</pre>
26 27 28	int	<pre>MPI_Reduce_scatter_block(const void *sendbuf, void *recvbuf,</pre>
29 30 31 32	int	<pre>MPI_Reduce_scatter_block_init(const void *sendbuf, void *recvbuf,</pre>
33 34 35	int	<pre>MPI_Reduce_scatter(const void *sendbuf, void *recvbuf,</pre>
36 37 38 39	int	<pre>MPI_Reduce_scatter_init(const void *sendbuf, void *recvbuf,</pre>
40 41	int	<pre>MPI_Scan(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)</pre>
42 43 44 45	int	<pre>MPI_Scan_init(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request)</pre>
46 47 48	int	<pre>MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)</pre>

<pre>int MPI_Scatter_init(const void *sendbuf, int sendcount,</pre>	1
<pre>MPI_Datatype sendtype, void *recvbuf, int recvcount,</pre>	2
MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info in	fo, ³
MPI_Request *request)	4
int MDI Constrong const word toondbuf const int condecunts[]	5
int MPI_Scatterv(const void *sendbuf, const int sendcounts[],	6
<pre>const int displs[], MPI_Datatype sendtype, void *recvbuf, int recveount MDI Datatype recutives int reat MDI Comm co</pre>	7 mm)
int recvcount, MPI_Datatype recvtype, int root, MPI_Comm co	шш) ₈
<pre>int MPI_Scatterv_init(const void *sendbuf, const int sendcounts[],</pre>	9
<pre>const int displs[], MPI_Datatype sendtype, void *recvbuf,</pre>	10
int recvcount, MPI_Datatype recvtype, int root, MPI_Comm co	mm, ¹¹
MPI_Info info, MPI_Request *request)	12
	13
A.2.4 Groups, Contexts, Communicators, and Caching C Bindings	14
A.2.4 Groups, Contexts, Communicators, and Cacining C bindings	15
<pre>int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)</pre>	16 17
<pre>int MPI_Comm_create_from_group(MPI_Group group, const char *stringtag,</pre>	18
MPI_Info info, MPI_Errhandler errhandler, MPI_Comm *newcomm	
	20
<pre>int MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag,</pre>	21
MPI_Comm *newcomm)	22
int MPI_Comm_create_keyval(MPI_Comm_copy_attr_function *comm_copy_attr_fi	1, ²³
MPI_Comm_delete_attr_function *comm_delete_attr_fn,	24
int *comm_keyval, void *extra_state)	25
int NDT (Law and the (NDT (Law MDT (Law and the second sec	26
<pre>int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)</pre>	27
<pre>int MPI_Comm_delete_attr(MPI_Comm comm, int comm_keyval)</pre>	28
int MPI_COMM_DUP_FN(MPI_Comm oldcomm, int comm_keyval, void *extra_state	29
void *attribute_val_in, void *attribute_val_out, int *flag)	30
	31
<pre>int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)</pre>	32
int MPI_Comm_dup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm *newcom	nm) $^{33}_{34}$
	34
<pre>int MPI_Comm_free_keyval(int *comm_keyval)</pre>	36
<pre>int MPI_Comm_free(MPI_Comm *comm)</pre>	37
int MDI Comm not attm/MDI Comm comm int comm koursel usid tottmibute us	
<pre>int MPI_Comm_get_attr(MPI_Comm comm, int comm_keyval, void *attribute_val</pre>	L , 39
Int *ilag)	40
<pre>int MPI_Comm_get_info(MPI_Comm comm, MPI_Info *info_used)</pre>	41
<pre>int MPI_Comm_get_name(MPI_Comm comm, char *comm_name, int *resultlen)</pre>	42
int in i_comm_get_name(in i_comm comm, endi /comm_name, int /icbuitten/	43
<pre>int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)</pre>	44
int MPI_Comm_idup(MPI_Comm comm, MPI_Comm *newcomm, MPI_Request *request)	45
	46
int MPI_Comm_idup_with_info(MPI_Comm comm, MPI_Info info,	47
MPI_Comm *newcomm, MPI_Request *request)	48

1 2 3	int	<pre>MPI_COMM_NULL_COPY_FN(MPI_Comm oldcomm, int comm_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag)</pre>
4 5 6	int	<pre>MPI_COMM_NULL_DELETE_FN(MPI_Comm comm, int comm_keyval,</pre>
7	int	MPI_Comm_rank(MPI_Comm comm, int *rank)
8 9	int	MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
10	int	MPI_Comm_remote_size(MPI_Comm comm, int *size)
11 12	int	MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void *attribute_val)
13	int	MPI_Comm_set_info(MPI_Comm comm, MPI_Info info)
14 15	int	MPI_Comm_set_name(MPI_Comm comm, const char *comm_name)
16	int	MPI_Comm_size(MPI_Comm comm, int *size)
17 18	int	MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
19 20 21	int	<pre>MPI_Comm_split_type(MPI_Comm comm, int split_type, int key, MPI_Info info, MPI_Comm *newcomm)</pre>
22	int	MPI_Comm_test_inter(MPI_Comm comm, int *flag)
23 24	int	MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result)
25 26	int	<pre>MPI_Group_difference(MPI_Group group1, MPI_Group group2,</pre>
27 28 29	int	<pre>MPI_Group_excl(MPI_Group group, int n, const int ranks[], MPI_Group *newgroup)</pre>
30	int	MPI_Group_free(MPI_Group *group)
31 32 33	int	<pre>MPI_Group_from_session_pset(MPI_Session session, const char *pset_name,</pre>
34 35	int	<pre>MPI_Group_incl(MPI_Group group, int n, const int ranks[], MPI_Group *newgroup)</pre>
36 37 38	int	<pre>MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)</pre>
39 40	int	<pre>MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)</pre>
41 42 43	int	<pre>MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)</pre>
44 45	int	MPI_Group_rank(MPI_Group group, int *rank)
45 46	int	MPI_Group_size(MPI_Group group, int *size)
47 48		
-10		

int	<pre>MPI_Group_translate_ranks(MPI_Group group1, int n, const int ranks1[], MPI_Group group2, int ranks2[])</pre>	1 2
int	<pre>MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)</pre>	3 4 5
int	<pre>MPI_Intercomm_create_from_groups(MPI_Group local_group,</pre>	6 7 8 9 10
int	<pre>MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,</pre>	11 12 13
int	MPI_Intercomm_merge(MPI_Comm intercomm, int high, MPI_Comm *newintracomm)	14 15 16
int	<pre>MPI_Type_create_keyval(MPI_Type_copy_attr_function *type_copy_attr_fn,</pre>	17 18 19
int	MPI_Type_delete_attr(MPI_Datatype datatype, int type_keyval)	20 21
int	<pre>MPI_TYPE_DUP_FN(MPI_Datatype oldtype, int type_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag)</pre>	21 22 23 24
int	MPI_Type_free_keyval(int *type_keyval)	25
int	<pre>MPI_Type_get_attr(MPI_Datatype datatype, int type_keyval, void *attribute_val, int *flag)</pre>	26 27 28
int	<pre>MPI_Type_get_name(MPI_Datatype datatype, char *type_name,</pre>	29 30 31
int	<pre>MPI_TYPE_NULL_COPY_FN(MPI_Datatype oldtype, int type_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag)</pre>	32 33 34
int	<pre>MPI_TYPE_NULL_DELETE_FN(MPI_Datatype datatype, int type_keyval, void *attribute_val, void *extra_state)</pre>	35 36 37
int	<pre>MPI_Type_set_attr(MPI_Datatype datatype, int type_keyval, void *attribute_val)</pre>	38 39
int	MPI_Type_set_name(MPI_Datatype datatype, const char *type_name)	40 41
int	<pre>MPI_Win_create_keyval(MPI_Win_copy_attr_function *win_copy_attr_fn,</pre>	42 43 44
int	MPI_Win_delete_attr(MPI_Win win, int win_keyval)	45 46
int	<pre>MPI_WIN_DUP_FN(MPI_Win oldwin, int win_keyval, void *extra_state,</pre>	47 48

1	<pre>void *attribute_val_in, void *attribute_val_out, int *flag)</pre>
$\frac{2}{3}$	int MPI_Win_free_keyval(int *win_keyval)
4 5	<pre>int MPI_Win_get_attr(MPI_Win win, int win_keyval, void *attribute_val,</pre>
6 7	int MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)
8 9	<pre>int MPI_WIN_NULL_COPY_FN(MPI_Win oldwin, int win_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag)</pre>
10 11 12	<pre>int MPI_WIN_NULL_DELETE_FN(MPI_Win win, int win_keyval,</pre>
13	int MPI_Win_set_attr(MPI_Win win, int win_keyval, void *attribute_val)
14 15 16 17	<pre>int MPI_Win_set_name(MPI_Win win, const char *win_name)</pre>
18	A.2.5 Process Topologies C Bindings
19	<pre>int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])</pre>
20 21 22	<pre>int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[],</pre>
23	int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
24 25 26	<pre>int MPI_Cart_get(MPI_Comm comm, int maxdims, int dims[], int periods[],</pre>
27 28	<pre>int MPI_Cart_map(MPI_Comm comm, int ndims, const int dims[],</pre>
29 30	<pre>int MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank)</pre>
31 32	<pre>int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,</pre>
$\frac{33}{34}$	<pre>int MPI_Cart_sub(MPI_Comm comm, const int remain_dims[], MPI_Comm *newcomm)</pre>
35	<pre>int MPI_Dims_create(int nnodes, int ndims, int dims[])</pre>
36 37 38 39 40	<pre>int MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree,</pre>
41 42 43 44 45	<pre>int MPI_Dist_graph_create(MPI_Comm comm_old, int n, const int sources[],</pre>
46 47 48	<pre>int MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree,</pre>

int	<pre>MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[],</pre>	1 2 3
int	<pre>MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int index[],</pre>	4 5 6
int	MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)	7
int	<pre>MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int index[],</pre>	8 9 10
int	<pre>MPI_Graph_map(MPI_Comm comm, int nnodes, const int index[],</pre>	11 12
int	MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)	13 14
int	<pre>MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,</pre>	15 16 17
int	<pre>MPI_Ineighbor_allgather(const void *sendbuf, int sendcount,</pre>	18 19 20
int	<pre>MPI_Ineighbor_allgatherv(const void *sendbuf, int sendcount,</pre>	21 22 23 24 25
int	<pre>MPI_Ineighbor_alltoall(const void *sendbuf, int sendcount,</pre>	26 27 28
int	<pre>MPI_Ineighbor_alltoallv(const void *sendbuf, const int sendcounts[],</pre>	29 30 31 32 33
int	<pre>MPI_Ineighbor_alltoallw(const void *sendbuf, const int sendcounts[],</pre>	34 35 36 37 38
int	<pre>MPI_Neighbor_allgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)</pre>	39 40 41 42
int	<pre>MPI_Neighbor_allgather_init(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, MPI_Request *request)</pre>	43 44 45 46
int	MPI_Neighbor_allgatherv(const void *sendbuf, int sendcount,	47 48

1MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], 2 const int displs[], MPI_Datatype recvtype, MPI_Comm comm) 3 int MPI_Neighbor_allgatherv_init(const void *sendbuf, int sendcount, 4 MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], 5const int displs[], MPI_Datatype recvtype, MPI_Comm comm, 6 MPI_Info info, MPI_Request *request) 7 8 int MPI_Neighbor_alltoall(const void *sendbuf, int sendcount, 9 MPI_Datatype sendtype, void *recvbuf, int recvcount, 10 MPI_Datatype recvtype, MPI_Comm comm) 11 int MPI_Neighbor_alltoall_init(const void *sendbuf, int sendcount, 12MPI_Datatype sendtype, void *recvbuf, int recvcount, 13 MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, 14 MPI_Request *request) 15int MPI_Neighbor_alltoallv(const void *sendbuf, const int sendcounts[], 1617 const int sdispls[], MPI_Datatype sendtype, void *recvbuf, 18 const int recvcounts[], const int rdispls[], 19 MPI_Datatype recvtype, MPI_Comm comm) 20int MPI_Neighbor_alltoallv_init(const void *sendbuf, 21const int sendcounts[], const int sdispls[], 22 MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], 23const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm, 24MPI_Info info, MPI_Request *request) 2526int MPI_Neighbor_alltoallw(const void *sendbuf, const int sendcounts[], 27const MPI_Aint sdispls[], const MPI_Datatype sendtypes[], 28void *recvbuf, const int recvcounts[], 29 const MPI_Aint rdispls[], const MPI_Datatype recvtypes[], 30 MPI_Comm comm) 31int MPI_Neighbor_alltoallw_init(const void *sendbuf, 32 const int sendcounts[], const MPI_Aint sdispls[], 33 const MPI_Datatype sendtypes[], void *recvbuf, 34 const int recvcounts[], const MPI_Aint rdispls[], 35const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info, 36 MPI_Request *request) 37 38 int MPI_Topo_test(MPI_Comm comm, int *status) 39 4041 A.2.6 MPI Environmental Management C Bindings 42int MPI_Add_error_class(int *errorclass) 43 44int MPI_Add_error_code(int errorclass, int *errorcode) 45int MPI_Add_error_string(int errorcode, const char *string) 4647int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr) 48

ANNEX A. LANGUAGE BINDINGS SUMMARY

<pre>int MPI_Comm_call_errhandler(MPI_Comm comm, int errorcode)</pre>	1
int MPI_Comm_create_errhandler(MPI_Comm_errhandler_function	2 3
<pre>*comm_errhandler_fn, MPI_Errhandler *errhandler)</pre>	4
int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)	5
int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)	6 7
int MPI_Errhandler_free(MPI_Errhandler *errhandler)	8
	9
<pre>int MPI_Error_class(int errorcode, int *errorclass)</pre>	10
<pre>int MPI_Error_string(int errorcode, char *string, int *resultlen)</pre>	11 12
<pre>int MPI_File_call_errhandler(MPI_File fh, int errorcode)</pre>	13
int MPI_File_create_errhandler(MPI_File_errhandler_function	14
<pre>*file_errhandler_fn, MPI_Errhandler *errhandler)</pre>	15 16
int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)	17
int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)	18
<pre>int MPI_Free_mem(void *base)</pre>	19 20
	21
<pre>int MPI_Get_library_version(char *version, int *resultlen)</pre>	22
<pre>int MPI_Get_processor_name(char *name, int *resultlen)</pre>	23 24
<pre>int MPI_Get_version(int *version, int *subversion)</pre>	25
int MPI_Session_call_errhandler(MPI_Session session, int errorcode)	26
int MPI_Session_create_errhandler(MPI_Session_errhandler_function	27 28
<pre>*session_errhandler_fn, MPI_Errhandler *errhandler)</pre>	28
int MPI_Session_get_errhandler(MPI_Session session,	30
MPI_Errhandler *errhandler)	31
int MPI_Session_set_errhandler(MPI_Session session,	32 33
MPI_Errhandler errhandler)	34
int MPI_Win_call_errhandler(MPI_Win win, int errorcode)	35
int MPI_Win_create_errhandler(MPI_Win_errhandler_function	36
*win_errhandler_fn, MPI_Errhandler *errhandler)	37 38
int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)	39
	40
<pre>int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)</pre>	41
double MPI_Wtick(void)	42
double MPI_Wtime(void)	43 44
	45
A 2.7 The Info Object C Rindings	46
A.2.7 The Info Object C Bindings	47
<pre>int MPI_Info_create_env(int argc, char argv[], MPI_Info *info)</pre>	48

```
1
     int MPI_Info_create(MPI_Info *info)
\mathbf{2}
     int MPI_Info_delete(MPI_Info info, const char *key)
3
4
     int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo)
5
     int MPI_Info_free(MPI_Info *info)
6
\overline{7}
     int MPI_Info_get(MPI_Info info, const char *key, int valuelen, char *value,
8
                   int *flag)
9
     int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)
10
11
     int MPI_Info_get_nthkey(MPI_Info info, int n, char *key)
12
     int MPI_Info_get_string(MPI_Info info, const char *key, int *buflen,
13
                   char *value, int *flag)
14
15
     int MPI_Info_get_valuelen(MPI_Info info, const char *key, int *valuelen,
16
                   int *flag)
17
     int MPI_Info_set(MPI_Info info, const char *key, const char *value)
18
19
20
     A.2.8 Process Creation and Management C Bindings
21
22
     int MPI_Abort(MPI_Comm comm, int errorcode)
23
     int MPI_Close_port(const char *port_name)
24
25
     int MPI_Comm_accept(const char *port_name, MPI_Info info, int root,
26
                   MPI_Comm comm, MPI_Comm *newcomm)
27
     int MPI_Comm_connect(const char *port_name, MPI_Info info, int root,
28
                   MPI_Comm comm, MPI_Comm *newcomm)
29
30
     int MPI_Comm_disconnect(MPI_Comm *comm)
31
     int MPI_Comm_get_parent(MPI_Comm *parent)
32
33
     int MPI_Comm_join(int fd, MPI_Comm *intercomm)
34
     int MPI_Comm_spawn(const char *command, char *argv[], int maxprocs,
35
                   MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm,
36
                   int array_of_errcodes[])
37
38
     int MPI_Comm_spawn_multiple(int count, char *array_of_commands[],
39
                   char **array_of_argv[], const int array_of_maxprocs[],
40
                   const MPI_Info array_of_info[], int root, MPI_Comm comm,
41
                   MPI_Comm *intercomm, int array_of_errcodes[])
42
     int MPI_Finalized(int *flag)
43
44
     int MPI_Finalize(void)
45
     int MPI_Initialized(int *flag)
46
47
     int MPI_Init(int *argc, char ***argv)
48
```

int	<pre>MPI_Init_thread(int *argc, char ***argv, int required, int *provided)</pre>	1
int	MPI_Is_thread_main(int *flag)	2 3
int	MPI_Lookup_name(const char *service_name, MPI_Info info,	4
	char *port_name)	5
int	MPI_Open_port(MPI_Info info, char *port_name)	6 7
int	MPI_Publish_name(const char *service_name, MPI_Info info,	8
THC	const char *port_name)	9
int	MPI_Query_thread(int *provided)	10 11
int	MPI_Session_finalize(MPI_Session *session)	12
	MPI_Session_get_info(MPI_Session session, MPI_Info *info_used)	13
1110	In I_bession_get_into(in I_bession bession; in I_into *into_used)	14 15
int	<pre>MPI_Session_get_nth_pset(MPI_Session session, MPI_Info info, int n,</pre>	16 17
int	MPI_Session_get_num_psets(MPI_Session session, MPI_Info info,	18
	int *npset_names)	19
int	MPI_Session_get_pset_info(MPI_Session session, const char *pset_name,	20
	MPI_Info *info)	21
int	MPI_Session_init(MPI_Info info, MPI_Errhandler errhandler,	22 23
1110	MPI_Session *session)	24
int	MPI_Unpublish_name(const char *service_name, MPI_Info info,	25
THC	const char *port_name)	26
		27
• •		28
A.2.	9 One-Sided Communications C Bindings	29 30
int	MPI_Accumulate(const void *origin_addr, int origin_count,	31
	MPI_Datatype origin_datatype, int target_rank,	32
	MPI_Aint target_disp, int target_count,	33
	MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)	34
int	MPI_Compare_and_swap(const void *origin_addr, const void *compare_addr,	35
	<pre>void *result_addr, MPI_Datatype datatype, int target_rank,</pre>	36
	MPI_Aint target_disp, MPI_Win win)	37
int	MPI_Fetch_and_op(const void *origin_addr, void *result_addr,	38
1110	MPI_Datatype datatype, int target_rank, MPI_Aint target_disp,	39
	MPI_Op op, MPI_Win win)	40
		41 42
int	<pre>MPI_Get_accumulate(const void *origin_addr, int origin_count,</pre>	43
	MPI_Datatype origin_datatype, void *result_addr,	44
	<pre>int result_count, MPI_Datatype result_datatype, int toward work MDI Aight toward diag int toward count</pre>	45
	int target_rank, MPI_Aint target_disp, int target_count,	46
	MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)	47
		48

```
1
     int MPI_Get(void *origin_addr, int origin_count,
\mathbf{2}
                   MPI_Datatype origin_datatype, int target_rank,
3
                   MPI_Aint target_disp, int target_count,
4
                   MPI_Datatype target_datatype, MPI_Win win)
5
     int MPI_Put(const void *origin_addr, int origin_count,
6
                   MPI_Datatype origin_datatype, int target_rank,
7
                   MPI_Aint target_disp, int target_count,
8
                   MPI_Datatype target_datatype, MPI_Win win)
9
10
     int MPI_Raccumulate(const void *origin_addr, int origin_count,
11
                   MPI_Datatype origin_datatype, int target_rank,
12
                   MPI_Aint target_disp, int target_count,
13
                   MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
14
                   MPI_Request *request)
15
     int MPI_Rget_accumulate(const void *origin_addr, int origin_count,
16
                   MPI_Datatype origin_datatype, void *result_addr,
17
                   int result_count, MPI_Datatype result_datatype,
18
                   int target_rank, MPI_Aint target_disp, int target_count,
19
                   MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
20
                   MPI_Request *request)
21
22
     int MPI_Rget(void *origin_addr, int origin_count,
23
                   MPI_Datatype origin_datatype, int target_rank,
24
                   MPI_Aint target_disp, int target_count,
25
                   MPI_Datatype target_datatype, MPI_Win win,
26
                   MPI_Request *request)
27
     int MPI_Rput(const void *origin_addr, int origin_count,
28
                   MPI_Datatype origin_datatype, int target_rank,
29
                   MPI_Aint target_disp, int target_count,
30
                   MPI_Datatype target_datatype, MPI_Win win,
31
                   MPI_Request *request)
32
33
     int MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info,
34
                   MPI_Comm comm, void *baseptr, MPI_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
38
     int MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size)
39
     int MPI_Win_complete(MPI_Win win)
40
41
     int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win)
42
     int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info,
43
                   MPI_Comm comm, MPI_Win *win)
44
45
     int MPI_Win_detach(MPI_Win win, const void *base)
46
47
     int MPI_Win_fence(int assert, MPI_Win win)
48
```

1 int MPI_Win_flush_all(MPI_Win win) $\mathbf{2}$ int MPI_Win_flush(int rank, MPI_Win win) 3 int MPI_Win_flush_local_all(MPI_Win win) 5 int MPI_Win_flush_local(int rank, MPI_Win win) 6 int MPI_Win_free(MPI_Win *win) int MPI_Win_get_group(MPI_Win win, MPI_Group *group) 9 10 int MPI_Win_get_info(MPI_Win win, MPI_Info *info_used) 11 int MPI_Win_lock_all(int assert, MPI_Win win) 1213 int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win) 14int MPI_Win_post(MPI_Group group, int assert, MPI_Win win) 1516int MPI_Win_set_info(MPI_Win win, MPI_Info info) 17 int MPI_Win_shared_query(MPI_Win win, int rank, MPI_Aint *size, 18 int *disp_unit, void *baseptr) 19 20int MPI_Win_start(MPI_Group group, int assert, MPI_Win win) 21int MPI_Win_sync(MPI_Win win) 22 23int MPI_Win_test(MPI_Win win, int *flag) 2425int MPI_Win_unlock_all(MPI_Win win) 26int MPI_Win_unlock(int rank, MPI_Win win) 2728 int MPI_Win_wait(MPI_Win win) 29 30 A.2.10 External Interfaces C Bindings 31 32 int MPI_Grequest_complete(MPI_Request request) 33 int MPI_Grequest_start(MPI_Grequest_query_function *query_fn, 34 MPI_Grequest_free_function *free_fn, 35 MPI_Grequest_cancel_function *cancel_fn, void *extra_state, 36 MPI_Request *request) 37 38 int MPI_Status_set_cancelled(MPI_Status *status, int flag) 39 int MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype, 40 int count) 41 42int MPI_Status_set_elements_x(MPI_Status *status, MPI_Datatype datatype, 43 MPI_Count count) 44 454647

48

1	A.2.11 I/O C Bindings
2 3 4	<pre>int MPI_CONVERSION_FN_NULL(void *userbuf, MPI_Datatype datatype, int count, void *filebuf, MPI_Offset position, void *extra_state)</pre>
5	<pre>int MPI_File_close(MPI_File *fh)</pre>
6 7	<pre>int MPI_File_delete(const char *filename, MPI_Info info)</pre>
8	<pre>int MPI_File_get_amode(MPI_File fh, int *amode)</pre>
9 10	<pre>int MPI_File_get_atomicity(MPI_File fh, int *flag)</pre>
11 12 13	<pre>int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset,</pre>
14	<pre>int MPI_File_get_group(MPI_File fh, MPI_Group *group)</pre>
15 16	<pre>int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)</pre>
17	<pre>int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)</pre>
18 19	<pre>int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)</pre>
20	int MPI_File_get_size(MPI_File fh, MPI_Offset *size)
21 22 23	<pre>int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype,</pre>
24 25	<pre>int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype, MPI_Datatype *filetype, char *datarep)</pre>
26 27 28	int MPI_File_iread_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)
29 30	<pre>int MPI_File_iread_at_all(MPI_File fh, MPI_Offset offset, void *buf,</pre>
31 32 33	<pre>int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)</pre>
34 35	<pre>int MPI_File_iread(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)</pre>
36 37 38	int MPI_File_iread_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)
39 40	int MPI_File_iwrite_all(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Request *request)
41 42 43	<pre>int MPI_File_iwrite_at_all(MPI_File fh, MPI_Offset offset, const void *buf,</pre>
44 45 46	int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Request *request)
40 47 48	int MPI_File_iwrite(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Request *request)

int	<pre>MPI_File_iwrite_shared(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Request *request)</pre>	1 2 2
int	<pre>MPI_File_open(MPI_Comm comm, const char *filename, int amode, MPI_Info info, MPI_File *fh)</pre>	3 4 5
int	MPI_File_preallocate(MPI_File fh, MPI_Offset size)	6
int	<pre>MPI_File_read_all_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype)</pre>	7 8 9
int	MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status)	10
int	<pre>MPI_File_read_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>	11 12 13
int	<pre>MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,</pre>	14 15 16
int	MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)	17
int	<pre>MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,</pre>	18 19 20
int	<pre>MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>	21 22
int	<pre>MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>	23 24 25
int	<pre>MPI_File_read_ordered_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype)</pre>	26 27
int	<pre>MPI_File_read_ordered_end(MPI_File fh, void *buf, MPI_Status *status)</pre>	28 29
int	<pre>MPI_File_read_ordered(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>	30 31
int	MPI_File_read_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)	32 33 34
int	MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)	35
int	MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence)	36 37
int	MPI_File_set_atomicity(MPI_File fh, int flag)	38
int	MPI_File_set_info(MPI_File fh, MPI_Info info)	39 40
int	MPI_File_set_size(MPI_File fh, MPI_Offset size)	41
int	<pre>MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, const char *datarep, MPI_Info info)</pre>	42 43 44
int	MPI_File_sync(MPI_File fh)	45
int	MPI_File_write_all_begin(MPI_File fh, const void *buf, int count, MPI_Datatype datatype)	46 47 48

1 2	nt MPI_File_write_all_end(MPI_File fh, const void *buf, MPI_Status *status)	
3 4 5	nt MPI_File_write_all(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)	
6 7 8	nt MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype)	
9 10	nt MPI_File_write_at_all_end(MPI_File fh, const void *buf, MPI_Status *status)	
11 12 13	nt MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void * int count, MPI_Datatype datatype, MPI_Status *status)	buf,
14 15	nt MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)	
16 17 18	nt MPI_File_write(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)	
19 20	nt MPI_File_write_ordered_begin(MPI_File fh, const void *buf, int cou MPI_Datatype datatype)	nt,
21 22 23	nt MPI_File_write_ordered_end(MPI_File fh, const void *buf, MPI_Status *status)	
24 25	nt MPI_File_write_ordered(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)	
26 27 28	nt MPI_File_write_shared(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)	
29 30 31 32 33 34	<pre>nt MPI_Register_datarep(const char *datarep,</pre>	
35 36	.2.12 Language Bindings C Bindings	
37 38	nt MPI_Status_f082f(MPI_F08_status *f08_status, MPI_Fint *f_status)	
39	nt MPI_Status_f2f08(MPI_Fint *f_status, MPI_F08_status *f08_status)	
40 41	nt MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype)	
42	nt MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)	
43	nt MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)	
44 45	nt MPI_Type_match_size(int typeclass, int size, MPI_Datatype *datatyp	e)
46	PI_Fint MPI_Comm_c2f(MPI_Comm comm)	
47 48	PI_Comm MPI_Comm_f2c(MPI_Fint comm)	

MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)	1
MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)	2 3
MPI_Fint MPI_File_c2f(MPI_File file)	4
MPI_File MPI_File_f2c(MPI_Fint file)	5 6
MPI_Fint MPI_Group_c2f(MPI_Group group)	7
MPI_Group MPI_Group_f2c(MPI_Fint group)	8 9
MPI_Fint MPI_Info_c2f(MPI_Info info)	10
MPI_Info MPI_Info_f2c(MPI_Fint info)	11 12
MPI_Fint MPI_Message_c2f(MPI_Message message)	12
MPI_Message MPI_Message_f2c(MPI_Fint message)	14 15
MPI_Fint MPI_Op_c2f(MPI_Op op)	16
	17
MPI_Op MPI_Op_f2c(MPI_Fint op)	18 19
MPI_Fint MPI_Request_c2f(MPI_Request request)	20
MPI_Request MPI_Request_f2c(MPI_Fint request)	21 22
MPI_Fint MPI_Session_c2f(MPI_Session session)	22
MPI_Session MPI_Session_f2c(MPI_Fint session)	24
int MPI_Status_c2f08(const MPI_Status *c_status,	25 26
MPI_F08_status *f08_status)	27
<pre>int MPI_Status_c2f(const MPI_Status *c_status, MPI_Fint *f_status)</pre>	28
<pre>int MPI_Status_f082c(const MPI_F08_status *f08_status, MPI_Status *c_status)</pre>	29 30 31
<pre>int MPI_Status_f2c(const MPI_Fint *f_status, MPI_Status *c_status)</pre>	32
MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)	33 34
MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)	35
MPI_Fint MPI_Win_c2f(MPI_Win win)	36
	37 38
MPI_Win MPI_Win_f2c(MPI_Fint win)	39
A 2.12 Table / Profiling Interface C Pindings	40
A.2.13 Tools / Profiling Interface C Bindings	41 42
<pre>int MPI_Pcontrol(const int level,)</pre>	43
	$44 \\ 45$
A.2.14 Tools / MPI Tool Information Interface C Bindings	45 46
<pre>int MPI_T_category_changed(int *stamp)</pre>	47
	48

1 int MPI_T_category_get_categories(int cat_index, int len, int indices[]) $\mathbf{2}$ int MPI_T_category_get_cvars(int cat_index, int len, int indices[]) 3 4 int MPI_T_category_get_events(int cat_index, int len, int indices[]) 5int MPI_T_category_get_index(const char *name, int *cat_index) 6 $\overline{7}$ int MPI_T_category_get_info(int cat_index, char *name, int *name_len, 8 char *desc, int *desc_len, int *num_cvars, int *num_pvars, 9 int *num_categories) 10 int MPI_T_category_get_num_events(int cat_index, int *num_events) 11 12int MPI_T_category_get_num(int *num_cat) 13 int MPI_T_category_get_pvars(int cat_index, int len, int indices[]) 1415int MPI_T_cvar_get_index(const char *name, int *cvar_index) 16int MPI_T_cvar_get_info(int cvar_index, char *name, int *name_len, 17int *verbosity, MPI_Datatype *datatype, MPI_T_enum *enumtype, 18 char *desc, int *desc_len, int *bind, int *scope) 19 20int MPI_T_cvar_get_num(int *num_cvar) 21int MPI_T_cvar_handle_alloc(int cvar_index, void *obj_handle, 22MPI_T_cvar_handle *handle, int *count) 23 24 int MPI_T_cvar_handle_free(MPI_T_cvar_handle *handle) 2526int MPI_T_cvar_read(MPI_T_cvar_handle handle, void *buf) 27int MPI_T_cvar_write(MPI_T_cvar_handle handle, const void *buf) 28 29int MPI_T_enum_get_info(MPI_T_enum enumtype, int *num, char *name, 30 int *name_len) 31 int MPI_T_enum_get_item(MPI_T_enum enumtype, int index, int *value, 32 char *name, int *name_len) 33 34int MPI_T_event_callback_get_info(35 MPI_T_event_registration event_registration, 36 MPI_T_cb_safety cb_safety, MPI_Info *info_used) 37 int MPI_T_event_callback_set_info(38 MPI_T_event_registration event_registration, 39 MPI_T_cb_safety cb_saftey, MPI_Info info) 4041 int MPI_T_event_copy(MPI_T_event_instance event_instance, void *buffer) 42int MPI_T_event_get_index(const char *name, int *event_index) 43 44int MPI_T_event_get_info(int event_index, char *name, int *name_len, 45int *verbosity, MPI_Datatype *array_of_datatypes, 46MPI_Aint *array_of_displacements, int *num_elements, 47MPI_T_enum *enumtype, MPI_Info* info, char *desc, 48

<pre>int *desc_len, int *bind)</pre>	$\frac{1}{2}$
<pre>int MPI_T_event_get_num(int *num_events)</pre>	3
int MPI_T_event_get_source(MPI_T_event_instance event_instance,	4
int *source_index)	5
<pre>int MPI_T_event_get_timestamp(MPI_T_event_instance event_instance,</pre>	6
MPI_Count *event_timestamp)	7 8
•	9
<pre>int MPI_T_event_handle_alloc(int event_index, void *obj_handle,</pre>	10
	11
int MPI_T_event_handle_free(MPI_T_event_registration event_registration,	12
MPI_T_event_free_cb_function free_cb_function)	13
<pre>int MPI_T_event_handle_get_info(</pre>	14 15
MPI_T_event_registration event_registration,	16
MPI_Info *info_used)	17
<pre>int MPI_T_event_handle_set_info(</pre>	18
<pre>MPI_T_event_registration event_registration, MPI_Info info)</pre>	19
<pre>int MPI_T_event_read(MPI_T_event_instance event_instance,</pre>	20
<pre>int element_index, void *buffer)</pre>	21 22
<pre>int MPI_T_event_register_callback(</pre>	23
MPI_T_event_registration event_registration,	24
<pre>MPI_T_cb_safety cb_safety, MPI_Info info, void *user_data,</pre>	25
MPI_T_event_cb_function event_cb_function)	26
int MPI_T_event_set_dropped_handler(27 28
MPI_T_event_registration event_registration,	29
<pre>MPI_T_event_dropped_cb_function dropped_cb_function)</pre>	30
<pre>int MPI_T_finalize(void)</pre>	31
<pre>int MPI_T_init_thread(int required, int *provided)</pre>	32
<pre>int MPI_T_pvar_get_index(const char *name, int var_class, int *pvar_index)</pre>	33 34
	35
<pre>int MPI_T_pvar_get_info(int pvar_index, char *name, int *name_len,</pre>	36
<pre>int *verbosity, int *var_class, MPI_Datatype *datatype, MPI_T_enum *enumtype, char *desc, int *desc_len, int *bind,</pre>	37
int *readonly, int *continuous, int *atomic)	38
	39 40
<pre>int MPI_T_pvar_get_num(int *num_pvar)</pre>	40
<pre>int MPI_T_pvar_handle_alloc(MPI_T_pvar_session session, int pvar_index,</pre>	42
<pre>void *obj_handle, MPI_T_pvar_handle *handle, int *count)</pre>	43
<pre>int MPI_T_pvar_handle_free(MPI_T_pvar_session session,</pre>	44
MPI_T_pvar_handle *handle)	45
<pre>int MPI_T_pvar_read(MPI_T_pvar_session session, MPI_T_pvar_handle handle,</pre>	46 47
void *buf)	48

1	<pre>int MPI_T_pvar_readreset(MPI_T_pvar_session session,</pre>
2	<pre>MPI_T_pvar_handle handle, void *buf)</pre>
$\frac{3}{4}$	<pre>int MPI_T_pvar_reset(MPI_T_pvar_session session, MPI_T_pvar_handle handle)</pre>
5	<pre>int MPI_T_pvar_session_create(MPI_T_pvar_session *session)</pre>
6 7	int MPI_T_pvar_session_free(MPI_T_pvar_session *session)
8	int MPI_T_pvar_start(MPI_T_pvar_session session, MPI_T_pvar_handle handle)
9 10	int MPI_T_pvar_stop(MPI_T_pvar_session session, MPI_T_pvar_handle handle)
11	int MPI_T_pvar_write(MPI_T_pvar_session session, MPI_T_pvar_handle handle,
12	const void *buf)
13 14	<pre>int MPI_T_source_get_info(int source_index, char *name, int *name_len,</pre>
15	char *desc, int *desc_len, MPI_T_source_order *ordering,
16	<pre>MPI_Count *ticks_per_second, MPI_Count *max_ticks,</pre>
17	MPI_Info *info)
18 19	<pre>int MPI_T_source_get_num(int *num_sources)</pre>
20	<pre>int MPI_T_source_get_timestamp(int source_index, MPI_Count *timestamp)</pre>
21	
22 23	A.2.15 Deprecated C Bindings
24	int MPI_Attr_delete(MPI_Comm comm, int keyval)
25 26	<pre>int MPI_Attr_get(MPI_Comm comm, int keyval, void *attribute_val, int *flag)</pre>
27	<pre>int MPI_Attr_put(MPI_Comm comm, int keyval, void *attribute_val)</pre>
28 29	int MPI_DUP_FN(MPI_Comm oldcomm, int keyval, void *extra_state,
30	<pre>void *attribute_val_in, void *attribute_val_out, int *flag)</pre>
31 32	<pre>int MPI_Info_get(MPI_Info info, const char *key, int valuelen, char *value,</pre>
33	<pre>int *flag)</pre>
34 35	<pre>int MPI_Info_get_valuelen(MPI_Info info, const char *key, int *valuelen,</pre>
36	<pre>int MPI_Keyval_create(MPI_Copy_function *copy_fn,</pre>
37	MPI_Delete_function *delete_fn, int *keyval,
38	void *extra_state)
39 40	int MPI_Keyval_free(int *keyval)
41	
42	<pre>int MPI_NULL_COPY_FN(MPI_Comm oldcomm, int keyval, void *extra_state,</pre>
43	C C
44 45	<pre>int MPI_NULL_DELETE_FN(MPI_Comm comm, int keyval, void *attribute_val, void *extra_state)</pre>
45	
47	
48	

A.3. FORTRAN 2008 BINDINGS WITH THE MPI_F08 MODULE	821
A.3 Fortran 2008 Bindings with the mpi_f08 Module	1
A.3.1 Point-to-Point Communication Fortran 2008 Bindings	2 3
Fortran 2008 binding	4
MPI_Bsend(buf, count, datatype, dest, tag, comm, ierror)	5
TYPE(*), DIMENSION(), INTENT(IN) :: buf	6 7
INTEGER, INTENT(IN) :: count, dest, tag	8
TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm	9
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10
	11
<pre>MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf</pre>	12 13
INTEGER, INTENT(IN) :: count, dest, tag	14
TYPE(MPI_Datatype), INTENT(IN) :: datatype	15
TYPE(MPI_Comm), INTENT(IN) :: comm	16
TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror	17 18
	19
MPI_Buffer_attach(buffer, size, ierror)	20
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buffer INTEGER, INTENT(IN) :: size	21
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	22
MPI_Buffer_detach(buffer_addr, size, ierror)	23 24
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	25
TYPE(C_PTR), INTENT(OUT) :: buffer_addr	26
INTEGER, INTENT(OUT) :: size	27
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	28 29
MPI_Cancel(request, ierror)	29 30
TYPE(MPI_Request), INTENT(IN) :: request	31
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	32
MPI_Get_count(status, datatype, count, ierror)	33
TYPE(MPI_Status), INTENT(IN) :: status	34 35
TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(OUT) :: count	36
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	37
MPI_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror)	38
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	39
INTEGER, INTENT(IN) :: count, dest, tag	40 41
TYPE(MPI_Datatype), INTENT(IN) :: datatype	42
TYPE(MPI_Comm), INTENT(IN) :: comm	43
TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror	44
	45
<pre>MPI_Improbe(source, tag, comm, flag, message, status, ierror) INTEGER, INTENT(IN) :: source, tag</pre>	46 47
INIEGEN, INIENI(IN) SOULCE, LAG	48

```
1
         TYPE(MPI_Comm), INTENT(IN) :: comm
2
         LOGICAL, INTENT(OUT) :: flag
3
         TYPE(MPI_Message), INTENT(OUT) :: message
4
         TYPE(MPI_Status) :: status
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     MPI_Imrecv(buf, count, datatype, message, request, ierror)
7
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
8
         INTEGER, INTENT(IN) :: count
9
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
10
         TYPE(MPI_Message), INTENT(INOUT) :: message
11
         TYPE(MPI_Request), INTENT(OUT) :: request
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     MPI_Iprobe(source, tag, comm, flag, status, ierror)
15
         INTEGER, INTENT(IN) :: source, tag
16
         TYPE(MPI_Comm), INTENT(IN) :: comm
17
         LOGICAL, INTENT(OUT) :: flag
18
         TYPE(MPI_Status) :: status
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
     MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror)
21
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
22
         INTEGER, INTENT(IN) :: count, source, tag
23
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
24
         TYPE(MPI_Comm), INTENT(IN) :: comm
25
         TYPE(MPI_Request), INTENT(OUT) :: request
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
     MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror)
29
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
30
         INTEGER, INTENT(IN) :: count, dest, tag
31
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
32
         TYPE(MPI_Comm), INTENT(IN) :: comm
33
         TYPE(MPI_Request), INTENT(OUT) :: request
34
        INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
     MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror)
36
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
37
         INTEGER, INTENT(IN) :: count, dest, tag
38
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
39
         TYPE(MPI_Comm), INTENT(IN) :: comm
40
         TYPE(MPI_Request), INTENT(OUT) :: request
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
     MPI_Isendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
44
                   comm, request, ierror)
45
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
46
         INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag
47
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
48
```

```
1
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   2
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Isendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
              recvcount, recvtype, source, recvtag, comm, request, ierror)
                                                                                   6
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source,
              recvtag
                                                                                   9
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                   10
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                   11
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   12
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  14
                                                                                   15
MPI_Issend(buf, count, datatype, dest, tag, comm, request, ierror)
                                                                                   16
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                   17
    INTEGER, INTENT(IN) :: count, dest, tag
                                                                                   18
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   19
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   20
                                                                                  21
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  22
MPI_Mprobe(source, tag, comm, message, status, ierror)
                                                                                  23
    INTEGER, INTENT(IN) :: source, tag
                                                                                   24
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  25
    TYPE(MPI_Message), INTENT(OUT) :: message
                                                                                   26
    TYPE(MPI_Status) :: status
                                                                                  27
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  28
                                                                                  29
MPI_Mrecv(buf, count, datatype, message, status, ierror)
    TYPE(*), DIMENSION(..) :: buf
                                                                                  30
                                                                                   31
    INTEGER, INTENT(IN) :: count
                                                                                   32
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   33
    TYPE(MPI_Message), INTENT(INOUT) :: message
                                                                                  34
   TYPE(MPI_Status) :: status
                                                                                  35
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  36
MPI_Probe(source, tag, comm, status, ierror)
                                                                                  37
    INTEGER, INTENT(IN) :: source, tag
                                                                                  38
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   39
    TYPE(MPI_Status) :: status
                                                                                   40
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  41
                                                                                  42
MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)
                                                                                  43
    TYPE(*), DIMENSION(..) :: buf
                                                                                   44
    INTEGER, INTENT(IN) :: count, source, tag
                                                                                   45
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   46
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   47
    TYPE(MPI_Status) :: status
                                                                                   48
```

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror)
3
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
4
         INTEGER, INTENT(IN) :: count, source, tag
5
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
6
         TYPE(MPI_Comm), INTENT(IN) :: comm
7
         TYPE(MPI_Request), INTENT(OUT) :: request
8
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10
     MPI_Request_free(request, ierror)
11
         TYPE(MPI_Request), INTENT(INOUT) :: request
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13
     MPI_Request_get_status(request, flag, status, ierror)
14
         TYPE(MPI_Request), INTENT(IN) :: request
15
         LOGICAL, INTENT(OUT) :: flag
16
         TYPE(MPI_Status) :: status
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
     MPI_Rsend(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_Rsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
26
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
27
         INTEGER, INTENT(IN) :: count, dest, tag
28
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
         TYPE(MPI_Comm), INTENT(IN) :: comm
30
         TYPE(MPI_Request), INTENT(OUT) :: request
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33
     MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
34
        TYPE(*), DIMENSION(..), INTENT(IN) :: buf
35
         INTEGER, INTENT(IN) :: count, dest, tag
36
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
37
         TYPE(MPI_Comm), INTENT(IN) :: comm
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
     MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)
40
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
41
         INTEGER, INTENT(IN) :: count, dest, tag
42
         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
48
```

```
MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
                                                                                   2
              comm, status, ierror)
                                                                                   3
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   5
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   6
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
                                                                                  10
             recvcount, recvtype, source, recvtag, comm, status, ierror)
                                                                                  11
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                  12
    INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source,
                                                                                  13
              recvtag
                                                                                  14
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  15
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  16
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  17
    TYPE(MPI_Status) :: status
                                                                                  18
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  19
MPI_Ssend(buf, count, datatype, dest, tag, comm, ierror)
                                                                                  20
                                                                                  21
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
                                                                                  22
    INTEGER, INTENT(IN) :: count, dest, tag
                                                                                  23
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  24
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  25
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  26
MPI_Ssend_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
                                                                                  32
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  33
                                                                                  34
MPI_Startall(count, array_of_requests, ierror)
                                                                                  35
    INTEGER, INTENT(IN) :: count
                                                                                  36
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
                                                                                  37
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  38
MPI_Start(request, ierror)
                                                                                  39
    TYPE(MPI_Request), INTENT(INOUT) :: request
                                                                                  40
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  41
                                                                                  42
MPI_Testall(count, array_of_requests, flag, array_of_statuses, ierror)
                                                                                  43
    INTEGER, INTENT(IN) :: count
                                                                                  44
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
                                                                                  45
    LOGICAL, INTENT(OUT) :: flag
                                                                                  46
    TYPE(MPI_Status) :: array_of_statuses(*)
                                                                                  47
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  48
```

```
1
    MPI_Testany(count, array_of_requests, index, flag, status, ierror)
\mathbf{2}
         INTEGER, INTENT(IN) :: count
3
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
4
         INTEGER, INTENT(OUT) :: index
5
         LOGICAL, INTENT(OUT) :: flag
6
         TYPE(MPI_Status) :: status
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
    MPI_Test_cancelled(status, flag, ierror)
9
         TYPE(MPI_Status), INTENT(IN) :: status
10
         LOGICAL, INTENT(OUT) :: flag
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13
     MPI_Test(request, flag, status, ierror)
14
         TYPE(MPI_Request), INTENT(INOUT) :: request
15
         LOGICAL, INTENT(OUT) :: flag
16
         TYPE(MPI_Status) :: status
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
     MPI_Testsome(incount, array_of_requests, outcount, array_of_indices,
19
                   array_of_statuses, ierror)
20
         INTEGER, INTENT(IN) :: incount
21
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)
22
         INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
23
         TYPE(MPI_Status) :: array_of_statuses(*)
24
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26
     MPI_Waitall(count, array_of_requests, array_of_statuses, ierror)
27
         INTEGER, INTENT(IN) :: count
28
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
29
         TYPE(MPI_Status) :: array_of_statuses(*)
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
     MPI_Waitany(count, array_of_requests, index, status, ierror)
32
         INTEGER, INTENT(IN) :: count
33
        TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
34
         INTEGER, INTENT(OUT) :: index
35
         TYPE(MPI_Status) :: status
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
     MPI_Wait(request, status, ierror)
39
         TYPE(MPI_Request), INTENT(INOUT) :: request
40
         TYPE(MPI_Status) :: status
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
     MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices,
43
                   array_of_statuses, ierror)
44
         INTEGER, INTENT(IN) :: incount
45
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)
46
         INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
47
         TYPE(MPI_Status) :: array_of_statuses(*)
48
```

A.3. FORTRAN 2008 BINDINGS WITH THE MPI_F08 MODULE	827
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1 2
A.3.2 Datatypes Fortran 2008 Bindings	3 4
INTEGER(KIND=MPI_ADDRESS_KIND) MPI_Aint_add(base, disp) INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: base, disp	5 6
INTEGER(KIND=MPI_ADDRESS_KIND) MPI_Aint_diff(addr1, addr2) INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: addr1, addr2	7 8 9
<pre>MPI_Get_address(location, address, ierror) TYPE(*), DIMENSION(), ASYNCHRONOUS :: location INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: address INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	10 11 12 13 14
<pre>MPI_Pack_external(datarep, inbuf, incount, datatype, outbuf, outsize,</pre>	15 16 17 18 19 20 21 22 23 24
<pre>MPI_Pack_external_size(datarep, incount, datatype, size, ierror) CHARACTER(LEN=*), INTENT(IN) :: datarep INTEGER, INTENT(IN) :: incount TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	25 26 27 28 29 30 31
<pre>MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierr TYPE(*), DIMENSION(), INTENT(IN) :: inbuf INTEGER, INTENT(IN) :: incount, outsize TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(*), DIMENSION() :: outbuf INTEGER, INTENT(INOUT) :: position TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	
<pre>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</pre>	40 41 42 43 44 45 46
<pre>MPI_Type_commit(datatype, ierror) TYPE(MPI_Datatype), INTENT(INOUT) :: datatype</pre>	47 48

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
2
     MPI_Type_contiguous(count, oldtype, newtype, ierror)
3
         INTEGER, INTENT(IN) :: count
4
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
5
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
     MPI_Type_create_darray(size, rank, ndims, array_of_gsizes,
9
                   array_of_distribs, array_of_dargs, array_of_psizes, order,
10
                   oldtype, newtype, ierror)
11
         INTEGER, INTENT(IN) :: size, rank, ndims, array_of_gsizes(ndims),
12
                   array_of_distribs(ndims), array_of_dargs(ndims),
13
                   array_of_psizes(ndims), order
14
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
15
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
     MPI_Type_create_hindexed_block(count, blocklength, array_of_displacements,
18
                   oldtype, newtype, ierror)
19
         INTEGER, INTENT(IN) :: count, blocklength
20
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
21
                   array_of_displacements(count)
22
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
23
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
24
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26
     MPI_Type_create_hindexed(count, array_of_blocklengths,
27
                   array_of_displacements, oldtype, newtype, ierror)
28
         INTEGER, INTENT(IN) :: count, array_of_blocklengths(count)
29
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
30
                   array_of_displacements(count)
31
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
32
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
     MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,
35
                   ierror)
36
         INTEGER, INTENT(IN) :: count, blocklength
37
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: stride
38
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
39
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
     MPI_Type_create_indexed_block(count, blocklength, array_of_displacements,
43
                   oldtype, newtype, ierror)
44
         INTEGER, INTENT(IN) :: count, blocklength,
45
                   array_of_displacements(count)
46
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
47
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
48
```

1 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2 MPI_Type_create_resized(oldtype, lb, extent, newtype, ierror) TYPE(MPI_Datatype), INTENT(IN) :: oldtype 4 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: lb, extent 5 TYPE(MPI_Datatype), INTENT(OUT) :: newtype 6 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 7 MPI_Type_create_struct(count, array_of_blocklengths, 9 array_of_displacements, array_of_types, newtype, ierror) 10 INTEGER, INTENT(IN) :: count, array_of_blocklengths(count) 11 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: array_of_displacements(count) 1213 TYPE(MPI_Datatype), INTENT(IN) :: array_of_types(count) 14 TYPE(MPI_Datatype), INTENT(OUT) :: newtype 15INTEGER, OPTIONAL, INTENT(OUT) :: ierror 16MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes, 17 array_of_starts, order, oldtype, newtype, ierror) 18 INTEGER, INTENT(IN) :: ndims, array_of_sizes(ndims), 19 array_of_subsizes(ndims), array_of_starts(ndims), order 20TYPE(MPI_Datatype), INTENT(IN) :: oldtype 21TYPE(MPI_Datatype), INTENT(OUT) :: newtype 22 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 23 24MPI_Type_dup(oldtype, newtype, ierror) 25TYPE(MPI_Datatype), INTENT(IN) :: oldtype 26TYPE(MPI_Datatype), INTENT(OUT) :: newtype 27INTEGER, OPTIONAL, INTENT(OUT) :: ierror 28 MPI_Type_free(datatype, ierror) 29 TYPE(MPI_Datatype), INTENT(INOUT) :: datatype 30 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 3132 MPI_Type_get_contents(datatype, max_integers, max_addresses, max_datatypes, 33 array_of_integers, array_of_addresses, array_of_datatypes, 34 ierror) 35 TYPE(MPI_Datatype), INTENT(IN) :: datatype 36 INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes 37 INTEGER, INTENT(OUT) :: array_of_integers(max_integers) 38 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: 39 array_of_addresses(max_addresses) 40 TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes) 41 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 42MPI_Type_get_elements(status, datatype, count, ierror) 43 TYPE(MPI_Status), INTENT(IN) :: status 44 TYPE(MPI_Datatype), INTENT(IN) :: datatype 45INTEGER, INTENT(OUT) :: count 46INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4748

```
1
     MPI_Type_get_elements_x(status, datatype, count, ierror)
\mathbf{2}
         TYPE(MPI_Status), INTENT(IN) :: status
3
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
4
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: count
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes,
7
                   combiner, ierror)
8
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
9
         INTEGER, INTENT(OUT) :: num_integers, num_addresses, num_datatypes,
10
                   combiner
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13
     MPI_Type_get_extent(datatype, lb, extent, ierror)
14
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
15
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: lb, extent
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
     MPI_Type_get_extent_x(datatype, lb, extent, ierror)
18
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
19
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: lb, extent
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
     MPI_Type_get_true_extent(datatype, true_lb, true_extent, ierror)
23
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
24
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: true_lb, true_extent
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
     MPI_Type_get_true_extent_x(datatype, true_lb, true_extent, ierror)
27
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
28
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: true_lb, true_extent
29
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31
     MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements,
32
                   oldtype, newtype, ierror)
33
         INTEGER, INTENT(IN) :: count, array_of_blocklengths(count),
34
                   array_of_displacements(count)
35
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
36
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
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
```

```
1
    INTEGER, INTENT(IN) :: count, blocklength, stride
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
                                                                                   2
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outcount,
                                                                                   6
              datatype, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: datarep
    TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: insize
                                                                                   10
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position
                                                                                   11
    TYPE(*), DIMENSION(..) :: outbuf
                                                                                   12
    INTEGER, INTENT(IN) :: outcount
                                                                                   13
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   14
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   15
                                                                                   16
MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm,
                                                                                   17
              ierror)
                                                                                   18
    TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
                                                                                   19
    INTEGER, INTENT(IN) :: insize, outcount
                                                                                  20
    INTEGER, INTENT(INOUT) :: position
                                                                                  21
    TYPE(*), DIMENSION(..) :: outbuf
                                                                                  22
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  23
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   24
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  25
                                                                                   26
A.3.3 Collective Communication Fortran 2008 Bindings
                                                                                  27
                                                                                  28
MPI_Allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                  29
              recvtype, comm, info, request, ierror)
                                                                                  30
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                   31
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                   32
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  33
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  34
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  35
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  36
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  37
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  38
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  39
              comm, ierror)
                                                                                   40
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                   41
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                  42
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                   43
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                   44
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   45
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   46
                                                                                   47
                                                                                   48
```

```
1
     MPI_Allgatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
\mathbf{2}
                   displs, recvtype, comm, info, request, ierror)
3
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
4
         INTEGER, INTENT(IN) :: sendcount
5
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
6
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
7
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
8
         TYPE(MPI_Comm), INTENT(IN) :: comm
9
         TYPE(MPI_Info), INTENT(IN) :: info
10
         TYPE(MPI_Request), INTENT(OUT) :: request
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
     MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
13
                   recvtype, comm, ierror)
14
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
15
         INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
16
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
17
         TYPE(*), DIMENSION(...) :: recvbuf
18
         TYPE(MPI_Comm), INTENT(IN) :: comm
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
     MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info,
22
                   request, ierror)
23
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
24
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
25
         INTEGER, INTENT(IN) :: count
26
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
         TYPE(MPI_Op), INTENT(IN) :: op
28
         TYPE(MPI_Comm), INTENT(IN) :: comm
29
         TYPE(MPI_Info), INTENT(IN) :: info
30
         TYPE(MPI_Request), INTENT(OUT) :: request
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
     MPI_Allreduce(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_Alltoall_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
42
                   recvtype, comm, info, request, ierror)
43
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
44
         INTEGER, INTENT(IN) :: sendcount, recvcount
45
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
46
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
47
         TYPE(MPI_Comm), INTENT(IN) :: comm
48
```

```
1
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   2
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             comm, ierror)
                                                                                   6
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                   9
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  10
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  11
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  12
                                                                                  13
MPI_Alltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
             recvcounts, rdispls, recvtype, comm, info, request, ierror)
                                                                                  14
                                                                                  15
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  16
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
                                                                                  17
              recvcounts(*), rdispls(*)
                                                                                  18
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  19
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  20
                                                                                  21
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  22
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  24
MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
                                                                                  25
             rdispls, recvtype, comm, ierror)
                                                                                  26
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                  27
    INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
                                                                                  28
              rdispls(*)
                                                                                  29
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  30
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  31
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  32
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  33
                                                                                  34
MPI_Alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                  35
             recvcounts, rdispls, recvtypes, comm, info, request, ierror)
                                                                                  36
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  37
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
                                                                                  38
              recvcounts(*), 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
MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
                                                                                  47
             rdispls, recvtypes, comm, ierror)
                                                                                  48
```

```
1
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
2
         INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
3
                   rdispls(*)
4
         TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
5
         TYPE(*), DIMENSION(..) :: recvbuf
6
         TYPE(MPI_Comm), INTENT(IN) :: comm
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
     MPI_Barrier(comm, ierror)
9
         TYPE(MPI_Comm), INTENT(IN) :: comm
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
     MPI_Barrier_init(comm, info, request, ierror)
13
         TYPE(MPI_Comm), INTENT(IN) :: comm
14
         TYPE(MPI_Info), INTENT(IN) :: info
15
         TYPE(MPI_Request), INTENT(OUT) :: request
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
     MPI_Bcast(buffer, count, datatype, root, comm, ierror)
18
         TYPE(*), DIMENSION(..) :: buffer
19
         INTEGER, INTENT(IN) :: count, root
20
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
21
         TYPE(MPI_Comm), INTENT(IN) :: comm
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
24
     MPI_Bcast_init(buffer, count, datatype, root, comm, info, request, ierror)
25
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
26
         INTEGER, INTENT(IN) :: count, root
27
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
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_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, 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
44
     MPI_Exscan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
45
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
46
         TYPE(*), DIMENSION(..) :: 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
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Gather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  6
             root, comm, info, request, ierror)
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
                                                                                  9
    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
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  14
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  15
                                                                                  16
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  17
             root, comm, ierror)
                                                                                  18
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  19
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  20
                                                                                  21
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  22
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  24
MPI_Gatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                  25
             recvtype, root, comm, info, request, ierror)
                                                                                  26
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  27
    INTEGER, INTENT(IN) :: sendcount, root
                                                                                  28
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  29
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                  30
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
                                                                                  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_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                  37
             recvtype, root, comm, ierror)
                                                                                  38
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                  39
    INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*), root
                                                                                  40
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  41
    TYPE(*), DIMENSION(...) :: recvbuf
                                                                                  42
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  43
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  44
MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  45
              comm, request, ierror)
                                                                                  46
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  47
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                  48
```

```
1
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
2
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
3
         TYPE(MPI_Comm), INTENT(IN) :: comm
4
         TYPE(MPI_Request), INTENT(OUT) :: request
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
7
                  recvtype, comm, request, ierror)
8
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
9
         INTEGER, INTENT(IN) :: sendcount
10
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
11
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
12
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
13
         TYPE(MPI_Comm), INTENT(IN) :: comm
14
         TYPE(MPI_Request), INTENT(OUT) :: request
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
     MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request,
18
                   ierror)
19
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
20
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
21
         INTEGER, INTENT(IN) :: count
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_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
28
                   comm, request, ierror)
29
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
30
         INTEGER, INTENT(IN) :: sendcount, recvcount
31
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
32
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
33
         TYPE(MPI_Comm), INTENT(IN) :: comm
34
         TYPE(MPI_Request), INTENT(OUT) :: request
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
     MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
38
                  rdispls, recvtype, comm, request, ierror)
39
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
40
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
41
                   recvcounts(*), rdispls(*)
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
```

```
1
MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                   2
              recvcounts, rdispls, recvtypes, comm, request, ierror)
                                                                                   3
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
              recvcounts(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
              recvtypes(*)
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  10
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  11
                                                                                  12
MPI_Ibarrier(comm, request, ierror)
                                                                                  13
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  14
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  15
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  16
                                                                                  17
MPI_Ibcast(buffer, count, datatype, root, comm, request, ierror)
                                                                                  18
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
                                                                                  19
    INTEGER, INTENT(IN) :: count, root
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  20
                                                                                  21
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  22
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  24
MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
                                                                                  25
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  26
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  27
    INTEGER, INTENT(IN) :: count
                                                                                  28
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  29
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  30
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  31
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  32
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  33
                                                                                  34
MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  35
              root, comm, request, ierror)
                                                                                  36
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  37
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
                                                                                  38
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  39
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  40
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  41
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  42
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  43
MPI_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                  44
             recvtype, root, comm, request, ierror)
                                                                                  45
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  46
    INTEGER, INTENT(IN) :: sendcount, displs(*), root
                                                                                  47
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  48
```

```
1
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
2
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
3
         TYPE(MPI_Comm), INTENT(IN) :: comm
4
         TYPE(MPI_Request), INTENT(OUT) :: request
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
7
                  request, ierror)
8
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
9
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
10
         INTEGER, INTENT(IN) :: recvcount
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(sendbuf, recvbuf, recvcounts, datatype, op, comm,
18
                  request, ierror)
19
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
20
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
21
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
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(sendbuf, recvbuf, count, datatype, op, root, comm, request,
28
                   ierror)
29
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
30
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
31
         INTEGER, INTENT(IN) :: count, root
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, 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
48
```

```
MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
              root, comm, request, ierror)
                                                                                   2
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                   5
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                   6
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   10
MPI_Iscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
                                                                                   11
              recvtype, root, comm, request, ierror)
                                                                                   12
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                   13
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*)
                                                                                   14
    INTEGER, INTENT(IN) :: displs(*), recvcount, root
                                                                                   15
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                   16
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                   17
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   18
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   19
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   20
                                                                                  21
MPI_Op_commutative(op, commute, ierror)
                                                                                  22
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  23
    LOGICAL, INTENT(OUT) :: commute
                                                                                   24
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   25
MPI_Op_create(user_fn, commute, op, ierror)
                                                                                   26
    PROCEDURE(MPI_User_function) :: user_fn
                                                                                  27
    LOGICAL, INTENT(IN) :: commute
                                                                                  28
    TYPE(MPI_Op), INTENT(OUT) :: op
                                                                                  29
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   30
                                                                                   31
MPI_Op_free(op, ierror)
                                                                                   32
    TYPE(MPI_Op), INTENT(INOUT) :: op
                                                                                   33
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  34
MPI_Reduce_init(sendbuf, recvbuf, count, datatype, op, root, comm, info,
                                                                                  35
             request, ierror)
                                                                                  36
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  37
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                   38
    INTEGER, INTENT(IN) :: count, root
                                                                                   39
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   40
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                   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_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror)
                                                                                   47
    TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
                                                                                   48
```

```
1
         TYPE(*), DIMENSION(..) :: inoutbuf
2
         INTEGER, INTENT(IN) :: count
3
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
4
         TYPE(MPI_Op), INTENT(IN) :: op
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     MPI_Reduce_scatter_block_init(sendbuf, recvbuf, recvcount, datatype, op,
7
                  comm, info, request, ierror)
8
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
9
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
10
         INTEGER, INTENT(IN) :: recvcount
11
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
12
         TYPE(MPI_Op), INTENT(IN) :: op
13
         TYPE(MPI_Comm), INTENT(IN) :: comm
14
         TYPE(MPI_Info), INTENT(IN) :: info
15
         TYPE(MPI_Request), INTENT(OUT) :: request
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18
     MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
19
                   ierror)
20
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
21
         TYPE(*), DIMENSION(..) :: recvbuf
22
         INTEGER, INTENT(IN) :: recvcount
23
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
24
         TYPE(MPI_Op), INTENT(IN) :: op
25
         TYPE(MPI_Comm), INTENT(IN) :: comm
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     MPI_Reduce_scatter_init(sendbuf, recvbuf, recvcounts, datatype, op, comm,
28
                   info, request, ierror)
29
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
30
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
31
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
32
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
33
         TYPE(MPI_Op), INTENT(IN) :: op
34
         TYPE(MPI_Comm), INTENT(IN) :: comm
35
         TYPE(MPI_Info), INTENT(IN) :: info
36
         TYPE(MPI_Request), INTENT(OUT) :: request
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
     MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
40
                   ierror)
41
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
42
         TYPE(*), DIMENSION(..) :: recvbuf
43
         INTEGER, INTENT(IN) :: recvcounts(*)
44
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
45
         TYPE(MPI_Op), INTENT(IN) :: op
46
         TYPE(MPI_Comm), INTENT(IN) :: comm
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

```
1
MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
                                                                                   2
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER, INTENT(IN) :: count, root
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   5
                                                                                   6
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Scan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
                                                                                  10
              ierror)
                                                                                  11
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  12
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  13
    INTEGER, INTENT(IN) :: count
                                                                                  14
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  15
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  16
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  17
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  18
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  19
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  20
                                                                                  21
MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
                                                                                  22
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  23
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  24
    INTEGER, INTENT(IN) :: count
                                                                                  25
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  26
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  27
                                                                                  28
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  29
MPI_Scatter_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                  30
             recvtype, root, comm, info, request, ierror)
                                                                                  31
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  32
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
                                                                                  33
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  34
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                  35
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  36
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  37
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  39
                                                                                  40
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  41
             root, comm, ierror)
                                                                                  42
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                  43
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
                                                                                  44
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  45
    TYPE(*), DIMENSION(...) :: recvbuf
                                                                                  46
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  47
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  48
```

```
1
     MPI_Scatterv_init(sendbuf, sendcounts, displs, sendtype, recvbuf,
\mathbf{2}
                   recvcount, recvtype, root, comm, info, request, ierror)
3
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
4
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
5
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
6
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
7
         INTEGER, INTENT(IN) :: recvcount, root
8
         TYPE(MPI_Comm), INTENT(IN) :: comm
9
         TYPE(MPI_Info), INTENT(IN) :: info
10
         TYPE(MPI_Request), INTENT(OUT) :: request
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
     MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
13
                   recvtype, root, comm, ierror)
14
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
15
         INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root
16
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
17
         TYPE(*), DIMENSION(...) :: recvbuf
18
         TYPE(MPI_Comm), INTENT(IN) :: comm
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
22
     A.3.4 Groups, Contexts, Communicators, and Caching Fortran 2008 Bindings
23
     MPI_Comm_compare(comm1, comm2, result, ierror)
24
         TYPE(MPI_Comm), INTENT(IN) :: comm1, comm2
25
         INTEGER, INTENT(OUT) :: result
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
     MPI_Comm_create(comm, group, newcomm, ierror)
29
         TYPE(MPI_Comm), INTENT(IN) :: comm
30
         TYPE(MPI_Group), INTENT(IN) :: group
31
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     MPI_Comm_create_from_group(group, stringtag, info, errhandler, newcomm,
                   ierror)
35
         TYPE(MPI_Group), INTENT(IN) :: group
36
         CHARACTER(LEN=*), INTENT(IN) :: stringtag
37
         TYPE(MPI_Info), INTENT(IN) :: info
38
         TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
39
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
     MPI_Comm_create_group(comm, group, tag, newcomm, ierror)
43
         TYPE(MPI_Comm), INTENT(IN) :: comm
44
         TYPE(MPI_Group), INTENT(IN) :: group
45
         INTEGER, INTENT(IN) :: tag
46
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

```
MPI_Comm_create_keyval(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval,
                                                                                   2
              extra_state, ierror)
                                                                                   3
    PROCEDURE(MPI_Comm_copy_attr_function), INTENT(IN) :: comm_copy_attr_fn
    PROCEDURE(MPI_Comm_delete_attr_function), INTENT(IN) ::
                                                                                   4
                                                                                   5
              comm_delete_attr_fn
                                                                                   6
    INTEGER, INTENT(OUT) :: comm_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
                                                                                   7
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   9
MPI_Comm_delete_attr(comm, comm_keyval, ierror)
                                                                                   10
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   11
    INTEGER, INTENT(IN) :: comm_keyval
                                                                                   12
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   13
                                                                                   14
MPI_Comm_dup(comm, newcomm, ierror)
                                                                                   15
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   16
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
                                                                                   17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   18
MPI_COMM_DUP_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,
                                                                                   19
              attribute_val_out, flag, ierror)
                                                                                   20
    TYPE(MPI_Comm) :: oldcomm
                                                                                  21
    INTEGER :: comm_keyval, ierror
                                                                                  22
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                  23
              attribute_val_out
                                                                                   24
    LOGICAL :: flag
                                                                                   25
                                                                                   26
MPI_Comm_dup_with_info(comm, info, newcomm, ierror)
                                                                                  27
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   28
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  29
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
                                                                                   30
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   31
MPI_Comm_free(comm, ierror)
                                                                                   32
    TYPE(MPI_Comm), INTENT(INOUT) :: comm
                                                                                   33
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  34
                                                                                  35
MPI_Comm_free_keyval(comm_keyval, ierror)
                                                                                  36
    INTEGER, INTENT(INOUT) :: comm_keyval
                                                                                  37
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  38
MPI_Comm_get_attr(comm, comm_keyval, attribute_val, flag, ierror)
                                                                                  39
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   40
    INTEGER, INTENT(IN) :: comm_keyval
                                                                                   41
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
                                                                                  42
    LOGICAL, INTENT(OUT) :: flag
                                                                                  43
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   44
                                                                                   45
MPI_Comm_get_info(comm, info_used, ierror)
                                                                                   46
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   47
    TYPE(MPI_Info), INTENT(OUT) :: info_used
                                                                                   48
```

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Comm_get_name(comm, comm_name, resultlen, ierror)
3
         TYPE(MPI_Comm), INTENT(IN) :: comm
4
         CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: comm_name
5
         INTEGER, INTENT(OUT) :: resultlen
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
     MPI_Comm_group(comm, group, ierror)
9
         TYPE(MPI_Comm), INTENT(IN) :: comm
10
         TYPE(MPI_Group), INTENT(OUT) :: group
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
     MPI_Comm_idup(comm, newcomm, request, ierror)
13
         TYPE(MPI_Comm), INTENT(IN) :: comm
14
         TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS :: newcomm
15
         TYPE(MPI_Request), INTENT(OUT) :: request
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18
     MPI_Comm_idup_with_info(comm, info, newcomm, request, ierror)
19
         TYPE(MPI_Comm), INTENT(IN) :: comm
20
         TYPE(MPI_Info), INTENT(IN) :: info
21
         TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS :: newcomm
22
         TYPE(MPI_Request), INTENT(OUT) :: request
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
     MPI_COMM_NULL_COPY_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,
25
                   attribute_val_out, flag, ierror)
26
         TYPE(MPI_Comm) :: oldcomm
27
         INTEGER :: comm_keyval, ierror
28
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
29
                   attribute_val_out
30
         LOGICAL :: flag
31
32
     MPI_COMM_NULL_DELETE_FN(comm, comm_keyval, attribute_val, extra_state,
33
                   ierror)
34
        TYPE(MPI_Comm) :: comm
35
         INTEGER :: comm_keyval, ierror
36
         INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
37
     MPI_Comm_rank(comm, rank, ierror)
38
         TYPE(MPI_Comm), INTENT(IN) :: comm
39
         INTEGER, INTENT(OUT) :: rank
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
    MPI_Comm_remote_group(comm, group, ierror)
43
         TYPE(MPI_Comm), INTENT(IN) :: comm
44
         TYPE(MPI_Group), INTENT(OUT) :: group
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
     MPI_Comm_remote_size(comm, size, ierror)
47
         TYPE(MPI_Comm), INTENT(IN) :: comm
48
```

1 INTEGER, INTENT(OUT) :: size 2 INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Comm_set_attr(comm, comm_keyval, attribute_val, ierror) $\mathbf{4}$ TYPE(MPI_Comm), INTENT(IN) :: comm 5 INTEGER, INTENT(IN) :: comm_keyval 6 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Comm_set_info(comm, info, ierror) 10 TYPE(MPI_Comm), INTENT(IN) :: comm 11 TYPE(MPI_Info), INTENT(IN) :: info INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1213 MPI_Comm_set_name(comm, comm_name, ierror) 14 TYPE(MPI_Comm), INTENT(IN) :: comm 15CHARACTER(LEN=*), INTENT(IN) :: comm_name 16INTEGER, OPTIONAL, INTENT(OUT) :: ierror 17 18 MPI_Comm_size(comm, size, ierror) 19 TYPE(MPI_Comm), INTENT(IN) :: comm 20INTEGER, INTENT(OUT) :: size 21INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22 MPI_Comm_split(comm, color, key, newcomm, ierror) 23TYPE(MPI_Comm), INTENT(IN) :: comm 24INTEGER, INTENT(IN) :: color, key 25TYPE(MPI_Comm), INTENT(OUT) :: newcomm 26INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2728MPI_Comm_split_type(comm, split_type, key, info, newcomm, ierror) 29 TYPE(MPI_Comm), INTENT(IN) :: comm 30 INTEGER, INTENT(IN) :: split_type, key 31TYPE(MPI_Info), INTENT(IN) :: info 32 TYPE(MPI_Comm), INTENT(OUT) :: newcomm 33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 34 MPI_Comm_test_inter(comm, flag, ierror) 35TYPE(MPI_Comm), INTENT(IN) :: comm 36 LOGICAL, INTENT(OUT) :: flag 37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 38 39 MPI_Group_compare(group1, group2, result, ierror) 40 TYPE(MPI_Group), INTENT(IN) :: group1, group2 41 INTEGER, INTENT(OUT) :: result 42INTEGER, OPTIONAL, INTENT(OUT) :: ierror 43 MPI_Group_difference(group1, group2, newgroup, ierror) 44 TYPE(MPI_Group), INTENT(IN) :: group1, group2 45TYPE(MPI_Group), INTENT(OUT) :: newgroup 46INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4748

```
1
     MPI_Group_excl(group, n, ranks, newgroup, ierror)
\mathbf{2}
         TYPE(MPI_Group), INTENT(IN) :: group
3
         INTEGER, INTENT(IN) :: n, ranks(n)
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
4
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     MPI_Group_free(group, ierror)
7
         TYPE(MPI_Group), INTENT(INOUT) :: group
8
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10
     MPI_Group_from_session_pset(session, pset_name, newgroup, ierror)
11
         TYPE(MPI_Session), INTENT(IN) :: session
12
         CHARACTER(LEN=*), INTENT(IN) :: pset_name
13
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
     MPI_Group_incl(group, n, ranks, newgroup, ierror)
16
         TYPE(MPI_Group), INTENT(IN) :: group
17
         INTEGER, INTENT(IN) :: n, ranks(n)
18
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
     MPI_Group_intersection(group1, group2, newgroup, ierror)
22
         TYPE(MPI_Group), INTENT(IN) :: group1, group2
23
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
24
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
     MPI_Group_range_excl(group, n, ranges, newgroup, ierror)
26
         TYPE(MPI_Group), INTENT(IN) :: group
27
         INTEGER, INTENT(IN) :: n, ranges(3, n)
28
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
29
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31
     MPI_Group_range_incl(group, n, ranges, newgroup, ierror)
32
         TYPE(MPI_Group), INTENT(IN) :: group
33
         INTEGER, INTENT(IN) :: n, ranges(3, n)
34
        TYPE(MPI_Group), INTENT(OUT) :: newgroup
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
     MPI_Group_rank(group, rank, ierror)
37
         TYPE(MPI_Group), INTENT(IN) :: group
38
         INTEGER, INTENT(OUT) :: rank
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     MPI_Group_size(group, size, ierror)
42
         TYPE(MPI_Group), INTENT(IN) :: group
43
         INTEGER, INTENT(OUT) :: size
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
     MPI_Group_translate_ranks(group1, n, ranks1, group2, ranks2, ierror)
46
         TYPE(MPI_Group), INTENT(IN) :: group1, group2
47
         INTEGER, INTENT(IN) :: n, ranks1(n)
48
```

```
1
    INTEGER, INTENT(OUT) :: ranks2(n)
                                                                                  2
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Group_union(group1, group2, newgroup, ierror)
    TYPE(MPI_Group), INTENT(IN) :: group1, group2
                                                                                  5
    TYPE(MPI_Group), INTENT(OUT) :: newgroup
                                                                                  6
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Intercomm_create_from_groups(local_group, local_leader, remote_group,
                                                                                  9
             remote_leader, stringtag, info, errhandler, newintercomm,
                                                                                  10
             ierror)
                                                                                  11
    TYPE(MPI_Group), INTENT(IN) :: local_group, remote_group
    INTEGER, INTENT(IN) :: local_leader, remote_leader
                                                                                  12
                                                                                  13
    CHARACTER(LEN=*), INTENT(IN) :: stringtag
                                                                                  14
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  15
    TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
                                                                                  16
    TYPE(MPI_Comm), INTENT(OUT) :: newintercomm
                                                                                  17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  18
MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader,
                                                                                  19
             tag, newintercomm, ierror)
                                                                                  20
    TYPE(MPI_Comm), INTENT(IN) :: local_comm, peer_comm
                                                                                  21
    INTEGER, INTENT(IN) :: local_leader, remote_leader, tag
                                                                                  22
    TYPE(MPI_Comm), INTENT(OUT) :: newintercomm
                                                                                  23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  24
                                                                                  25
MPI_Intercomm_merge(intercomm, high, newintracomm, ierror)
                                                                                  26
    TYPE(MPI_Comm), INTENT(IN) :: intercomm
                                                                                  27
    LOGICAL, INTENT(IN) :: high
                                                                                  28
    TYPE(MPI_Comm), INTENT(OUT) :: newintracomm
                                                                                  29
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  30
MPI_Type_create_keyval(type_copy_attr_fn, type_delete_attr_fn, type_keyval,
                                                                                  31
              extra_state, ierror)
                                                                                  32
    PROCEDURE(MPI_Type_copy_attr_function), INTENT(IN) :: type_copy_attr_fn
                                                                                  33
   PROCEDURE(MPI_Type_delete_attr_function), INTENT(IN) ::
                                                                                  34
              type_delete_attr_fn
                                                                                  35
    INTEGER, INTENT(OUT) :: type_keyval
                                                                                  36
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
                                                                                  37
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  38
                                                                                  39
MPI_Type_delete_attr(datatype, type_keyval, ierror)
                                                                                  40
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  41
    INTEGER, INTENT(IN) :: type_keyval
                                                                                  42
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  43
MPI_TYPE_DUP_FN(oldtype, type_keyval, extra_state, attribute_val_in,
                                                                                  44
             attribute_val_out, flag, ierror)
                                                                                  45
    TYPE(MPI_Datatype) :: oldtype
                                                                                  46
    INTEGER :: type_keyval, ierror
                                                                                  47
                                                                                  48
```

```
1
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
2
                   attribute_val_out
3
         LOGICAL :: flag
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_NULL_COPY_FN(oldtype, type_keyval, extra_state, attribute_val_in,
21
                   attribute_val_out, flag, ierror)
22
         TYPE(MPI_Datatype) :: oldtype
23
         INTEGER :: type_keyval, ierror
24
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
25
                   attribute_val_out
26
         LOGICAL :: flag
27
     MPI_TYPE_NULL_DELETE_FN(datatype, type_keyval, attribute_val, extra_state,
28
                   ierror)
29
         TYPE(MPI_Datatype) :: datatype
30
         INTEGER :: type_keyval
31
         INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
32
         INTEGER, INTENT(OUT) :: ierror
33
34
     MPI_Type_set_attr(datatype, type_keyval, attribute_val, ierror)
35
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
36
         INTEGER, INTENT(IN) :: type_keyval
37
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
     MPI_Type_set_name(datatype, type_name, ierror)
40
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
41
         CHARACTER(LEN=*), INTENT(IN) :: type_name
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44
     MPI_Win_create_keyval(win_copy_attr_fn, win_delete_attr_fn, win_keyval,
45
                   extra_state, ierror)
46
         PROCEDURE(MPI_Win_copy_attr_function), INTENT(IN) :: win_copy_attr_fn
47
48
```

```
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
MPI_Win_delete_attr(win, win_keyval, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, INTENT(IN) :: win_keyval
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  10
                                                                                  11
MPI_WIN_DUP_FN(oldwin, win_keyval, extra_state, attribute_val_in,
             attribute_val_out, flag, ierror)
                                                                                  12
                                                                                  13
    TYPE(MPI_Win) :: oldwin
                                                                                  14
    INTEGER :: win_keyval, ierror
                                                                                  15
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                  16
              attribute_val_out
                                                                                  17
    LOGICAL :: flag
                                                                                  18
MPI_Win_free_keyval(win_keyval, ierror)
                                                                                  19
    INTEGER, INTENT(INOUT) :: win_keyval
                                                                                  20
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  21
                                                                                  22
MPI_Win_get_attr(win, win_keyval, attribute_val, flag, ierror)
                                                                                  23
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  24
    INTEGER, INTENT(IN) :: win_keyval
                                                                                  25
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
                                                                                  26
    LOGICAL, INTENT(OUT) :: flag
                                                                                  27
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  28
MPI_Win_get_name(win, win_name, resultlen, ierror)
                                                                                  29
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  30
    CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: win_name
                                                                                  31
    INTEGER, INTENT(OUT) :: resultlen
                                                                                  32
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  33
                                                                                  34
MPI_WIN_NULL_COPY_FN(oldwin, win_keyval, extra_state, attribute_val_in,
                                                                                  35
             attribute_val_out, flag, ierror)
                                                                                  36
    TYPE(MPI_Win) :: oldwin
                                                                                  37
    INTEGER :: win_keyval, ierror
                                                                                  38
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                  39
              attribute_val_out
                                                                                  40
    LOGICAL :: flag
                                                                                  41
MPI_WIN_NULL_DELETE_FN(win, win_keyval, attribute_val, 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
MPI_Win_set_attr(win, win_keyval, attribute_val, ierror)
                                                                                  47
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  48
```

```
1
         INTEGER, INTENT(IN) :: win_keyval
2
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Win_set_name(win, win_name, ierror)
5
         TYPE(MPI_Win), INTENT(IN) :: win
6
         CHARACTER(LEN=*), INTENT(IN) :: win_name
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
9
10
     A.3.5 Process Topologies Fortran 2008 Bindings
11
    MPI_Cart_coords(comm, rank, maxdims, coords, ierror)
12
         TYPE(MPI_Comm), INTENT(IN) :: comm
13
         INTEGER, INTENT(IN) :: rank, maxdims
14
         INTEGER, INTENT(OUT) :: coords(maxdims)
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
     MPI_Cart_create(comm_old, ndims, dims, periods, reorder, comm_cart, ierror)
18
         TYPE(MPI_Comm), INTENT(IN) :: comm_old
19
         INTEGER, INTENT(IN) :: ndims, dims(ndims)
20
         LOGICAL, INTENT(IN) :: periods(ndims), reorder
21
         TYPE(MPI_Comm), INTENT(OUT) :: comm_cart
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
     MPI_Cartdim_get(comm, ndims, ierror)
^{24}
         TYPE(MPI_Comm), INTENT(IN) :: comm
25
         INTEGER, INTENT(OUT) :: ndims
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
     MPI_Cart_get(comm, maxdims, dims, periods, coords, ierror)
29
         TYPE(MPI_Comm), INTENT(IN) :: comm
30
         INTEGER, INTENT(IN) :: maxdims
31
         INTEGER, INTENT(OUT) :: dims(maxdims), coords(maxdims)
32
         LOGICAL, INTENT(OUT) :: periods(maxdims)
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
     MPI_Cart_map(comm, ndims, dims, periods, newrank, ierror)
35
         TYPE(MPI_Comm), INTENT(IN) :: comm
36
         INTEGER, INTENT(IN) :: ndims, dims(ndims)
37
         LOGICAL, INTENT(IN) :: periods(ndims)
38
         INTEGER, INTENT(OUT) :: newrank
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     MPI_Cart_rank(comm, coords, rank, ierror)
42
         TYPE(MPI_Comm), INTENT(IN) :: comm
43
         INTEGER, INTENT(IN) :: coords(*)
44
         INTEGER, INTENT(OUT) :: rank
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
     MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror)
48
         TYPE(MPI_Comm), INTENT(IN) :: comm
```

```
1
    INTEGER, INTENT(IN) :: direction, disp
                                                                                   2
    INTEGER, INTENT(OUT) :: rank_source, rank_dest
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Cart_sub(comm, remain_dims, newcomm, ierror)
                                                                                   5
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   6
    LOGICAL, INTENT(IN) :: remain_dims(*)
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  10
MPI_Dims_create(nnodes, ndims, dims, ierror)
                                                                                  11
    INTEGER, INTENT(IN) :: nnodes, ndims
    INTEGER, INTENT(INOUT) :: dims(ndims)
                                                                                  12
                                                                                  13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  14
MPI_Dist_graph_create_adjacent(comm_old, indegree, sources, sourceweights,
                                                                                  15
              outdegree, destinations, destweights, info, reorder,
                                                                                  16
              comm_dist_graph, ierror)
                                                                                  17
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
                                                                                  18
    INTEGER, INTENT(IN) :: indegree, sources(indegree), sourceweights(*),
                                                                                  19
              outdegree, destinations(outdegree), destweights(*)
                                                                                  20
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  21
    LOGICAL, INTENT(IN) :: reorder
                                                                                  22
    TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
                                                                                  23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  24
                                                                                  25
MPI_Dist_graph_create(comm_old, n, sources, degrees, destinations, weights,
                                                                                  26
              info, reorder, comm_dist_graph, ierror)
                                                                                  27
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
                                                                                  28
    INTEGER, INTENT(IN) :: n, sources(n), degrees(n), destinations(*),
                                                                                  29
              weights(*)
                                                                                  30
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  31
    LOGICAL, INTENT(IN) :: reorder
                                                                                  32
    TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
                                                                                  33
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  34
MPI_Dist_graph_neighbors(comm, maxindegree, sources, sourceweights,
                                                                                  35
             maxoutdegree, destinations, destweights, ierror)
                                                                                  36
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  37
    INTEGER, INTENT(IN) :: maxindegree, maxoutdegree
                                                                                  38
    INTEGER, INTENT(OUT) :: sources(maxindegree),
                                                                                  39
              destinations(maxoutdegree)
                                                                                  40
    INTEGER :: sourceweights(*), destweights(*)
                                                                                  41
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  42
                                                                                  43
MPI_Dist_graph_neighbors_count(comm, indegree, outdegree, weighted, ierror)
                                                                                  44
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  45
    INTEGER, INTENT(OUT) :: indegree, outdegree
                                                                                  46
    LOGICAL, INTENT(OUT) :: weighted
                                                                                  47
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  48
```

```
1
     MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph,
2
                   ierror)
3
         TYPE(MPI_Comm), INTENT(IN) :: comm_old
4
         INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
5
         LOGICAL, INTENT(IN) :: reorder
6
         TYPE(MPI_Comm), INTENT(OUT) :: comm_graph
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
     MPI_Graphdims_get(comm, nnodes, nedges, ierror)
9
         TYPE(MPI_Comm), INTENT(IN) :: comm
10
         INTEGER, INTENT(OUT) :: nnodes, nedges
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13
     MPI_Graph_get(comm, maxindex, maxedges, index, edges, ierror)
14
         TYPE(MPI_Comm), INTENT(IN) :: comm
15
         INTEGER, INTENT(IN) :: maxindex, maxedges
16
         INTEGER, INTENT(OUT) :: index(maxindex), edges(maxedges)
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
     MPI_Graph_map(comm, nnodes, index, edges, newrank, ierror)
19
         TYPE(MPI_Comm), INTENT(IN) :: comm
20
         INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
21
         INTEGER, INTENT(OUT) :: newrank
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
24
     MPI_Graph_neighbors(comm, rank, maxneighbors, neighbors, ierror)
25
         TYPE(MPI_Comm), INTENT(IN) :: comm
26
         INTEGER, INTENT(IN) :: rank, maxneighbors
27
         INTEGER, INTENT(OUT) :: neighbors(maxneighbors)
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
     MPI_Graph_neighbors_count(comm, rank, nneighbors, ierror)
30
         TYPE(MPI_Comm), INTENT(IN) :: comm
31
         INTEGER, INTENT(IN) :: rank
32
         INTEGER, INTENT(OUT) :: nneighbors
33
        INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
     MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
36
                  recvtype, comm, request, ierror)
37
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
38
         INTEGER, INTENT(IN) :: sendcount, recvcount
39
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
40
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
41
         TYPE(MPI_Comm), INTENT(IN) :: comm
42
         TYPE(MPI_Request), INTENT(OUT) :: request
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
     MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
45
                  displs, recvtype, comm, request, ierror)
46
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
47
         INTEGER, INTENT(IN) :: sendcount
48
```

```
1
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  2
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  5
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  6
MPI_Ineighbor_alltoall(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_Ineighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
                                                                                  18
             recvcounts, rdispls, recvtype, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  19
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
                                                                                  20
                                                                                  21
              recvcounts(*), rdispls(*)
                                                                                  22
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  23
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  24
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  25
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  26
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  27
MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                  28
             recvcounts, rdispls, recvtypes, comm, request, ierror)
                                                                                  29
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  30
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
                                                                                  31
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                  32
              rdispls(*)
                                                                                  33
   TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                  34
              recvtypes(*)
                                                                                  35
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  36
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  37
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  39
                                                                                  40
MPI_Neighbor_allgather_init(sendbuf, sendcount, sendtype, recvbuf,
                                                                                  41
             recvcount, recvtype, comm, info, request, ierror)
                                                                                  42
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  43
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                  44
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  45
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  46
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  47
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  48
```

```
1
         TYPE(MPI_Request), INTENT(OUT) :: request
2
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
4
                  recvtype, comm, ierror)
5
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
6
         INTEGER, INTENT(IN) :: sendcount, recvcount
7
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
8
         TYPE(*), DIMENSION(..) :: recvbuf
9
         TYPE(MPI_Comm), INTENT(IN) :: comm
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
    MPI_Neighbor_allgatherv_init(sendbuf, sendcount, sendtype, recvbuf,
13
                  recvcounts, displs, recvtype, comm, info, request, ierror)
14
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
15
         INTEGER, INTENT(IN) :: sendcount, displs(*)
16
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
17
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
18
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
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
    MPI_Neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
24
                  displs, recvtype, comm, ierror)
25
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
26
         INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
27
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
28
         TYPE(*), DIMENSION(..) :: recvbuf
29
         TYPE(MPI_Comm), INTENT(IN) :: comm
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32
     MPI_Neighbor_alltoall_init(sendbuf, sendcount, sendtype, recvbuf,
33
                  recvcount, 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, recytype
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
     MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
43
                  recvtype, comm, ierror)
44
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
45
         INTEGER, INTENT(IN) :: sendcount, recvcount
46
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
47
         TYPE(*), DIMENSION(...) :: recvbuf
48
```

```
1
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   2
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Neighbor_alltoallv_init(sendbuf, sendcounts, sdispls, sendtype,
             recvbuf, recvcounts, rdispls, recvtype, comm, info, request,
              ierror)
                                                                                   6
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
              recvcounts(*), rdispls(*)
                                                                                   9
    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
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  14
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  15
                                                                                  16
MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
                                                                                  17
             recvcounts, rdispls, recvtype, comm, ierror)
                                                                                  18
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  19
    INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
                                                                                  20
              rdispls(*)
                                                                                  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_Neighbor_alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes,
                                                                                  26
             recvbuf, recvcounts, rdispls, recvtypes, comm, info, request,
                                                                                  27
             ierror)
                                                                                  28
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  29
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
                                                                                  30
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                  31
              rdispls(*)
                                                                                  32
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                  33
              recvtypes(*)
                                                                                  34
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                  35
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  36
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  37
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  39
                                                                                  40
MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                  41
             recvcounts, rdispls, recvtypes, comm, ierror)
                                                                                  42
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                  43
    INTEGER, INTENT(IN) :: sendcounts(*), recvcounts(*)
                                                                                  44
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
                                                                                  45
    TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
                                                                                  46
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  47
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  48
```

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Topo_test(comm, status, ierror)
3
         TYPE(MPI_Comm), INTENT(IN) :: comm
4
         INTEGER, INTENT(OUT) :: status
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
7
8
     A.3.6 MPI Environmental Management Fortran 2008 Bindings
9
    DOUBLE PRECISION MPI_Wtick()
10
11
     DOUBLE PRECISION MPI_Wtime()
12
    MPI_Add_error_class(errorclass, ierror)
13
         INTEGER, INTENT(OUT) :: errorclass
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     MPI_Add_error_code(errorclass, errorcode, ierror)
17
         INTEGER, INTENT(IN) :: errorclass
18
         INTEGER, INTENT(OUT) :: errorcode
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
     MPI_Add_error_string(errorcode, string, ierror)
21
         INTEGER, INTENT(IN) :: errorcode
22
         CHARACTER(LEN=*), INTENT(IN) :: string
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
25
     MPI_Alloc_mem(size, info, baseptr, ierror)
26
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
27
         INTEGER(KIND=MP1_ADDRESS_KIND), INTENT(IN) :: size
28
         TYPE(MPI_Info), INTENT(IN) :: info
29
         TYPE(C_PTR), INTENT(OUT) :: baseptr
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
     MPI_Comm_call_errhandler(comm, errorcode, ierror)
32
33
         TYPE(MPI_Comm), INTENT(IN) :: comm
34
        INTEGER, INTENT(IN) :: errorcode
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
     MPI_Comm_create_errhandler(comm_errhandler_fn, errhandler, ierror)
37
         PROCEDURE(MPI_Comm_errhandler_function), INTENT(IN) ::
38
                    comm_errhandler_fn
39
         TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
     MPI_Comm_get_errhandler(comm, errhandler, ierror)
43
         TYPE(MPI_Comm), INTENT(IN) :: comm
44
         TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
     MPI_Comm_set_errhandler(comm, errhandler, ierror)
47
         TYPE(MPI_Comm), INTENT(IN) :: comm
48
```

TYPE(MPI_Errhandler), INTENT(IN) :: errhandler INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1 2
	3
MPI_Errhandler_free(errhandler, ierror)	4
TYPE(MPI_Errhandler), INTENT(INOUT) :: errhandler	5
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	6
MDI Error alaga(arranada, arranalaga, iarran)	7
MPI_Error_class(errorcode, errorclass, ierror)	8
INTEGER, INTENT(IN) :: errorcode INTEGER, INTENT(OUT) :: errorclass	9
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10
INTEGER, OFFICINAL, INTENT(COT) TEFFOR	11
MPI_Error_string(errorcode, string, resultlen, ierror)	12
INTEGER, INTENT(IN) :: errorcode	13
CHARACTER(LEN=MPI_MAX_ERROR_STRING), INTENT(OUT) :: string	14
INTEGER, INTENT(OUT) :: resultlen	15
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	16
WDT File call emphandles (fb emparede iemer)	17
MPI_File_call_errhandler(fh, errorcode, ierror)	18
TYPE(MPI_File), INTENT(IN) :: fh	19
INTEGER, INTENT(IN) :: errorcode INTEGER, OPTIONAL, INTENT(OUT) :: ierror	20
INIEGER, OPIIONAL, INIENI(UOI) :: IEIIOI	21
MPI_File_create_errhandler(file_errhandler_fn, errhandler, ierror)	22
PROCEDURE(MPI_File_errhandler_function), INTENT(IN) ::	23
file_errhandler_fn	24
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler	25
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	26
MPI_File_get_errhandler(file, errhandler, ierror)	27
TYPE(MPI_File), INTENT(IN) :: file	28
TYPE(MPI_FILe), INTENT(IN) IIIe TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler	29
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	30
INTEGER, OFFICIARE, INTENT(COT) TEFFOR	31
MPI_File_set_errhandler(file, errhandler, ierror)	32
TYPE(MPI_File), INTENT(IN) :: file	33
TYPE(MPI_Errhandler), INTENT(IN) :: errhandler	34
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	35
MPI_Free_mem(base, ierror)	36
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: base	37
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	38
INTEGER, OFITONAL, INTENT(UOI) TETIOT	39
MPI_Get_library_version(version, resultlen, ierror)	40
CHARACTER(LEN=MPI_MAX_LIBRARY_VERSION_STRING), INTENT(OUT) :: version	41
INTEGER, INTENT(OUT) :: resultlen	42
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	43
MPI (ot processor name(name, recultion, ierren)	44
<pre>MPI_Get_processor_name(name, resultlen, ierror)</pre>	45
INTEGER, INTENT(OUT) :: resultlen	46
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	47
INIDOLO, OLITOWAL, INIDAL(UOI/ ICIIOI	48

```
1
     MPI_Get_version(version, subversion, ierror)
\mathbf{2}
         INTEGER, INTENT(OUT) :: version, subversion
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Session_call_errhandler(session, errorcode, ierror)
5
         TYPE(MPI_Session), INTENT(IN) :: session
6
         INTEGER, INTENT(IN) :: errorcode
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
9
     MPI_Session_create_errhandler(session_errhandler_fn, errhandler, ierror)
10
         PROCEDURE(MPI_Session_errhandler_function), INTENT(IN) ::
11
                   session_errhandler_fn
12
         TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
     MPI_Session_get_errhandler(session, errhandler, ierror)
15
         TYPE(MPI_Session), INTENT(IN) :: session
16
         TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
     MPI_Session_set_errhandler(session, errhandler, ierror)
20
         TYPE(MPI_Session), INTENT(IN) :: session
21
         TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
     MPI_Win_call_errhandler(win, errorcode, ierror)
24
         TYPE(MPI_Win), INTENT(IN) :: win
25
         INTEGER, INTENT(IN) :: errorcode
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
     MPI_Win_create_errhandler(win_errhandler_fn, errhandler, ierror)
29
         PROCEDURE (MPI_Win_errhandler_function), INTENT(IN) :: win_errhandler_fn
30
         TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
     MPI_Win_get_errhandler(win, errhandler, ierror)
33
        TYPE(MPI_Win), INTENT(IN) :: win
34
         TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
     MPI_Win_set_errhandler(win, errhandler, ierror)
38
         TYPE(MPI_Win), INTENT(IN) :: win
39
         TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
     A.3.7 The Info Object Fortran 2008 Bindings
43
44
     MPI_Info_create_env(info, ierror)
45
         TYPE(MPI_Info), INTENT(OUT) :: info
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
     MPI_Info_create(info, ierror)
```

1 TYPE(MPI_Info), INTENT(OUT) :: info 2 INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Info_delete(info, key, ierror) TYPE(MPI_Info), INTENT(IN) :: info 5 CHARACTER(LEN=*), INTENT(IN) :: key 6 INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Info_dup(info, newinfo, ierror) TYPE(MPI_Info), INTENT(IN) :: info 10 TYPE(MPI_Info), INTENT(OUT) :: newinfo 11 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 12MPI_Info_free(info, ierror) 13 TYPE(MPI_Info), INTENT(INOUT) :: info 14 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1516MPI_Info_get(info, key, valuelen, value, flag, ierror) 17 TYPE(MPI_Info), INTENT(IN) :: info 18 CHARACTER(LEN=*), INTENT(IN) :: key 19 INTEGER, INTENT(IN) :: valuelen 20CHARACTER(LEN=valuelen), INTENT(OUT) :: value 21LOGICAL, INTENT(OUT) :: flag 22 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 23MPI_Info_get_nkeys(info, nkeys, ierror) 24TYPE(MPI_Info), INTENT(IN) :: info 25INTEGER, INTENT(OUT) :: nkeys 26INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2728 MPI_Info_get_nthkey(info, n, key, ierror) 29 TYPE(MPI_Info), INTENT(IN) :: info 30 INTEGER, INTENT(IN) :: n 31CHARACTER(LEN=*), INTENT(OUT) :: key 32 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 33 MPI_Info_get_string(info, key, buflen, value, flag, ierror) 34 TYPE(MPI_Info), INTENT(IN) :: info 35CHARACTER(LEN=*), INTENT(IN) :: key 36 INTEGER, INTENT(INOUT) :: buflen 37 CHARACTER(LEN=*), INTENT(OUT) :: value 38 LOGICAL, INTENT(OUT) :: flag 39 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 40 41 MPI_Info_get_valuelen(info, key, valuelen, flag, ierror) 42TYPE(MPI_Info), INTENT(IN) :: info 43 CHARACTER(LEN=*), INTENT(IN) :: key 44 INTEGER, INTENT(OUT) :: valuelen 45LOGICAL, INTENT(OUT) :: flag 46INTEGER, OPTIONAL, INTENT(OUT) :: ierror 47MPI_Info_set(info, key, value, ierror) 48

```
1
         TYPE(MPI_Info), INTENT(IN) :: info
2
         CHARACTER(LEN=*), INTENT(IN) :: key, value
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
5
     A.3.8 Process Creation and Management Fortran 2008 Bindings
6
7
     MPI_Abort(comm, errorcode, ierror)
8
         TYPE(MPI_Comm), INTENT(IN) :: comm
9
         INTEGER, INTENT(IN) :: errorcode
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
     MPI_Close_port(port_name, ierror)
12
         CHARACTER(LEN=*), INTENT(IN) :: port_name
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
     MPI_Comm_accept(port_name, info, root, comm, newcomm, ierror)
16
         CHARACTER(LEN=*), INTENT(IN) :: port_name
17
         TYPE(MPI_Info), INTENT(IN) :: info
18
         INTEGER, INTENT(IN) :: root
19
         TYPE(MPI_Comm), INTENT(IN) :: comm
20
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
     MPI_Comm_connect(port_name, info, root, comm, newcomm, ierror)
23
         CHARACTER(LEN=*), INTENT(IN) :: port_name
24
         TYPE(MPI_Info), INTENT(IN) :: info
25
         INTEGER, INTENT(IN) :: root
26
         TYPE(MPI_Comm), INTENT(IN) :: comm
27
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
     MPI_Comm_disconnect(comm, ierror)
31
         TYPE(MPI_Comm), INTENT(INOUT) :: comm
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     MPI_Comm_get_parent(parent, ierror)
         TYPE(MPI_Comm), INTENT(OUT) :: parent
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
     MPI_Comm_join(fd, intercomm, ierror)
38
         INTEGER, INTENT(IN) :: fd
39
         TYPE(MPI_Comm), INTENT(OUT) :: intercomm
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
     MPI_Comm_spawn(command, argv, maxprocs, info, root, comm, intercomm,
                   array_of_errcodes, ierror)
43
44
         CHARACTER(LEN=*), INTENT(IN) :: command, argv(*)
45
         INTEGER, INTENT(IN) :: maxprocs, root
         TYPE(MPI_Info), INTENT(IN) :: info
46
47
         TYPE(MPI_Comm), INTENT(IN) :: comm
         TYPE(MPI_Comm), INTENT(OUT) :: intercomm
48
```

```
1
    INTEGER :: array_of_errcodes(*)
                                                                                   2
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Comm_spawn_multiple(count, array_of_commands, array_of_argv,
              array_of_maxprocs, array_of_info, root, comm, intercomm,
              array_of_errcodes, ierror)
                                                                                   6
    INTEGER, INTENT(IN) :: count, array_of_maxprocs(*), root
    CHARACTER(LEN=*), INTENT(IN) :: array_of_commands(*),
              array_of_argv(count, *)
    TYPE(MPI_Info), INTENT(IN) :: array_of_info(*)
                                                                                   10
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   11
    TYPE(MPI_Comm), INTENT(OUT) :: intercomm
                                                                                   12
    INTEGER :: array_of_errcodes(*)
                                                                                   13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   14
                                                                                   15
MPI_Finalized(flag, ierror)
                                                                                   16
    LOGICAL, INTENT(OUT) :: flag
                                                                                   17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   18
MPI_Finalize(ierror)
                                                                                   19
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   20
                                                                                   21
MPI_Initialized(flag, ierror)
                                                                                   22
    LOGICAL, INTENT(OUT) :: flag
                                                                                   23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   24
MPI_Init(ierror)
                                                                                   25
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   26
                                                                                   27
MPI_Init_thread(required, provided, ierror)
                                                                                   28
    INTEGER, INTENT(IN) :: required
                                                                                   29
    INTEGER, INTENT(OUT) :: provided
                                                                                   30
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   31
MPI_Is_thread_main(flag, ierror)
                                                                                   32
    LOGICAL, INTENT(OUT) :: flag
                                                                                   33
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   34
                                                                                   35
MPI_Lookup_name(service_name, info, port_name, ierror)
                                                                                   36
    CHARACTER(LEN=*), INTENT(IN) :: service_name
                                                                                   37
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   38
    CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name
                                                                                   39
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   40
MPI_Open_port(info, port_name, ierror)
                                                                                   41
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   42
    CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name
                                                                                   43
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   44
                                                                                   45
MPI_Publish_name(service_name, info, port_name, ierror)
                                                                                   46
    CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name
                                                                                   47
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   48
```

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Query_thread(provided, ierror)
3
         INTEGER, INTENT(OUT) :: provided
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
6
     MPI_Session_finalize(session, ierror)
7
         TYPE(MPI_Session), INTENT(INOUT) :: session
8
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
     MPI_Session_get_info(session, info_used, ierror)
10
         TYPE(MPI_Session), INTENT(IN) :: session
11
         TYPE(MPI_Info), INTENT(OUT) :: info_used
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     MPI_Session_get_nth_pset(session, info, n, pset_len, pset_name, ierror)
15
         TYPE(MPI_Session), INTENT(IN) :: session
16
         TYPE(MPI_Info), INTENT(IN) :: info
17
         INTEGER, INTENT(IN) :: n
18
         INTEGER, INTENT(INOUT) :: pset_len
19
         CHARACTER(LEN=*), INTENT(OUT) :: pset_name
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
     MPI_Session_get_num_psets(session, info, npset_names, ierror)
22
         TYPE(MPI_Session), INTENT(IN) :: session
23
         TYPE(MPI_Info), INTENT(IN) :: info
24
         INTEGER, INTENT(OUT) :: npset_names
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27
     MPI_Session_get_pset_info(session, pset_name, info, ierror)
28
         TYPE(MPI_Session), INTENT(IN) :: session
29
         CHARACTER(LEN=*), INTENT(IN) :: pset_name
30
         TYPE(MPI_Info), INTENT(OUT) :: info
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
     MPI_Session_init(info, errhandler, session, ierror)
33
        TYPE(MPI_Info), INTENT(IN) :: info
34
         TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
35
         TYPE(MPI_Session), INTENT(OUT) :: session
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
     MPI_Unpublish_name(service_name, info, port_name, ierror)
39
         CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name
40
         TYPE(MPI_Info), INTENT(IN) :: info
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
     A.3.9 One-Sided Communications Fortran 2008 Bindings
44
45
     MPI_Accumulate(origin_addr, origin_count, origin_datatype, target_rank,
46
                   target_disp, target_count, target_datatype, op, win, ierror)
47
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
48
```

```
1
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
                                                                                  2
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  5
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  6
MPI_Compare_and_swap(origin_addr, compare_addr, result_addr, datatype,
             target_rank, target_disp, win, ierror)
                                                                                  9
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr,
                                                                                  10
              compare_addr
                                                                                  11
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
                                                                                  12
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  13
    INTEGER, INTENT(IN) :: target_rank
                                                                                  14
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                  15
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  16
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  17
                                                                                  18
MPI_Fetch_and_op(origin_addr, result_addr, datatype, target_rank,
                                                                                  19
              target_disp, op, win, ierror)
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
                                                                                  20
                                                                                  21
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
                                                                                  22
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  23
    INTEGER, INTENT(IN) :: target_rank
                                                                                  24
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                  25
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  26
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  27
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  28
MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
                                                                                  29
              result_count, result_datatype, target_rank, target_disp,
                                                                                  30
              target_count, target_datatype, op, win, ierror)
                                                                                  31
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
                                                                                  32
    INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
                                                                                  33
              target_count
                                                                                  34
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
                                                                                  35
              target_datatype
                                                                                  36
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
                                                                                  37
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                  38
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  39
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  40
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  41
                                                                                  42
MPI_Get(origin_addr, origin_count, origin_datatype, target_rank,
                                                                                  43
              target_disp, target_count, target_datatype, win, ierror)
                                                                                  44
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: origin_addr
                                                                                  45
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
                                                                                  46
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
                                                                                  47
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                  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, INTENT(IN) :: origin_count, target_rank, target_count
7
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
8
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
9
         TYPE(MPI_Win), INTENT(IN) :: win
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
    MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
13
                  target_disp, target_count, target_datatype, op, win, request,
14
                  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_Op), INTENT(IN) :: op
20
         TYPE(MPI_Win), INTENT(IN) :: win
21
         TYPE(MPI_Request), INTENT(OUT) :: request
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
    MPI_Rget_accumulate(origin_addr, origin_count, origin_datatype,
24
                  result_addr, result_count, result_datatype, target_rank,
25
                  target_disp, target_count, target_datatype, op, win, request,
26
                  ierror)
27
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
28
         INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
29
                   target_count
30
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
31
                   target_datatype
32
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
33
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
34
         TYPE(MPI_Op), INTENT(IN) :: op
35
         TYPE(MPI_Win), INTENT(IN) :: win
36
         TYPE(MPI_Request), INTENT(OUT) :: request
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
     MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank,
40
                  target_disp, target_count, target_datatype, win, request,
41
                  ierror)
42
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: origin_addr
43
         INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
44
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
45
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
46
         TYPE(MPI_Win), INTENT(IN) :: win
47
         TYPE(MPI_Request), INTENT(OUT) :: request
48
```

1 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2 MPI_Rput(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win, request, ierror) TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr 6 INTEGER, INTENT(IN) :: origin_count, target_rank, target_count TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp 9 TYPE(MPI_Win), INTENT(IN) :: win 10 TYPE(MPI_Request), INTENT(OUT) :: request 11 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1213 MPI_Win_allocate_shared(size, disp_unit, info, comm, baseptr, win, ierror) 14USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR 15INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size 16INTEGER, INTENT(IN) :: disp_unit 17TYPE(MPI_Info), INTENT(IN) :: info 18 TYPE(MPI_Comm), INTENT(IN) :: comm 19 TYPE(C_PTR), INTENT(OUT) :: baseptr TYPE(MPI_Win), INTENT(OUT) :: win 2021INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22 MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror) 23USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR 24INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size 25INTEGER, INTENT(IN) :: disp_unit 26TYPE(MPI_Info), INTENT(IN) :: info 27TYPE(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 3132 MPI_Win_attach(win, base, size, ierror) 33 TYPE(MPI_Win), INTENT(IN) :: win 34 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base 35 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size 36 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 37 MPI_Win_complete(win, ierror) 38 TYPE(MPI_Win), INTENT(IN) :: win 39 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 40 41 MPI_Win_create(base, size, disp_unit, info, comm, win, ierror) 42TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base 43 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size 44 INTEGER, INTENT(IN) :: disp_unit 45TYPE(MPI_Info), INTENT(IN) :: info 46TYPE(MPI_Comm), INTENT(IN) :: comm 47TYPE(MPI_Win), INTENT(OUT) :: win 48

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Win_create_dynamic(info, comm, win, ierror)
3
         TYPE(MPI_Info), INTENT(IN) :: info
4
         TYPE(MPI_Comm), INTENT(IN) :: comm
5
         TYPE(MPI_Win), INTENT(OUT) :: win
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
     MPI_Win_detach(win, base, ierror)
9
         TYPE(MPI_Win), INTENT(IN) :: win
10
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
     MPI_Win_fence(assert, win, ierror)
13
         INTEGER, INTENT(IN) :: assert
14
         TYPE(MPI_Win), INTENT(IN) :: win
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
     MPI_Win_flush_all(win, ierror)
18
         TYPE(MPI_Win), INTENT(IN) :: win
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
     MPI_Win_flush_local_all(win, ierror)
21
         TYPE(MPI_Win), INTENT(IN) :: win
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
24
     MPI_Win_flush_local(rank, win, ierror)
25
         INTEGER, INTENT(IN) :: rank
26
         TYPE(MPI_Win), INTENT(IN) :: win
27
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
     MPI_Win_flush(rank, win, ierror)
29
         INTEGER, INTENT(IN) :: rank
30
         TYPE(MPI_Win), INTENT(IN) :: win
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33
     MPI_Win_free(win, ierror)
34
        TYPE(MPI_Win), INTENT(INOUT) :: win
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
     MPI_Win_get_group(win, group, ierror)
37
         TYPE(MPI_Win), INTENT(IN) :: win
38
         TYPE(MPI_Group), INTENT(OUT) :: group
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     MPI_Win_get_info(win, info_used, ierror)
42
         TYPE(MPI_Win), INTENT(IN) :: win
43
         TYPE(MPI_Info), INTENT(OUT) :: info_used
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
     MPI_Win_lock_all(assert, win, ierror)
46
47
         INTEGER, INTENT(IN) :: assert
         TYPE(MPI_Win), INTENT(IN) :: win
48
```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1
MPI_Win_lock(lock_type, rank, assert, win, ierror)	2
INTEGER, INTENT(IN) :: lock_type, rank, assert	3
TYPE(MPI_Win), INTENT(IN) :: win	4 5
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	6
	7
MPI_Win_post(group, assert, win, ierror)	8
TYPE(MPI_Group), INTENT(IN) :: group INTEGER, INTENT(IN) :: assert	9
TYPE(MPI_Win), INTENT(IN) :: win	10
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	11
	12
MPI_Win_set_info(win, info, ierror)	13
TYPE(MPI_Win), INTENT(IN) :: win	14
TYPE(MPI_Info), INTENT(IN) :: info	15
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	16
MPI_Win_shared_query(win, rank, size, disp_unit, baseptr, ierror)	17
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	18
TYPE(MPI_Win), INTENT(IN) :: win	19
INTEGER, INTENT(IN) :: rank	20 21
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size	22
INTEGER, INTENT(OUT) :: disp_unit	23
TYPE(C_PTR), INTENT(OUT) :: baseptr INTEGER, OPTIONAL, INTENT(OUT) :: ierror	24
INILIALI, BETTONAL, INILAT(001) TETTOT	25
MPI_Win_start(group, assert, win, ierror)	26
TYPE(MPI_Group), INTENT(IN) :: group	27
INTEGER, INTENT(IN) :: assert	28
TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror	29
INTEGER, UPTIONAL, INTENI(U01) :: Terror	30
MPI_Win_sync(win, ierror)	31
TYPE(MPI_Win), INTENT(IN) :: win	32 33
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	34
MPI_Win_test(win, flag, ierror)	35
TYPE(MPI_Win), INTENT(IN) :: win	36
LOGICAL, INTENT(OUT) :: flag	37
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	38
MPI_Win_unlock_all(win, ierror)	39
TYPE(MPI_Win), INTENT(IN) :: win	40
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	41
	42
MPI_Win_unlock(rank, win, ierror)	43
INTEGER, INTENT(IN) :: rank	44
TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror	45 46
INTEGER, OF ITOWAE, INTENT(UOT) TETTOT	46 47
MPI_Win_wait(win, ierror)	48

```
1
         TYPE(MPI_Win), INTENT(IN) :: win
2
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
4
     A.3.10 External Interfaces Fortran 2008 Bindings
5
6
    MPI_Grequest_complete(request, ierror)
7
         TYPE(MPI_Request), INTENT(IN) :: request
8
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
     MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,
10
                   ierror)
11
         PROCEDURE(MPI_Grequest_query_function), INTENT(IN) :: query_fn
12
         PROCEDURE(MPI_Grequest_free_function), INTENT(IN) :: free_fn
13
         PROCEDURE(MPI_Grequest_cancel_function), INTENT(IN) :: cancel_fn
14
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
15
         TYPE(MPI_Request), INTENT(OUT) :: request
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18
     MPI_Status_set_cancelled(status, flag, ierror)
19
         TYPE(MPI_Status), INTENT(INOUT) :: status
20
         LOGICAL, INTENT(IN) :: flag
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
     MPI_Status_set_elements(status, datatype, count, ierror)
23
         TYPE(MPI_Status), INTENT(INOUT) :: status
24
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
25
         INTEGER, INTENT(IN) :: count
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
     MPI_Status_set_elements_x(status, datatype, count, ierror)
29
         TYPE(MPI_Status), INTENT(INOUT) :: status
30
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
31
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     A.3.11 I/O Fortran 2008 Bindings
35
36
     MPI_CONVERSION_FN_NULL(userbuf, datatype, count, filebuf, position,
37
                   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
45
     MPI_File_close(fh, ierror)
46
         TYPE(MPI_File), INTENT(INOUT) :: fh
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

MPI_File_delete(filename, info, ierror) CHARACTER(LEN=*), INTENT(IN) :: filename TYPE(MPI_Info), INTENT(IN) :: info INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_amode(fh, amode, ierror) TYPE(MPI_File), INTENT(IN) :: fh INTEGER, INTENT(OUT) :: amode INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_atomicity(fh, flag, ierror) TYPE(MPI_File), INTENT(IN) :: fh LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_byte_offset(fh, offset, disp, ierror) TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_group(fh, group, ierror) TYPE(MPI_File), INTENT(IN) :: fh TYPE(MPI_Group), INTENT(OUT) :: group INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_info(fh, info_used, ierror) TYPE(MPI_File), INTENT(IN) :: fh TYPE(MPI_Info), INTENT(OUT) :: info_used INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_position(fh, offset, ierror) TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_position_shared(fh, offset, ierror) TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_size(fh, size, ierror) TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: size INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_type_extent(fh, datatype, extent, ierror) TYPE(MPI_File), INTENT(IN) :: fh TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: extent INTEGER, OPTIONAL, INTENT(OUT) :: ierror

1

2

5

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 48

```
1
    MPI_File_get_view(fh, disp, etype, filetype, datarep, ierror)
\mathbf{2}
         TYPE(MPI_File), INTENT(IN) :: fh
3
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp
4
         TYPE(MPI_Datatype), INTENT(OUT) :: etype, filetype
5
         CHARACTER(LEN=*), INTENT(OUT) :: datarep
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
    MPI_File_iread_all(fh, buf, count, datatype, request, ierror)
8
         TYPE(MPI_File), INTENT(IN) :: fh
9
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
10
         INTEGER, INTENT(IN) :: count
11
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
12
         TYPE(MPI_Request), INTENT(OUT) :: request
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
    MPI_File_iread_at_all(fh, offset, buf, count, datatype,
                                                               request, ierror)
16
         TYPE(MPI_File), INTENT(IN) :: fh
17
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
18
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
19
         INTEGER, INTENT(IN) :: count
20
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
21
         TYPE(MPI_Request), INTENT(OUT) :: request
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
    MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror)
24
         TYPE(MPI_File), INTENT(IN) :: fh
25
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
26
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
27
         INTEGER, INTENT(IN) :: count
28
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
         TYPE(MPI_Request), INTENT(OUT) :: request
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32
     MPI_File_iread(fh, buf, count, datatype, request, ierror)
33
         TYPE(MPI_File), INTENT(IN) :: fh
34
        TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
35
         INTEGER, INTENT(IN) :: count
36
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
37
         TYPE(MPI_Request), INTENT(OUT) :: request
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
     MPI_File_iread_shared(fh, buf, count, datatype, request, ierror)
40
         TYPE(MPI_File), INTENT(IN) :: fh
41
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
42
         INTEGER, INTENT(IN) :: count
43
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
44
         TYPE(MPI_Request), INTENT(OUT) :: request
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
     MPI_File_iwrite_all(fh, buf, count, datatype, request, ierror)
48
```

```
TYPE(MPI_File), INTENT(IN) :: fh
                                                                                   1
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                   2
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   5
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   6
MPI_File_iwrite_at_all(fh, offset, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                   9
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                  10
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                  11
    INTEGER, INTENT(IN) :: count
                                                                                  12
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  13
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  14
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  15
                                                                                  16
MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror)
                                                                                  17
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  18
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                  19
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
                                                                                  20
                                                                                  21
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  22
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  ^{24}
MPI_File_iwrite(fh, buf, count, datatype, request, ierror)
                                                                                  25
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  26
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                  27
    INTEGER, INTENT(IN) :: count
                                                                                  28
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  29
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  30
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  31
                                                                                  32
MPI_File_iwrite_shared(fh, buf, count, datatype, request, 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
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  39
MPI_File_open(comm, filename, amode, info, fh, ierror)
                                                                                  40
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  41
    CHARACTER(LEN=*), INTENT(IN) :: filename
                                                                                  42
    INTEGER, INTENT(IN) :: amode
                                                                                  43
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  44
    TYPE(MPI_File), INTENT(OUT) :: fh
                                                                                  45
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  46
                                                                                  47
MPI_File_preallocate(fh, size, ierror)
                                                                                  48
```

```
1
         TYPE(MPI_File), INTENT(IN) :: fh
2
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_File_read_all_begin(fh, buf, count, datatype, ierror)
5
         TYPE(MPI_File), INTENT(IN) :: fh
6
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
7
         INTEGER, INTENT(IN) :: count
8
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
     MPI_File_read_all_end(fh, buf, status, ierror)
12
         TYPE(MPI_File), INTENT(IN) :: fh
13
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
14
         TYPE(MPI_Status) :: status
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
     MPI_File_read_all(fh, buf, count, datatype, status, ierror)
17
         TYPE(MPI_File), INTENT(IN) :: fh
18
         TYPE(*), DIMENSION(..) :: 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_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, INTENT(IN) :: count
29
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
     MPI_File_read_at_all_end(fh, buf, status, ierror)
32
         TYPE(MPI_File), INTENT(IN) :: fh
33
        TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
34
         TYPE(MPI_Status) :: status
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
     MPI_File_read_at_all(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(..) :: buf
41
         INTEGER, INTENT(IN) :: count
42
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
43
         TYPE(MPI_Status) :: status
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
    MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror)
46
         TYPE(MPI_File), INTENT(IN) :: fh
47
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
48
```

```
1
    TYPE(*), DIMENSION(..) :: buf
                                                                                   2
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   5
MPI_File_read(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
    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_ordered_begin(fh, buf, count, datatype, ierror)
                                                                                   15
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                   16
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                   17
    INTEGER, INTENT(IN) :: count
                                                                                   18
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   19
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   20
MPI_File_read_ordered_end(fh, buf, status, ierror)
                                                                                   21
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                   22
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                   23
    TYPE(MPI_Status) :: status
                                                                                   24
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   25
                                                                                   26
MPI_File_read_ordered(fh, buf, count, datatype, status, ierror)
                                                                                   27
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                   28
    TYPE(*), DIMENSION(..) :: buf
                                                                                   29
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   30
                                                                                   31
    TYPE(MPI_Status) :: status
                                                                                   32
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   33
MPI_File_read_shared(fh, buf, count, datatype, status, ierror)
                                                                                   34
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                   35
    TYPE(*), DIMENSION(..) :: buf
                                                                                   36
    INTEGER, INTENT(IN) :: count
                                                                                   37
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   38
    TYPE(MPI_Status) :: status
                                                                                   39
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   40
                                                                                   41
MPI_File_seek(fh, offset, whence, ierror)
                                                                                   42
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                   43
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                   44
    INTEGER, INTENT(IN) :: whence
                                                                                   45
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   46
MPI_File_seek_shared(fh, offset, whence, ierror)
                                                                                   47
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                   48
```

```
1
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
\mathbf{2}
         INTEGER, INTENT(IN) :: whence
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_File_set_atomicity(fh, flag, ierror)
5
         TYPE(MPI_File), INTENT(IN) :: fh
6
         LOGICAL, INTENT(IN) :: flag
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
9
     MPI_File_set_info(fh, info, ierror)
10
         TYPE(MPI_File), INTENT(IN) :: fh
11
         TYPE(MPI_Info), INTENT(IN) :: info
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
     MPI_File_set_size(fh, size, ierror)
14
         TYPE(MPI_File), INTENT(IN) :: fh
15
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18
     MPI_File_set_view(fh, disp, etype, filetype, datarep, info, ierror)
19
         TYPE(MPI_File), INTENT(IN) :: fh
20
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: disp
21
         TYPE(MPI_Datatype), INTENT(IN) :: etype, filetype
22
         CHARACTER(LEN=*), INTENT(IN) :: datarep
23
         TYPE(MPI_Info), INTENT(IN) :: info
24
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
     MPI_File_sync(fh, ierror)
26
         TYPE(MPI_File), INTENT(IN) :: fh
27
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
     MPI_File_write_all_begin(fh, buf, count, datatype, ierror)
30
         TYPE(MPI_File), INTENT(IN) :: fh
31
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
32
         INTEGER, INTENT(IN) :: count
33
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
34
        INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
     MPI_File_write_all_end(fh, buf, status, ierror)
36
         TYPE(MPI_File), INTENT(IN) :: fh
37
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
38
         TYPE(MPI_Status) :: status
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     MPI_File_write_all(fh, buf, count, datatype, status, ierror)
42
         TYPE(MPI_File), INTENT(IN) :: fh
43
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
44
         INTEGER, INTENT(IN) :: count
45
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
46
         TYPE(MPI_Status) :: status
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

```
1
MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror)
                                                                                   2
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
                                                                                   5
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_File_write_at_all_end(fh, buf, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  10
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                  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, 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(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(fh, buf, count, datatype, status, ierror)
                                                                                  32
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  33
    TYPE(*), DIMENSION(...), INTENT(IN) :: buf
                                                                                  34
   INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  35
                                                                                  36
    TYPE(MPI_Status) :: status
                                                                                  37
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  38
MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror)
                                                                                  39
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  40
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                  41
    INTEGER, INTENT(IN) :: count
                                                                                  42
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  43
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  44
                                                                                  45
MPI_File_write_ordered_end(fh, buf, status, ierror)
                                                                                  46
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  47
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                  48
```

```
1
         TYPE(MPI_Status) :: status
\mathbf{2}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_File_write_ordered(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_shared(fh, buf, count, datatype, status, ierror)
12
         TYPE(MPI_File), INTENT(IN) :: fh
13
         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
14
         INTEGER, INTENT(IN) :: count
15
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
16
         TYPE(MPI_Status) :: status
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
     MPI_Register_datarep(datarep, read_conversion_fn, write_conversion_fn,
19
                   dtype_file_extent_fn, extra_state, ierror)
20
         CHARACTER(LEN=*), INTENT(IN) :: datarep
21
         PROCEDURE(MPI_Datarep_conversion_function) :: read_conversion_fn,
22
                   write_conversion_fn
23
         PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn
24
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27
28
     A.3.12 Language Bindings Fortran 2008 Bindings
29
     MPI_F_sync_reg(buf)
30
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
31
32
     MPI_Status_f082f(f08_status, f_status, ierror)
33
         TYPE(MPI_Status), INTENT(IN) :: f08_status
34
         INTEGER, INTENT(OUT) :: f_status(MPI_STATUS_SIZE)
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
     MPI_Status_f2f08(f_status, f08_status, ierror)
37
         INTEGER, INTENT(IN) :: f_status(MPI_STATUS_SIZE)
38
         TYPE(MPI_Status), INTENT(OUT) :: f08_status
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     MPI_Type_create_f90_complex(p, r, newtype, ierror)
42
         INTEGER, INTENT(IN) :: p, r
43
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
     MPI_Type_create_f90_integer(r, newtype, ierror)
46
47
         INTEGER, INTENT(IN) :: r
48
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
```

```
1
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     2
MPI_Type_create_f90_real(p, r, newtype, ierror)
    INTEGER, INTENT(IN) :: p, r
                                                                                     4
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                     5
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     6
MPI_Type_match_size(typeclass, size, datatype, ierror)
    INTEGER, INTENT(IN) :: typeclass, size
    TYPE(MPI_Datatype), INTENT(OUT) :: datatype
                                                                                     a
                                                                                     10
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     11
                                                                                     12
A.3.13 Tools / Profiling Interface Fortran 2008 Bindings
                                                                                     13
                                                                                     14
MPI_Pcontrol(level)
                                                                                     15
    INTEGER, INTENT(IN) :: level
                                                                                     16
                                                                                     17
        Deprecated Fortran 2008 Bindings
                                                                                     18
A.3.14
                                                                                     19
MPI_Info_get(info, key, valuelen, value, flag, ierror)
                                                                                     20
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                     21
    CHARACTER(LEN=*), INTENT(IN) :: key
                                                                                     22
    INTEGER, INTENT(IN) :: valuelen
                                                                                     23
    CHARACTER(LEN=valuelen), INTENT(OUT) :: value
                                                                                     24
    LOGICAL, INTENT(OUT) :: flag
                                                                                     25
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     26
MPI_Info_get_valuelen(info, key, valuelen, flag, ierror)
                                                                                     27
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                     28
    CHARACTER(LEN=*), INTENT(IN) :: key
                                                                                     29
                                                                                     30
    INTEGER, INTENT(OUT) :: valuelen
                                                                                     31
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     32
                                                                                     33
MPI_Sizeof(x, size, ierror)
                                                                                     34
    TYPE(*), DIMENSION(..) :: x
                                                                                     35
    INTEGER, INTENT(OUT) :: size
                                                                                     36
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     37
                                                                                     38
                                                                                     39
                                                                                     40
                                                                                     41
                                                                                     42
                                                                                     43
                                                                                     44
                                                                                     45
                                                                                     46
```

47 48

Fortran Bindings with mpif.h or the mpi Module 1 A.4 $\mathbf{2}$ A.4.1 Point-to-Point Communication Fortran Bindings 3 4 Fortran binding 5MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) 6 <type> BUF(*) 7 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR 8 9 MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 10<type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 11 12MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR) 13 <type> BUFFER(*) 14INTEGER SIZE, IERROR 1516MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR) 17<type> BUFFER_ADDR(*) 18 INTEGER SIZE, IERROR 19MPI_CANCEL(REQUEST, IERROR) 20INTEGER REQUEST, IERROR 2122MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR) 23INTEGER STATUS (MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR 24 MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 25<type> BUF(*) 26INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 2728MPI_IMPROBE(SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS, IERROR) 29INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR 30 LOGICAL FLAG 31 MPI_IMRECV(BUF, COUNT, DATATYPE, MESSAGE, REQUEST, IERROR) 32 <type> BUF(*) 33 INTEGER COUNT, DATATYPE, MESSAGE, REQUEST, IERROR 34 35 MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR) 36 INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR 37 LOGICAL FLAG 38 MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR) 39 <type> BUF(*) 40INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR 41 42MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 43 <type> BUF(*) 44INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 45MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 46 <type> BUF(*) 47INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 48

MPI_ISENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, REQUEST, IERROR)	1 2
<type> BUF(*)</type>	3
	4
INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, REQUEST,	5
IERROR	
MPI_ISENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,	6
RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, REQUEST, IERROR)	7
<type> SENDBUF(*), RECVBUF(*)</type>	8
INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE,	9
SOURCE, RECVTAG, COMM, REQUEST, IERROR	10
	11
MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	12
<type> BUF(*)</type>	13
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR	14
	15
MPI_MPROBE(SOURCE, TAG, COMM, MESSAGE, STATUS, IERROR)	16
INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR	17
MPI_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR)	18
<type> BUF(*)</type>	19
INTEGER COUNT, DATATYPE, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR	20
	21
MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)	22
INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR	23
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)	24
<pre><type> BUF(*)</type></pre>	25
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),	26
INTEGER COONT, DRINTIE, DUCKOE, IRG, COMM, DIRIOD(INT_DIRIOD_DIZE), IERROR	27
	28
MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)	29
<type> BUF(*)</type>	30
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR	31
MPI_REQUEST_FREE(REQUEST, IERROR)	32
INTEGER REQUEST, IERROR	33
INTEGER REQUEST, TERRUR	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(*)</type>	40
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR	41
MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	42
<pre><type> BUF(*)</type></pre>	43
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR	44
	45
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)	46
<type> BUF(*)</type>	47
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR	48

```
1
    MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
\mathbf{2}
         <type> BUF(*)
3
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
4
     MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
5
                   COMM, STATUS, IERROR)
6
         <type> BUF(*)
7
         INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM,
8
                   STATUS(MPI_STATUS_SIZE), IERROR
9
10
     MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,
11
                   RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
12
         <type> SENDBUF(*), RECVBUF(*)
13
         INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE,
14
                   SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
15
     MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
16
         <type> BUF(*)
17
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
18
19
     MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
20
         <type> BUF(*)
21
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
22
     MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)
23
         INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR
24
25
    MPI_START(REQUEST, IERROR)
26
         INTEGER REQUEST, IERROR
27
     MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR)
28
         INTEGER COUNT, ARRAY_OF_REQUESTS(*), ARRAY_OF_STATUSES(MPI_STATUS_SIZE,
29
                   *). IERROR
30
         LOGICAL FLAG
31
32
     MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)
33
         INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
34
                   IERROR
35
         LOGICAL FLAG
36
     MPI_TEST_CANCELLED(STATUS, FLAG, IERROR)
37
         INTEGER STATUS(MPI_STATUS_SIZE), IERROR
38
         LOGICAL FLAG
39
40
     MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
41
         INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
42
         LOGICAL FLAG
43
    MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
44
                   ARRAY_OF_STATUSES, IERROR)
45
         INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
46
                   ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *), IERROR
47
48
```

MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR) 1 $\mathbf{2}$ INTEGER COUNT, ARRAY_OF_REQUESTS(*), ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *), IERROR MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR) 5 INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE), 6 IERROR MPI_WAIT(REQUEST, STATUS, IERROR) INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR 10 MPI WAITSOME (INCOUNT, ARRAY OF REQUESTS, OUTCOUNT, ARRAY OF INDICES, 11 ARRAY_OF_STATUSES, IERROR) 12INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*), 13 ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *), IERROR 141516A.4.2 Datatypes Fortran Bindings 17INTEGER(KIND=MPI_ADDRESS_KIND) MPI_AINT_ADD(BASE, DISP) 18 INTEGER(KIND=MPI_ADDRESS_KIND) BASE, DISP 19 20INTEGER(KIND=MPI_ADDRESS_KIND) MPI_AINT_DIFF(ADDR1, ADDR2) 21INTEGER(KIND=MPI_ADDRESS_KIND) ADDR1, ADDR2 22 MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR) 23<type> LOCATION(*) 24 INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS 25INTEGER IERROR 2627MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, 28POSITION, IERROR) 29 CHARACTER*(*) DATAREP 30 <type> INBUF(*), OUTBUF(*) 31INTEGER INCOUNT, DATATYPE, IERROR 32 INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, POSITION 33 MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR) 34 CHARACTER*(*) DATAREP 35 INTEGER INCOUNT, DATATYPE, IERROR 36 INTEGER(KIND=MPI_ADDRESS_KIND) SIZE 37 38 MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR) 39 <type> INBUF(*), OUTBUF(*) 40 INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR 41 42MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR) INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR 43 44MPI_TYPE_COMMIT(DATATYPE, IERROR) 45INTEGER DATATYPE, IERROR 4647MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR) 48

1	INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
2 3 4	MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS, ARRAY_OF_GSIZES, ARRAY_OF_DISTRIBS, ARRAY_OF_DARGS, ARRAY_OF_PSIZES, ORDER, OLDTYPE, NEWTYPE, IERROR)
5 6 7 8	<pre>INTEGER SIZE, RANK, NDIMS, ARRAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*), ARRAY_OF_DARGS(*), ARRAY_OF_PSIZES(*), ORDER, OLDTYPE, NEWTYPE, IERROR</pre>
9 10 11 12 13	<pre>MPI_TYPE_CREATE_HINDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,</pre>
13 14 15 16 17	<pre>MPI_TYPE_CREATE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,</pre>
18 19 20 21	MPI_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE
22 23 24 25 26	<pre>MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,</pre>
27 28 29 30	MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR) INTEGER OLDTYPE, NEWTYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT
31 32 33 34 35	<pre>MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,</pre>
36 37 38 39 40	<pre>MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES, ARRAY_OF_SUBSIZES, ARRAY_OF_STARTS, ORDER, OLDTYPE, NEWTYPE, IERROR) INTEGER NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*), ARRAY_OF_STARTS(*), ORDER, OLDTYPE, NEWTYPE, IERROR</pre>
41 42	MPI_TYPE_DUP(OLDTYPE, NEWTYPE, IERROR) INTEGER OLDTYPE, NEWTYPE, IERROR
43 44 45	MPI_TYPE_FREE(DATATYPE, IERROR) INTEGER DATATYPE, IERROR
46 47 48	MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES, ARRAY_OF_INTEGERS, ARRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES, IERROR)

INTEGER DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,	1
ARRAY_OF_INTEGERS(*), ARRAY_OF_DATATYPES(*), IERROR	2
<pre>INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_ADDRESSES(*)</pre>	3
MPI_TYPE_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)	4
INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR	5
	6 7
MPI_TYPE_GET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)	8
INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR INTEGER(KIND=MPI_COUNT_KIND) COUNT	9
INTEGER(KIND=MF1_COUNT_KIND) COUNT	10
MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES,	11
COMBINER, IERROR)	12
INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER,	13
IERROR	14
MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR)	15
INTEGER DATATYPE, IERROR	16
INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT	17
MPI_TYPE_GET_EXTENT_X(DATATYPE, LB, EXTENT, IERROR)	18 19
INTEGER DATATYPE, IERROR	20
INTEGER(KIND=MPI_COUNT_KIND) LB, EXTENT	21
MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)	22
INTEGER DATATYPE, IERROR	23
INTEGER(KIND=MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT	24
	25
MPI_TYPE_GET_TRUE_EXTENT_X(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)	26
INTEGER DATATYPE, IERROR	27
INTEGER(KIND=MPI_COUNT_KIND) TRUE_LB, TRUE_EXTENT	28
MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,	29 30
OLDTYPE, NEWTYPE, IERROR)	30 31
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),	32
OLDTYPE, NEWTYPE, IERROR	33
MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)	34
INTEGER DATATYPE, SIZE, IERROR	35
MPI_TYPE_SIZE_X(DATATYPE, SIZE, IERROR)	36
INTEGER DATATYPE, IERROR	37
INTEGER(KIND=MPI_COUNT_KIND) SIZE	38
	39
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)	40 41
INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR	41
MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT,	43
DATATYPE, IERROR)	44
CHARACTER*(*) DATAREP	45
<pre><type> INBUF(*), OUTBUF(*) INTEGEP(KIND-MDI ADDRESS KIND) INSIZE DOSITION</type></pre>	46
INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION INTEGER OUTCOUNT, DATATYPE, IERROR	47
INTEGER OUTOONT, DATATILE, IERROR	48

1MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM, $\mathbf{2}$ IERROR) 3 <type> INBUF(*), OUTBUF(*) 4 INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR 56 A.4.3 Collective Communication Fortran Bindings 7 8 MPI_ALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 9 RECVTYPE, COMM, INFO, REQUEST, IERROR) 10 <type> SENDBUF(*), RECVBUF(*) 11 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, 12IERROR 13 MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 14 COMM, IERROR) 15<type> SENDBUF(*), RECVBUF(*) 16 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR 1718 MPI_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, 19 DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR) 20<type> SENDBUF(*), RECVBUF(*) 21INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, 22 INFO, REQUEST, IERROR 23MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, 24 RECVTYPE, COMM, IERROR) 25<type> SENDBUF(*), RECVBUF(*) 26INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE. COMM. 27IERROR 2829MPI_ALLREDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, 30 REQUEST, IERROR) 31<type> SENDBUF(*), RECVBUF(*) 32 INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR 33 34 MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) 35 INTEGER COUNT, DATATYPE, OP, COMM, IERROR 36 37 MPI_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 38 RECVTYPE, COMM, INFO, REQUEST, IERROR) 39 <type> SENDBUF(*), RECVBUF(*) 40 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, 41 IERROR 42MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 43 44 COMM, IERROR) 45<type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR 4647 48

MPI_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,	1
RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)	2
<type> SENDBUF(*), RECVBUF(*)</type>	3
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),	4
RECVTYPE, COMM, INFO, REQUEST, IERROR	5
MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,	6
RDISPLS, RECVTYPE, COMM, IERROR)	7 8
<type> SENDBUF(*), RECVBUF(*)</type>	9
<pre>INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),</pre>	10
RECVTYPE, COMM, IERROR	11
MDT ALLTOALLU INIT (GENDDIE GENDCOUNTS CDICE GENDTYDES DECUDIE	12
MPI_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,	13
RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST, IERROR)	14
<pre><type> SENDBUF(*), RECVBUF(*) </type></pre>	15
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),	16
RDISPLS(*), RECVTYPES(*), COMM, INFO, REQUEST, IERROR	17
MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS,	18
RDISPLS, RECVTYPES, COMM, IERROR)	19
<type> SENDBUF(*), RECVBUF(*)</type>	20
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),	20 21
RDISPLS(*), RECVTYPES(*), COMM, IERROR	21
	22
MPI_BARRIER(COMM, IERROR)	23 24
INTEGER COMM, IERROR	24 25
MPI_BARRIER_INIT(COMM, INFO, REQUEST, IERROR)	25 26
INTEGER COMM, INFO, REQUEST, IERROR	
	27
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)	28 29
<type> BUFFER(*)</type>	29 30
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR	
MPI_BCAST_INIT(BUFFER, COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR)	31 32
<pre>suppe> BUFFER(*)</pre>	32
INTEGER COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR	33 34
MPI_EXSCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,	35 36
IERROR)	30 37
<type> SENDBUF(*), RECVBUF(*)</type>	38
INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR	39
MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)	40
<pre><type> SENDBUF(*), RECVBUF(*)</type></pre>	40
INTEGER COUNT, DATATYPE, OP, COMM, IERROR	41
INIDER COORT, DATATILE, OF, COURT, IERGOR	
MPI_GATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,	43 44
ROOT, COMM, INFO, REQUEST, IERROR)	44 45
<type> SENDBUF(*), RECVBUF(*)</type>	45 46
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO,	40 47
REQUEST, IERROR	47 48
	48

1 MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, $\mathbf{2}$ ROOT, COMM, IERROR) 3 <type> SENDBUF(*), RECVBUF(*) 4 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR 5MPI_GATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, 6 RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR) 7 <type> SENDBUF(*), RECVBUF(*) 8 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT, 9 COMM, INFO, REQUEST, IERROR 1011MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, 12RECVTYPE, ROOT, COMM, IERROR) 13 <type> SENDBUF(*), RECVBUF(*) 14INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT, 15COMM. IERROR 16MPI_IALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 17COMM, REQUEST, IERROR) 18 <type> SENDBUF(*), RECVBUF(*) 19 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR 2021MPI_IALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, 22 RECVTYPE, COMM, REQUEST, IERROR) 23<type> SENDBUF(*), RECVBUF(*) 24 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, 25REQUEST, IERROR 26MPI_IALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, 27IERROR) 28<type> SENDBUF(*), RECVBUF(*) 29 INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR 30 31MPI_IALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 32 COMM, REQUEST, IERROR) 33 <type> SENDBUF(*), RECVBUF(*) 34 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR 35 MPI_IALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, 36 RDISPLS, RECVTYPE, COMM, REQUEST, IERROR) 37 <type> SENDBUF(*), RECVBUF(*) 38 INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), 39 RECVTYPE, COMM, REQUEST, IERROR 4041 MPI_IALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, 42RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR) 43 <type> SENDBUF(*), RECVBUF(*) 44 INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*), 45 RDISPLS(*), RECVTYPES(*), COMM, REQUEST, IERROR 46MPI_IBARRIER(COMM, REQUEST, IERROR) 47INTEGER COMM, REQUEST, IERROR 48

MPI_IBCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR) <type> BUFFER(*)</type>	1 2
INTEGER COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR	3
MPI_IEXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)	4
<pre><type> SENDBUF(*), RECVBUF(*)</type></pre>	5
INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR	6
	7
MPI_IGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,	8
ROOT, COMM, REQUEST, IERROR)	9
<type> SENDBUF(*), RECVBUF(*)</type>	10 11
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,	11
IERROR	12
MPI_IGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,	14
RECVTYPE, ROOT, COMM, REQUEST, IERROR)	15
<type> SENDBUF(*), RECVBUF(*)</type>	16
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,	17
COMM, REQUEST, IERROR	18
MPI_IREDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,	19
REQUEST, IERROR)	20
<type> SENDBUF(*), RECVBUF(*)</type>	21
INTEGER RECVCOUNT, DATATYPE, OP, COMM, REQUEST, IERROR	22
MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,	23 24
REQUEST, IERROR)	24
<type> SENDBUF(*), RECVBUF(*)</type>	26
INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, REQUEST, IERROR	27
MPI_IREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST,	28
IERROR)	29
<type> SENDBUF(*), RECVBUF(*)</type>	30
INTEGER COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, IERROR	31
	32
MPI_ISCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)	33
<pre><type> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR</type></pre>	34
INTEGER COONT, DATATILE, OF, COMM, REQUEST, TERROR	35 36
MPI_ISCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,	37
ROOT, COMM, REQUEST, IERROR)	38
<type> SENDBUF(*), RECVBUF(*)</type>	39
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,	40
IERROR	41
MPI_ISCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,	42
RECVTYPE, ROOT, COMM, REQUEST, IERROR)	43
<type> SENDBUF(*), RECVBUF(*)</type>	44
INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,	45
COMM, REQUEST, IERROR	46
MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR)	47
	48

```
1
         INTEGER OP, IERROR
\mathbf{2}
         LOGICAL COMMUTE
3
     MPI_OP_CREATE(USER_FN, COMMUTE, OP, IERROR)
4
         EXTERNAL USER_FN
5
         LOGICAL COMMUTE
6
         INTEGER OP, IERROR
7
8
     MPI_OP_FREE(OP, IERROR)
9
         INTEGER OP, IERROR
10
     MPI_REDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, INFO,
11
                   REQUEST, IERROR)
12
         <type> SENDBUF(*), RECVBUF(*)
13
         INTEGER COUNT, DATATYPE, OP, ROOT, COMM, INFO, REQUEST, IERROR
14
15
     MPI_REDUCE_LOCAL (INBUF, INOUTBUF, COUNT, DATATYPE, OP, IERROR)
16
         <type> INBUF(*), INOUTBUF(*)
17
         INTEGER COUNT, DATATYPE, OP, IERROR
18
    MPI_REDUCE_SCATTER_BLOCK_INIT(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP,
19
                   COMM, INFO, REQUEST, IERROR)
20
         <type> SENDBUF(*), RECVBUF(*)
21
         INTEGER RECVCOUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
22
23
     MPI_REDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
24
                   IERROR)
25
         <type> SENDBUF(*), RECVBUF(*)
26
         INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR
27
     MPI_REDUCE_SCATTER_INIT(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
28
                   INFO, REQUEST, IERROR)
29
         <type> SENDBUF(*), RECVBUF(*)
30
         INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, INFO, REQUEST, IERROR
31
32
     MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
33
                   IERROR)
34
         <type> SENDBUF(*), RECVBUF(*)
35
         INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR
36
     MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
37
         <type> SENDBUF(*), RECVBUF(*)
38
         INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
39
40
     MPI_SCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,
41
                   IERROR)
42
         <type> SENDBUF(*), RECVBUF(*)
43
         INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
44
     MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
45
         <type> SENDBUF(*), RECVBUF(*)
46
         INTEGER COUNT, DATATYPE, OP, COMM, IERROR
47
48
```

	-
MPI_SCATTER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,	1
RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)	2
<type> SENDBUF(*), RECVBUF(*)</type>	3
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO,	4
REQUEST, IERROR	5
	6
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,	7
ROOT, COMM, IERROR)	8
<type> SENDBUF(*), RECVBUF(*)</type>	9
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR	10
MPI_SCATTERV_INIT(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF,	11
RECVCOUNT, RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)	12
<type> SENDBUF(*), RECVBUF(*)</type>	13
INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,	14
COMM, INFO, REQUEST, IERROR	15
COMP, INFO, REQUEST, TERROR	16
MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,	17
RECVTYPE, ROOT, COMM, IERROR)	18
<type> SENDBUF(*), RECVBUF(*)</type>	19
INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,	20
COMM, IERROR	21
	22
	23
A.4.4 Groups, Contexts, Communicators, and Caching Fortran Bindings	20
MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)	25
INTEGER COMM1, COMM2, RESULT, IERROR	26
	27
MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)	28
INTEGER COMM, GROUP, NEWCOMM, IERROR	20
MPI_COMM_CREATE_FROM_GROUP(GROUP, STRINGTAG, INFO, ERRHANDLER, NEWCOMM,	30
IERROR)	31
INTEGER GROUP, INFO, ERRHANDLER, NEWCOMM, IERROR	32
CHARACTER*(*) STRINGTAG	33
CHARACIER*(*) SIRINGIAG	
MPI_COMM_CREATE_GROUP(COMM, GROUP, TAG, NEWCOMM, IERROR)	34
INTEGER COMM, GROUP, TAG, NEWCOMM, IERROR	35
	36
MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL,	37
EXTRA_STATE, IERROR)	38
EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN	39
INTEGER COMM_KEYVAL, IERROR	40
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	41
MPI_COMM_DELETE_ATTR(COMM, COMM_KEYVAL, IERROR)	42
INTEGER COMM, COMM_KEYVAL, IERROR	43
	44
MPI_COMM_DUP(COMM, NEWCOMM, IERROR)	45
INTEGER COMM, NEWCOMM, IERROR	46
	47
	48

```
1
     MPI_COMM_DUP_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
\mathbf{2}
                   ATTRIBUTE_VAL_OUT, FLAG, IERROR)
3
         INTEGER OLDCOMM, COMM_KEYVAL, IERROR
4
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
5
                    ATTRIBUTE_VAL_OUT
6
         LOGICAL FLAG
7
     MPI_COMM_DUP_WITH_INFO(COMM, INFO, NEWCOMM, IERROR)
8
         INTEGER COMM, INFO, NEWCOMM, IERROR
9
10
     MPI_COMM_FREE(COMM, IERROR)
11
         INTEGER COMM, IERROR
12
     MPI_COMM_FREE_KEYVAL(COMM_KEYVAL, IERROR)
13
         INTEGER COMM_KEYVAL, IERROR
14
15
     MPI_COMM_GET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
16
         INTEGER COMM, COMM_KEYVAL, IERROR
17
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
18
         LOGICAL FLAG
19
    MPI_COMM_GET_INFO(COMM, INFO_USED, IERROR)
20
         INTEGER COMM, INFO_USED, IERROR
21
22
     MPI_COMM_GET_NAME(COMM, COMM_NAME, RESULTLEN, IERROR)
23
         INTEGER COMM, RESULTLEN, IERROR
^{24}
         CHARACTER*(*) COMM_NAME
25
     MPI_COMM_GROUP(COMM, GROUP, IERROR)
26
         INTEGER COMM, GROUP, IERROR
27
28
     MPI_COMM_IDUP(COMM, NEWCOMM, REQUEST, IERROR)
29
         INTEGER COMM, NEWCOMM, REQUEST, IERROR
30
     MPI_COMM_IDUP_WITH_INFO(COMM, INFO, NEWCOMM, REQUEST, IERROR)
^{31}
         INTEGER COMM, INFO, NEWCOMM, REQUEST, IERROR
32
33
     MPI_COMM_NULL_COPY_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
34
                   ATTRIBUTE_VAL_OUT, FLAG, IERROR)
35
         INTEGER OLDCOMM, COMM_KEYVAL, IERROR
36
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
37
                   ATTRIBUTE_VAL_OUT
38
         LOGICAL FLAG
39
     MPI_COMM_NULL_DELETE_FN(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE,
40
                   IERROR)
41
         INTEGER COMM, COMM_KEYVAL, IERROR
42
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
43
44
     MPI_COMM_RANK(COMM, RANK, IERROR)
45
         INTEGER COMM, RANK, IERROR
46
47
     MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR)
48
         INTEGER COMM, GROUP, IERROR
```

MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR) INTEGER COMM, SIZE, IERROR	1 2
MPI_COMM_SET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, IERROR)	3 4
INTEGER COMM, COMM_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL	5 6
	7
MPI_COMM_SET_INFO(COMM, INFO, IERROR)	8
INTEGER COMM, INFO, IERROR	9
MPI_COMM_SET_NAME(COMM, COMM_NAME, IERROR)	10
INTEGER COMM, IERROR	11
CHARACTER*(*) COMM_NAME	12
	13
MPI_COMM_SIZE(COMM, SIZE, IERROR) INTEGER COMM, SIZE, IERROR	14
MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)	15
INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR	16
INTEGER COMM, COLOR, REF, NEWCOMM, TERROR	17
MPI_COMM_SPLIT_TYPE(COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR)	18
INTEGER COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR	19
MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)	20
INTEGER COMM, IERROR	21
LOGICAL FLAG	22
LUGICAL FLAG	23 24
MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR)	24 25
INTEGER GROUP1, GROUP2, RESULT, IERROR	26
MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)	27
INTEGER GROUP1, GROUP2, NEWGROUP, IERROR	28
	29
MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)	30
INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR	31
MPI_GROUP_FREE(GROUP, IERROR)	32
INTEGER GROUP, IERROR	33
	34
MPI_GROUP_FROM_SESSION_PSET(SESSION, PSET_NAME, NEWGROUP, IERROR)	35
INTEGER SESSION, NEWGROUP, IERROR	36
CHARACTER*(*) PSET_NAME	37
MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)	38
INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR	39
MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR)	40
INTEGER GROUP1, GROUP2, NEWGROUP, IERROR	41
INTEGER GROOPI, GROOPZ, NEWGROOP, IERROR	42
MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR)	43
INTEGER GROUP, N, RANGES(3, *), NEWGROUP, IERROR	44
MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)	45
INTEGER GROUP, N, RANGES(3, *), NEWGROUP, IERROR	46
	47
	48

1	MPI_GROUP_RANK(GROUP, RANK, IERROR)
2	INTEGER GROUP, RANK, IERROR
3 4 5	MPI_GROUP_SIZE(GROUP, SIZE, IERROR) INTEGER GROUP, SIZE, IERROR
6 7 8	MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR) INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR
9	MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR)
10	INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
11	MPI_INTERCOMM_CREATE_FROM_GROUPS(LOCAL_GROUP, LOCAL_LEADER, REMOTE_GROUP,
12	REMOTE_LEADER, STRINGTAG, INFO, ERRHANDLER, NEWINTERCOMM,
13	IERROR)
14	INTEGER LOCAL_GROUP, LOCAL_LEADER, REMOTE_GROUP, REMOTE_LEADER, INFO,
15	ERRHANDLER, NEWINTERCOMM, IERROR
16	CHARACTER*(*) STRINGTAG
17	MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER,
18	TAG, NEWINTERCOMM, IERROR)
19	INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
20 21 22	NEWINTERCOMM, IERROR
23	MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR)
24	INTEGER INTERCOMM, NEWINTRACOMM, IERROR
25	LOGICAL HIGH
25 26 27 28 29 30	<pre>MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL, EXTRA_STATE, IERROR) EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN INTEGER TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE</pre>
31 32 33	MPI_TYPE_DELETE_ATTR(DATATYPE, TYPE_KEYVAL, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR
34	<pre>MPI_TYPE_DUP_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,</pre>
35	ATTRIBUTE_VAL_OUT, FLAG, IERROR)
36	INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
37	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
38	ATTRIBUTE_VAL_OUT
39	LOGICAL FLAG
40 41 42	MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR) INTEGER TYPE_KEYVAL, IERROR
43	MPI_TYPE_GET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
44	INTEGER DATATYPE, TYPE_KEYVAL, IERROR
45	INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
46	LOGICAL FLAG
47 48	MPI_TYPE_GET_NAME(DATATYPE, TYPE_NAME, RESULTLEN, IERROR)

	1
INTEGER DATATYPE, RESULTLEN, IERROR CHARACTER*(*) TYPE_NAME	2
CHARACTER*(*) TITE_NAME	3
MPI_TYPE_NULL_COPY_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	4
ATTRIBUTE_VAL_OUT, FLAG, IERROR)	5
INTEGER OLDTYPE, TYPE_KEYVAL, IERROR	6
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	7
ATTRIBUTE_VAL_OUT	8
LOGICAL FLAG	9
MPI_TYPE_NULL_DELETE_FN(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE,	10
IERROR)	11
INTEGER DATATYPE, TYPE_KEYVAL, IERROR	12
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	13
MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)	14 15
INTEGER DATATYPE, TYPE_KEYVAL, IERROR	16
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL	17
MDT TYDE CET NAME (DATATYDE TYDE NAME TEDDOD)	18
MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR) INTEGER DATATYPE, IERROR	19
CHARACTER*(*) TYPE_NAME	20
	21
MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,	22
EXTRA_STATE, IERROR)	23
EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN	24
INTEGER WIN_KEYVAL, IERROR	25
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	26
MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR)	27
INTEGER WIN, WIN_KEYVAL, IERROR	28 29
MPI_WIN_DUP_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	29 30
ATTRIBUTE_VAL_OUT, FLAG, IERROR)	31
INTEGER OLDWIN, WIN_KEYVAL, IERROR	32
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	33
ATTRIBUTE_VAL_OUT	34
LOGICAL FLAG	35
MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)	36
INTEGER WIN_KEYVAL, IERROR	37
	38
MPI_WIN_GET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)	39
INTEGER WIN, WIN_KEYVAL, IERROR	40
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL LOGICAL FLAG	41 42
FORICAL LEAG	42 43
MPI_WIN_GET_NAME(WIN, WIN_NAME, RESULTLEN, IERROR)	43 44
INTEGER WIN, RESULTLEN, IERROR	45
CHARACTER*(*) WIN_NAME	46
MPI_WIN_NULL_COPY_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	47
ATTRIBUTE_VAL_OUT, FLAG, IERROR)	48

```
1
         INTEGER OLDWIN, WIN_KEYVAL, IERROR
\mathbf{2}
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
3
                    ATTRIBUTE_VAL_OUT
4
         LOGICAL FLAG
5
     MPI_WIN_NULL_DELETE_FN(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)
6
         INTEGER WIN, WIN_KEYVAL, IERROR
7
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
8
9
     MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR)
10
         INTEGER WIN, WIN_KEYVAL, IERROR
11
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
12
     MPI_WIN_SET_NAME(WIN, WIN_NAME, IERROR)
13
         INTEGER WIN, IERROR
14
         CHARACTER*(*) WIN_NAME
15
16
17
     A.4.5 Process Topologies Fortran Bindings
18
     MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
19
         INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR
20
21
     MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
22
         INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
23
         LOGICAL PERIODS(*), REORDER
24
     MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
25
26
         INTEGER COMM, NDIMS, IERROR
27
     MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
28
         INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
29
         LOGICAL PERIODS(*)
30
     MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)
31
         INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR
32
33
         LOGICAL PERIODS(*)
34
     MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
35
         INTEGER COMM, COORDS(*), RANK, IERROR
36
37
    MPI_CART_SHIFT (COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)
         INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR
38
39
     MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)
40
         INTEGER COMM, NEWCOMM, IERROR
41
         LOGICAL REMAIN_DIMS(*)
42
43
     MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
44
         INTEGER NNODES, NDIMS, DIMS(*), IERROR
45
     MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS,
46
                   OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER,
47
                   COMM_DIST_GRAPH, IERROR)
48
```

<pre>INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE,</pre>	1
<pre>DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH,</pre>	2
IERROR	3
LOGICAL REORDER	4
	5
MPI_DIST_GRAPH_CREATE(COMM_OLD, N, SOURCES, DEGREES, DESTINATIONS, WEIGHTS,	6
INFO, REORDER, COMM_DIST_GRAPH, IERROR)	7
<pre>INTEGER COMM_OLD, N, SOURCES(*), DEGREES(*), DESTINATIONS(*),</pre>	8
WEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR	9
LOGICAL REORDER	10
MPI_DIST_GRAPH_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, SOURCEWEIGHTS,	11
MAXOUTDEGREE, DESTINATIONS, DESTWEIGHTS, IERROR)	12
INTEGER COMM, MAXINDEGREE, SOURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE,	13
DESTINATIONS(*), DESTWEIGHTS(*), IERROR	14
DESIINATIONS(*), DESIWEIGHIS(*), IERRUR	15
MPI_DIST_GRAPH_NEIGHBORS_COUNT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR)	16
INTEGER COMM, INDEGREE, OUTDEGREE, IERROR	17
LOGICAL WEIGHTED	18
	19
MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH,	20
IERROR)	20
INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR	21
LOGICAL REORDER	
MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR)	23
INTEGER COMM, NNODES, NEDGES, IERROR	24
INTEGER COMP, MODED; NEDGED; IERROR	25
MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR)	26
INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR	27
MDT CDADII MAD (COMM NNODEC TNDEY EDCEC NEUDANIK TEDDOD)	28
MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR)	29
INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR	30
MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)	31
INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR	32
	33
MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)	34
INTEGER COMM, RANK, NNEIGHBORS, IERROR	35
MPI_INEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,	36
RECVTYPE, COMM, REQUEST, IERROR)	37
<type> SENDBUF(*), RECVBUF(*)</type>	38
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR	39
INITALI DEMOCIONI, DEMOTILE, RECOCCONI, RECOTILE, COMI, REQUEDI, ILIROR	40
MPI_INEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,	41
DISPLS, RECVTYPE, COMM, REQUEST, IERROR)	42
<type> SENDBUF(*), RECVBUF(*)</type>	43
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,	44
REQUEST, IERROR	45
	46
MPI_INEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,	47
RECVTYPE, COMM, REQUEST, IERROR)	48

```
1
         <type> SENDBUF(*), RECVBUF(*)
\mathbf{2}
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
3
     MPI_INEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
4
                   RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)
5
         <type> SENDBUF(*), RECVBUF(*)
6
         INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
7
                   RECVTYPE, COMM, REQUEST, IERROR
8
9
     MPI_INEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
10
                   RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)
11
         <type> SENDBUF(*), RECVBUF(*)
12
         INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
13
                   REQUEST, IERROR
14
         INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
15
     MPI_NEIGHBOR_ALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
16
                   RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR)
17
         <type> SENDBUF(*), RECVBUF(*)
18
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
19
                   IERROR
20
21
     MPI_NEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
22
                   RECVTYPE, COMM, IERROR)
23
         <type> SENDBUF(*), RECVBUF(*)
24
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
25
     MPI_NEIGHBOR_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
26
                   RECVCOUNTS, DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)
27
         <type> SENDBUF(*), RECVBUF(*)
28
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
29
                   INFO, REQUEST, IERROR
30
31
     MPI_NEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
32
                   DISPLS, RECVTYPE, COMM, IERROR)
33
         <type> SENDBUF(*), RECVBUF(*)
34
        INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
35
                   IERROR
36
     MPI_NEIGHBOR_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
37
                   RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR)
38
         <type> SENDBUF(*), RECVBUF(*)
39
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
40
                   IERROR
41
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
     MPI_NEIGHBOR_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE,
47
                   RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST,
48
```

IERROR)	1
<type> SENDBUF(*), RECVBUF(*)</type>	2
<pre>INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RECVTYPE, COMM, INFO, REQUEST, IERROR</pre>	3 4
	5
MPI_NEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,	6
RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*)</type>	7
<pre>INTEGER SENDEDF(*), RECVEOF(*) INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),</pre>	8
RECVTYPE, COMM, IERROR	9 10
MPI_NEIGHBOR_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES,	11
RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST,	12
IERROR)	13
<type> SENDBUF(*), RECVBUF(*)</type>	14
INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,	15 16
INFO, REQUEST, IERROR	10
<pre>INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)</pre>	18
MPI_NEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,	19
RECVCOUNTS, RDISPLS, RECVTYPES, COMM, IERROR)	20
<type> SENDBUF(*), RECVBUF(*)</type>	21
<pre>INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,</pre>	22
IERROR	23
<pre>INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)</pre>	24
MPI_TOPO_TEST(COMM, STATUS, IERROR)	25
MP1_TUP0_TEST(COMM, STATUS, IERROR) INTEGER COMM, STATUS, IERROR	26
	26 27
INTEGER COMM, STATUS, IERROR	26 27 28
INTEGER COMM, STATUS, IERROR A.4.6 MPI Environmental Management Fortran Bindings	26 27
INTEGER COMM, STATUS, IERROR	26 27 28 29
INTEGER COMM, STATUS, IERROR A.4.6 MPI Environmental Management Fortran Bindings	26 27 28 29 30
INTEGER COMM, STATUS, IERROR A.4.6 MPI Environmental Management Fortran Bindings DOUBLE PRECISION MPI_WTICK() DOUBLE PRECISION MPI_WTIME()	26 27 28 29 30 31
INTEGER COMM, STATUS, IERROR A.4.6 MPI Environmental Management Fortran Bindings DOUBLE PRECISION MPI_WTICK() DOUBLE PRECISION MPI_WTIME() MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR)	26 27 28 29 30 31 32
INTEGER COMM, STATUS, IERROR A.4.6 MPI Environmental Management Fortran Bindings DOUBLE PRECISION MPI_WTICK() DOUBLE PRECISION MPI_WTIME() MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR) INTEGER ERRORCLASS, IERROR	26 27 28 29 30 31 32 33 34 35
<pre>INTEGER COMM, STATUS, IERROR A.4.6 MPI Environmental Management Fortran Bindings DOUBLE PRECISION MPI_WTICK() DOUBLE PRECISION MPI_WTIME() MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR) INTEGER ERRORCLASS, IERROR MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR)</pre>	26 27 28 29 30 31 32 33 34 35 36
INTEGER COMM, STATUS, IERROR A.4.6 MPI Environmental Management Fortran Bindings DOUBLE PRECISION MPI_WTICK() DOUBLE PRECISION MPI_WTIME() MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR) INTEGER ERRORCLASS, IERROR	26 27 28 29 30 31 32 33 34 35 36 37
<pre>INTEGER COMM, STATUS, IERROR A.4.6 MPI Environmental Management Fortran Bindings DOUBLE PRECISION MPI_WTICK() DOUBLE PRECISION MPI_WTIME() MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR) INTEGER ERRORCLASS, IERROR MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR)</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38
<pre>INTEGER COMM, STATUS, IERROR A.4.6 MPI Environmental Management Fortran Bindings DOUBLE PRECISION MPI_WTICK() DOUBLE PRECISION MPI_WTIME() MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR) INTEGER ERRORCLASS, IERROR MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR) INTEGER ERRORCLASS, ERRORCODE, IERROR)</pre>	26 27 28 29 30 31 32 33 34 35 36 37
<pre>INTEGER COMM, STATUS, IERROR A.4.6 MPI Environmental Management Fortran Bindings DOUBLE PRECISION MPI_WTICK() DOUBLE PRECISION MPI_WTIME() MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR) INTEGER ERRORCLASS, IERROR) MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR) INTEGER ERRORCLASS, ERRORCODE, IERROR) MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR)</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39
<pre>INTEGER COMM, STATUS, IERROR A.4.6 MPI Environmental Management Fortran Bindings DOUBLE PRECISION MPI_WTICK() DOUBLE PRECISION MPI_WTIME() MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR) INTEGER ERRORCLASS, IERROR MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR) INTEGER ERRORCLASS, ERRORCODE, IERROR MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR) INTEGER ERRORCODE, IERROR CHARACTER*(*) STRING</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
<pre>INTEGER COMM, STATUS, IERROR A.4.6 MPI Environmental Management Fortran Bindings DOUBLE PRECISION MPI_WTICK() DOUBLE PRECISION MPI_WTIME() MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR) INTEGER ERRORCLASS, IERROR) MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR) INTEGER ERRORCLASS, ERRORCODE, IERROR) INTEGER ERRORCLASS, ERRORCODE, IERROR MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR) INTEGER ERRORCODE, IERROR</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41
<pre>INTEGER COMM, STATUS, IERROR A.4.6 MPI Environmental Management Fortran Bindings DOUBLE PRECISION MPI_WTICK() DOUBLE PRECISION MPI_WTIME() MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR) INTEGER ERRORCLASS, IERROR MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR) INTEGER ERRORCLASS, ERRORCODE, IERROR) INTEGER ERRORCLASS, ERRORCODE, IERROR) INTEGER ERRORCLASS, ERRORCODE, IERROR) MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR) INTEGER ERRORCODE, IERROR CHARACTER*(*) STRING MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
<pre>INTEGER COMM, STATUS, IERROR A.4.6 MPI Environmental Management Fortran Bindings DOUBLE PRECISION MPI_WTICK() DOUBLE PRECISION MPI_WTIME() MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR) INTEGER ERRORCLASS, IERROR MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR) INTEGER ERRORCLASS, ERRORCODE, IERROR) MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR) INTEGER ERRORCODE, IERROR CHARACTER*(*) STRING MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR) INTEGER (KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR INTEGER INFO, IERROR</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
<pre>INTEGER COMM, STATUS, IERROR A.4.6 MPI Environmental Management Fortran Bindings DOUBLE PRECISION MPI_WTICK() DOUBLE PRECISION MPI_WTIME() MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR) INTEGER ERRORCLASS, IERROR MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR) INTEGER ERRORCLASS, ERRORCODE, IERROR MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR) INTEGER ERRORCODE, IERROR CHARACTER*(*) STRING MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR) INTEGER INFO, IERROR INTEGER INFO, IERROR If the Fortran compiler provides TYPE(C_PTR), then overloaded by:</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
<pre>INTEGER COMM, STATUS, IERROR A.4.6 MPI Environmental Management Fortran Bindings DOUBLE PRECISION MPI_WTICK() DOUBLE PRECISION MPI_WTIME() MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR) INTEGER ERRORCLASS, IERROR MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR) INTEGER ERRORCLASS, ERRORCODE, IERROR) MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR) INTEGER ERRORCODE, IERROR CHARACTER*(*) STRING MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR) INTEGER (KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR INTEGER INFO, IERROR</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44

```
1
           IMPORT :: MPI_ADDRESS_KIND
2
           INTEGER :: INFO, IERROR
3
           INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR
4
         END SUBROUTINE
\mathbf{5}
         SUBROUTINE MPI_ALLOC_MEM_CPTR(SIZE, INFO, BASEPTR, IERROR)
6
           USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
7
           IMPORT :: MPI_ADDRESS_KIND
8
           INTEGER :: INFO, IERROR
9
           INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
10
           TYPE(C_PTR) :: BASEPTR
11
         END SUBROUTINE
12
       END INTERFACE
13
     MPI_COMM_CALL_ERRHANDLER(COMM, ERRORCODE, IERROR)
14
         INTEGER COMM, ERRORCODE, IERROR
15
16
     MPI_COMM_CREATE_ERRHANDLER(COMM_ERRHANDLER_FN, ERRHANDLER, IERROR)
17
         EXTERNAL COMM_ERRHANDLER_FN
18
         INTEGER ERRHANDLER, IERROR
19
     MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
20
         INTEGER COMM, ERRHANDLER, IERROR
21
22
     MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
23
         INTEGER COMM, ERRHANDLER, IERROR
24
     MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)
25
         INTEGER ERRHANDLER, IERROR
26
27
     MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
28
         INTEGER ERRORCODE, ERRORCLASS, IERROR
29
     MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)
30
         INTEGER ERRORCODE, RESULTLEN, IERROR
^{31}
         CHARACTER*(*) STRING
32
33
     MPI_FILE_CALL_ERRHANDLER(FH, ERRORCODE, IERROR)
34
         INTEGER FH, ERRORCODE, IERROR
35
     MPI_FILE_CREATE_ERRHANDLER(FILE_ERRHANDLER_FN, ERRHANDLER, IERROR)
36
37
         EXTERNAL FILE_ERRHANDLER_FN
         INTEGER ERRHANDLER, IERROR
38
39
     MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
40
         INTEGER FILE, ERRHANDLER, IERROR
41
42
     MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
         INTEGER FILE, ERRHANDLER, IERROR
43
44
     MPI_FREE_MEM(BASE, IERROR)
45
         <type> BASE(*)
46
         INTEGER IERROR
47
48
     MPI_GET_LIBRARY_VERSION(VERSION, RESULTLEN, IERROR)
```

CHARACTER*(*) VERSION INTEGER RESULTLEN, IERROR	1 2
MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERROR)	3 4
CHARACTER*(*) NAME INTEGER RESULTLEN, IERROR	5 6
MPI_GET_VERSION(VERSION, SUBVERSION, IERROR) INTEGER VERSION, SUBVERSION, IERROR	7 8
MPI_SESSION_CALL_ERRHANDLER(SESSION, ERRORCODE, IERROR) INTEGER SESSION, ERRORCODE, IERROR	9 10 11
MPI_SESSION_CREATE_ERRHANDLER(SESSION_ERRHANDLER_FN, ERRHANDLER, IERROR) EXTERNAL SESSION_ERRHANDLER_FN INTEGER ERRHANDLER, IERROR	12 13 14
MPI_SESSION_GET_ERRHANDLER(SESSION, ERRHANDLER, IERROR) INTEGER SESSION, ERRHANDLER, IERROR	15 16 17
MPI_SESSION_SET_ERRHANDLER(SESSION, ERRHANDLER, IERROR) INTEGER SESSION, ERRHANDLER, IERROR	18 19 20
MPI_WIN_CALL_ERRHANDLER(WIN, ERRORCODE, IERROR) INTEGER WIN, ERRORCODE, IERROR	20 21 22
MPI_WIN_CREATE_ERRHANDLER(WIN_ERRHANDLER_FN, ERRHANDLER, IERROR) EXTERNAL WIN_ERRHANDLER_FN INTEGER ERRHANDLER, IERROR	23 24 25 26
MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR) INTEGER WIN, ERRHANDLER, IERROR	20 27 28
MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR) INTEGER WIN, ERRHANDLER, IERROR	29 30 31
	32
A.4.7 The Info Object Fortran Bindings	33
MPI_INFO_CREATE_ENV(INFO, IERROR) INTEGER INFO, IERROR	34 35 36
MPI_INFO_CREATE(INFO, IERROR) INTEGER INFO, IERROR	37 38
MPI_INFO_DELETE(INFO, KEY, IERROR)	39 40
INTEGER INFO, IERROR CHARACTER*(*) KEY	41 42
MPI_INFO_DUP(INFO, NEWINFO, IERROR)	43
INTEGER INFO, NEWINFO, IERROR	44
MPI_INFO_FREE(INFO, IERROR)	45 46
INTEGER INFO, IERROR	47
	48

```
1
     MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)
\mathbf{2}
         INTEGER INFO, VALUELEN, IERROR
3
         CHARACTER*(*) KEY, VALUE
4
         LOGICAL FLAG
5
     MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)
6
         INTEGER INFO, NKEYS, IERROR
7
8
     MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERROR)
9
         INTEGER INFO, N, IERROR
10
         CHARACTER*(*) KEY
11
     MPI_INFO_GET_STRING(INFO, KEY, BUFLEN, VALUE, FLAG, IERROR)
12
         INTEGER INFO, BUFLEN, IERROR
13
         CHARACTER*(*) KEY, VALUE
14
         LOGICAL FLAG
15
16
     MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
17
         INTEGER INFO, VALUELEN, IERROR
18
         CHARACTER*(*) KEY
19
         LOGICAL FLAG
20
     MPI_INFO_SET(INFO, KEY, VALUE, IERROR)
21
         INTEGER INFO, IERROR
22
         CHARACTER*(*) KEY, VALUE
23
^{24}
25
     A.4.8 Process Creation and Management Fortran Bindings
26
     MPI_ABORT(COMM, ERRORCODE, IERROR)
27
         INTEGER COMM, ERRORCODE, IERROR
28
29
     MPI_CLOSE_PORT(PORT_NAME, IERROR)
30
         CHARACTER*(*) PORT_NAME
^{31}
         INTEGER IERROR
32
     MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
33
34
        CHARACTER*(*) PORT_NAME
         INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
35
36
     MPI_COMM_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
37
         CHARACTER*(*) PORT_NAME
38
         INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
39
40
     MPI_COMM_DISCONNECT(COMM, IERROR)
41
         INTEGER COMM, IERROR
42
     MPI_COMM_GET_PARENT(PARENT, IERROR)
43
         INTEGER PARENT, IERROR
44
45
     MPI_COMM_JOIN(FD, INTERCOMM, IERROR)
46
         INTEGER FD, INTERCOMM, IERROR
47
48
```

MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES, IERROR)	$\frac{1}{2}$
CHARACTER*(*) COMMAND, ARGV(*)	3
INTEGER MAXPROCS, INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*),	4
IERROR	5
MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,	6
ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,	7 8
ARRAY_OF_ERRCODES, IERROR)	9
INTEGER COUNT, ARRAY_OF_MAXPROCS(*), ARRAY_OF_INFO(*), ROOT, COMM,	10
INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR	11
CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)	12
MPI_FINALIZED(FLAG, IERROR)	13
LOGICAL FLAG	14
INTEGER IERROR	15
MPI_FINALIZE(IERROR)	16
INTEGER IERROR	17 18
	18
MPI_INITIALIZED(FLAG, IERROR)	20
LOGICAL FLAG INTEGER IERROR	21
	22
MPI_INIT(IERROR)	23
INTEGER IERROR	24
MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)	25
INTEGER REQUIRED, PROVIDED, IERROR	26
MPI_IS_THREAD_MAIN(FLAG, IERROR)	27 28
LOGICAL FLAG	28 29
INTEGER IERROR	30
MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)	31
CHARACTER*(*) SERVICE_NAME, INFO, PORT_NAME, IERROR)	32
INTEGER INFO, IERROR	33
	34
MPI_OPEN_PORT(INFO, PORT_NAME, IERROR)	35
INTEGER INFO, IERROR CHARACTER*(*) PORT_NAME	36
CHARACTER*(*) FURI_NAME	37 38
MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)	39
CHARACTER*(*) SERVICE_NAME, PORT_NAME	40
INTEGER INFO, IERROR	41
MPI_QUERY_THREAD(PROVIDED, IERROR)	42
INTEGER PROVIDED, IERROR	43
MPI_SESSION_FINALIZE(SESSION, IERROR)	44
INTEGER SESSION, IERROR	45
MDT GEGGION GET INEO (GEGGION INEO NGED IEDDOD)	$46 \\ 47$
MPI_SESSION_GET_INFO(SESSION, INFO_USED, IERROR) INTEGER SESSION, INFO_USED, IERROR	47
	-0

```
1
    MPI_SESSION_GET_NTH_PSET(SESSION, INFO, N, PSET_LEN, PSET_NAME, IERROR)
\mathbf{2}
         INTEGER SESSION, INFO, N, PSET_LEN, IERROR
3
         CHARACTER*(*) PSET_NAME
4
     MPI_SESSION_GET_NUM_PSETS(SESSION, INFO, NPSET_NAMES, IERROR)
5
         INTEGER SESSION, INFO, NPSET_NAMES, IERROR
6
\overline{7}
     MPI_SESSION_GET_PSET_INFO(SESSION, PSET_NAME, INFO, IERROR)
8
         INTEGER SESSION, INFO, IERROR
9
         CHARACTER*(*) PSET_NAME
10
     MPI_SESSION_INIT(INFO, ERRHANDLER, SESSION, IERROR)
11
         INTEGER INFO, ERRHANDLER, SESSION, IERROR
12
13
    MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
14
         CHARACTER*(*) SERVICE_NAME, PORT_NAME
15
         INTEGER INFO, IERROR
16
17
     A.4.9 One-Sided Communications Fortran Bindings
18
19
    MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
20
                   TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
21
         <type> ORIGIN_ADDR(*)
22
         INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
23
                   TARGET_DATATYPE, OP, WIN, IERROR
24
         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
25
    MPI_COMPARE_AND_SWAP(ORIGIN_ADDR, COMPARE_ADDR, RESULT_ADDR, DATATYPE,
26
                   TARGET_RANK, TARGET_DISP, WIN, IERROR)
27
         <type> ORIGIN_ADDR(*), COMPARE_ADDR(*), RESULT_ADDR(*)
28
         INTEGER DATATYPE, TARGET RANK, WIN, IERROR
29
         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
30
31
     MPI_FETCH_AND_OP(ORIGIN_ADDR, RESULT_ADDR, DATATYPE, TARGET_RANK,
32
                   TARGET_DISP, OP, WIN, IERROR)
33
         <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
34
         INTEGER DATATYPE, TARGET_RANK, OP, WIN, IERROR
35
         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
36
37
     MPI_GET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_ADDR,
                   RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, TARGET_DISP,
38
                   TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
39
         <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
40
41
         INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
42
                   TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR
         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
43
44
     MPI_GET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
45
                   TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
46
         <type> ORIGIN_ADDR(*)
47
48
```

1 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT, 2 TARGET_DATATYPE, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP 4 MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, 5 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR) 6 <type> ORIGIN_ADDR(*) 7 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT, 8 TARGET_DATATYPE, WIN, IERROR 9 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP 10 11 MPI_RACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST, 1213 IERROR) 14<type> ORIGIN_ADDR(*) 15INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST, IERROR 1617INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP 18 MPI_RGET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, 19 RESULT_ADDR, RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, 20TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST, 21IERROR) 22 <type> ORIGIN_ADDR(*), RESULT_ADDR(*) 23INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE, 24TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST, 25IERROR 26INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP 2728 MPI_RGET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, 29TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST, 30 IERROR) 31<type> ORIGIN_ADDR(*) 32 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT, 33 TARGET_DATATYPE, WIN, REQUEST, IERROR 34 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP 35 MPI_RPUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, 36 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST, 37 IERROR) 38 <type> ORIGIN_ADDR(*) 39 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT, 40TARGET_DATATYPE, WIN, REQUEST, IERROR 41 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP 4243 MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR) 44 INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR 45INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR 46If the Fortran compiler provides TYPE(C_PTR), then overloaded by: 47INTERFACE MPI_WIN_ALLOCATE_SHARED 48

```
1
         SUBROUTINE MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, &
2
               BASEPTR, WIN, IERROR)
3
           IMPORT :: MPI_ADDRESS_KIND
4
           INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
5
           INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR
6
         END SUBROUTINE
7
         SUBROUTINE MPI_WIN_ALLOCATE_SHARED_CPTR(SIZE, DISP_UNIT, INFO, COMM, &
8
               BASEPTR, WIN, IERROR)
9
           USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
10
           IMPORT :: MPI_ADDRESS_KIND
11
           INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
12
           INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
13
           TYPE(C_PTR) :: BASEPTR
14
         END SUBROUTINE
15
       END INTERFACE
16
     MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
17
         INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
18
         INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
19
     If the Fortran compiler provides TYPE(C_PTR), then overloaded by:
20
21
       INTERFACE MPI_WIN_ALLOCATE
22
         SUBROUTINE MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
23
               WIN, IERROR)
24
           IMPORT :: MPI_ADDRESS_KIND
25
           INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
26
           INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR
27
         END 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
31
           IMPORT :: 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
     MPI_WIN_ATTACH(WIN, BASE, SIZE, IERROR)
38
         INTEGER WIN, IERROR
39
         <type> BASE(*)
40
         INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
41
42
     MPI_WIN_COMPLETE(WIN, IERROR)
43
         INTEGER WIN, IERROR
44
    MPI_WIN_CREATE(BASE, SIZE, DISP_UNIT, INFO, COMM, WIN, IERROR)
45
         <type> BASE(*)
46
         INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
47
         INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
48
```

MPI_WIN_CREATE_DYNAMIC(INFO, COMM, WIN, IERROR)	1
INTEGER INFO, COMM, WIN, IERROR	3
MPI_WIN_DETACH(WIN, BASE, IERROR)	4
INTEGER WIN, IERROR	5
<type> BASE(*)</type>	6
MPI_WIN_FENCE(ASSERT, WIN, IERROR)	7
INTEGER ASSERT, WIN, IERROR	8
	9
MPI_WIN_FLUSH_ALL(WIN, IERROR)	10
INTEGER WIN, IERROR	11
MPI_WIN_FLUSH_LOCAL_ALL(WIN, IERROR)	12
INTEGER WIN, IERROR	13
MPI_WIN_FLUSH_LOCAL(RANK, WIN, IERROR)	14
INTEGER RANK, WIN, IERROR	15 16
MPI_WIN_FLUSH(RANK, WIN, IERROR)	17
INTEGER RANK, WIN, IERROR	18
MPI_WIN_FREE(WIN, IERROR)	19
INTEGER WIN, IERROR	20
	21 22
MPI_WIN_GET_GROUP(WIN, GROUP, IERROR)	22
INTEGER WIN, GROUP, IERROR	24
MPI_WIN_GET_INFO(WIN, INFO_USED, IERROR)	25
INTEGER WIN, INFO_USED, IERROR	26
NDT LITH LOCK ALL (ACCEPT LITH TERROR)	27
MPI_WIN_LOCK_ALL(ASSERT, WIN, IERROR) INTEGER ASSERT, WIN, IERROR	28
INTEGER ASSENT, WIN, TENROR	29
MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR)	30
INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR	31
MPI_WIN_POST(GROUP, ASSERT, WIN, IERROR)	32
INTEGER GROUP, ASSERT, WIN, IERROR	33
	34
MPI_WIN_SET_INFO(WIN, INFO, IERROR)	35 36
INTEGER WIN, INFO, IERROR	30
MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, BASEPTR, IERROR)	38
INTEGER WIN, RANK, DISP_UNIT, IERROR	39
INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR	40
If the Fortran compiler provides TYPE(C_PTR), then overloaded by:	41
INTERFACE MPI_WIN_SHARED_QUERY	42
SUBROUTINE MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, &	43
BASEPTR, IERROR)	44
IMPORT :: MPI_ADDRESS_KIND	45
INTEGER :: WIN, RANK, DISP_UNIT, IERROR	46
INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR	47
END SUBROUTINE	48

```
1
         SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, &
2
               BASEPTR, IERROR)
3
           USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
4
           IMPORT :: MPI_ADDRESS_KIND
5
           INTEGER :: WIN, RANK, DISP_UNIT, IERROR
6
           INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
7
           TYPE(C_PTR) :: BASEPTR
8
         END SUBROUTINE
9
       END INTERFACE
10
     MPI_WIN_START(GROUP, ASSERT, WIN, IERROR)
11
         INTEGER GROUP, ASSERT, WIN, IERROR
12
13
     MPI_WIN_SYNC(WIN, IERROR)
14
         INTEGER WIN, IERROR
15
     MPI_WIN_TEST(WIN, FLAG, IERROR)
16
         INTEGER WIN, IERROR
17
         LOGICAL FLAG
18
19
     MPI_WIN_UNLOCK_ALL(WIN, IERROR)
20
         INTEGER WIN, IERROR
21
     MPI_WIN_UNLOCK(RANK, WIN, IERROR)
22
         INTEGER RANK, WIN, IERROR
23
^{24}
     MPI_WIN_WAIT(WIN, IERROR)
25
         INTEGER WIN, IERROR
26
27
     A.4.10 External Interfaces Fortran Bindings
28
29
     MPI_GREQUEST_COMPLETE(REQUEST, IERROR)
30
         INTEGER REQUEST, IERROR
^{31}
32
     MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST,
33
                   IERROR)
34
        EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN
35
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
36
         INTEGER REQUEST, IERROR
37
     MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR)
38
         INTEGER STATUS(MPI_STATUS_SIZE), IERROR
39
         LOGICAL FLAG
40
41
     MPI_STATUS_SET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
42
         INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
43
     MPI_STATUS_SET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)
44
         INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR
45
         INTEGER(KIND=MPI_COUNT_KIND) COUNT
46
47
48
```

A.4.11 I/O Fortran Bindings	1
MPI_CONVERSION_FN_NULL(USERBUF, DATATYPE, COUNT, FILEBUF, POSITION, EXTRA_STATE, IERROR)	2 3
<type> USERBUF(*), FILEBUF(*)</type>	4 5
INTEGER DATATYPE, COUNT, IERROR	6
INTEGER(KIND=MPI_OFFSET_KIND) POSITION	7
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	8
MPI_FILE_CLOSE(FH, IERROR)	9
INTEGER FH, IERROR	10
MPI_FILE_DELETE(FILENAME, INFO, IERROR)	11 12
CHARACTER*(*) FILENAME	13
INTEGER INFO, IERROR	14
MPI_FILE_GET_AMODE(FH, AMODE, IERROR)	15
INTEGER FH, AMODE, IERROR	16
MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR)	17
INTEGER FH, IERROR	18 19
LOGICAL FLAG	19 20
	21
MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR) INTEGER FH, IERROR	22
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP	23
	24
MPI_FILE_GET_GROUP(FH, GROUP, IERROR) INTEGER FH, GROUP, IERROR	25
	26 27
MPI_FILE_GET_INFO(FH, INFO_USED, IERROR)	27
INTEGER FH, INFO_USED, IERROR	29
MPI_FILE_GET_POSITION(FH, OFFSET, IERROR)	30
INTEGER FH, IERROR	31
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	32
MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERROR)	33
INTEGER FH, IERROR	34 35
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	36
MPI_FILE_GET_SIZE(FH, SIZE, IERROR)	37
INTEGER FH, IERROR	38
INTEGER(KIND=MPI_OFFSET_KIND) SIZE	39
MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERROR)	40
INTEGER FH, DATATYPE, IERROR	41
INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT	42 43
MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)	43 44
INTEGER FH, ETYPE, FILETYPE, IERROR	45
INTEGER(KIND=MPI_OFFSET_KIND) DISP	46
CHARACTER*(*) DATAREP	47
	48

```
1
    MPI_FILE_IREAD_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
\mathbf{2}
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
3
         <type> BUF(*)
4
     MPI_FILE_IREAD_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_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
10
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
11
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
12
         <type> BUF(*)
13
     MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
14
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
15
         <type> BUF(*)
16
17
     MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
18
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
19
         <type> BUF(*)
20
     MPI_FILE_IWRITE_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
21
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
22
         <type> BUF(*)
23
24
     MPI_FILE_IWRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
25
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
26
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
27
         <type> BUF(*)
28
     MPI_FILE_IWRITE_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
33
     MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
34
        INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
35
         <type> BUF(*)
36
     MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
37
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
38
         <type> BUF(*)
39
40
     MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR)
41
         INTEGER COMM, AMODE, INFO, FH, IERROR
42
         CHARACTER*(*) FILENAME
43
    MPI_FILE_PREALLOCATE(FH, SIZE, IERROR)
44
         INTEGER FH, IERROR
45
         INTEGER(KIND=MPI_OFFSET_KIND) SIZE
46
47
     MPI_FILE_READ_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
48
```

1 INTEGER FH, COUNT, DATATYPE, IERROR 2 <type> BUF(*) MPI_FILE_READ_ALL_END(FH, BUF, STATUS, IERROR) INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR <type> BUF(*) MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 9 <type> BUF(*) 10 MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR) 11 INTEGER FH, COUNT, DATATYPE, IERROR 12INTEGER(KIND=MPI_OFFSET_KIND) OFFSET 13 <type> BUF(*) 1415MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR) 16INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR 17 <type> BUF(*) 18 MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) 19 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 20INTEGER(KIND=MPI_OFFSET_KIND) OFFSET 21<type> BUF(*) 22 23MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) 24INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 25INTEGER(KIND=MPI_OFFSET_KIND) OFFSET 26<type> BUF(*) 27MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) 28 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 29 <type> BUF(*) 30 31MPI FILE READ ORDERED BEGIN (FH, BUF, COUNT, DATATYPE, IERROR) 32 INTEGER FH, COUNT, DATATYPE, IERROR 33 <type> BUF(*) 34 MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR) 35 INTEGER FH, STATUS (MPI_STATUS_SIZE), IERROR 36 <type> BUF(*) 37 38 MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) 39 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 40 <type> BUF(*) 41 MPI_FILE_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) 42INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 43 <type> BUF(*) 4445MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR) 46INTEGER FH, WHENCE, IERROR 47INTEGER(KIND=MPI_OFFSET_KIND) OFFSET 48

```
1
     MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR)
\mathbf{2}
         INTEGER FH, WHENCE, IERROR
3
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
4
     MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR)
5
         INTEGER FH, IERROR
6
         LOGICAL FLAG
7
8
     MPI_FILE_SET_INFO(FH, INFO, IERROR)
9
         INTEGER FH, INFO, IERROR
10
     MPI_FILE_SET_SIZE(FH, SIZE, IERROR)
11
         INTEGER FH, IERROR
12
         INTEGER(KIND=MPI_OFFSET_KIND) SIZE
13
14
     MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)
15
         INTEGER FH, ETYPE, FILETYPE, INFO, IERROR
16
         INTEGER(KIND=MPI_OFFSET_KIND) DISP
17
         CHARACTER*(*) DATAREP
18
    MPI_FILE_SYNC(FH, IERROR)
19
         INTEGER FH, IERROR
20
21
     MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
22
         INTEGER FH, COUNT, DATATYPE, IERROR
23
         <type> BUF(*)
24
     MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)
25
         INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
26
         <type> BUF(*)
27
28
     MPI_FILE_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
29
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
30
         <type> BUF(*)
^{31}
     MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
32
         INTEGER FH, COUNT, DATATYPE, IERROR
33
        INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
34
         <type> BUF(*)
35
36
     MPI_FILE_WRITE_AT_ALL_END(FH, BUF, STATUS, IERROR)
37
         INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
38
         <type> BUF(*)
39
     MPI_FILE_WRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
40
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
41
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
42
         <type> BUF(*)
43
44
     MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
45
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
46
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
47
         <type> BUF(*)
48
```

MPI_	_FILE_WRI	ITE(F	FH, BU	JF,	COUNT,	DA	ATATYPE,	ST/	ATUS,	IERROR)	
	INTEGER	FH,	COUNT	Г,	DATATYPE	Ξ,	STATUS (MPI_	STATU	JS_SIZE),	IERROR
	<type> H</type>	3UF(*	×)								

- MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
 INTEGER FH, COUNT, DATATYPE, IERROR
 <type> BUF(*)
- MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)
 INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
 <type> BUF(*)

MPI_	_FILE_	WRI	TE_	ORDERED	(FH,	BUF,	COUNT,	DATAT	ΓYΡE,	ST	ATUS,	Ι	ERROR)
	INTEG	ER	FH,	COUNT,	DAT	ATYPE,	STATUS	S(MPI	STAT	JS_S	SIZE)	,	IERROR
	<type< td=""><td>> E</td><td>UF (</td><td>*)</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></type<>	> E	UF (*)									

```
MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
<type> BUF(*)
```

MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR) CHARACTER*(*) DATAREP EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

INTEGER IERROR

A.4.12 Language Bindings Fortran Bindings

- MPI_STATUS_F082F(F08_STATUS, F_STATUS, IERROR)
 TYPE(MPI_Status) :: F08_STATUS
 INTEGER :: F_STATUS(MPI_STATUS_SIZE), IERROR
- MPI_STATUS_F2F08(F_STATUS, F08_STATUS, IERROR)
 INTEGER :: F_STATUS(MPI_STATUS_SIZE), IERROR
 TYPE(MPI_Status) :: F08_STATUS
- MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR) INTEGER P, R, NEWTYPE, IERROR
- MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR) INTEGER R, NEWTYPE, IERROR
- MPI_TYPE_CREATE_F90_REAL(P, R, NEWTYPE, IERROR) INTEGER P, R, NEWTYPE, IERROR
- MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, DATATYPE, IERROR) INTEGER TYPECLASS, SIZE, DATATYPE, IERROR

```
A.4.13 Tools / Profiling Interface Fortran Bindings
1
\mathbf{2}
     MPI_PCONTROL(LEVEL)
3
         INTEGER LEVEL
4
5
6
     A.4.14 Deprecated Fortran Bindings
7
     MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)
8
         INTEGER COMM, KEYVAL, IERROR
9
10
     MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
11
         INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
12
         LOGICAL FLAG
13
     MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
14
         INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
15
16
     MPI_DUP_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
17
                   ATTRIBUTE_VAL_OUT, FLAG, IERR)
18
         INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
19
                    ATTRIBUTE_VAL_OUT, IERR
20
         LOGICAL FLAG
21
     MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)
22
         INTEGER INFO, VALUELEN, IERROR
23
         CHARACTER*(*) KEY, VALUE
24
         LOGICAL FLAG
25
26
     MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
27
         INTEGER INFO, VALUELEN, IERROR
28
         CHARACTER*(*) KEY
29
         LOGICAL FLAG
30
     MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
31
         EXTERNAL COPY_FN, DELETE_FN
32
         INTEGER KEYVAL, EXTRA_STATE, IERROR
33
34
     MPI_KEYVAL_FREE(KEYVAL, IERROR)
35
         INTEGER KEYVAL, IERROR
36
37
     MPI_NULL_COPY_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
38
                   ATTRIBUTE_VAL_OUT, FLAG, IERR)
39
         INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
40
                    ATTRIBUTE_VAL_OUT, IERR
41
         LOGICAL FLAG
42
     MPI_NULL_DELETE_FN(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)
43
         INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR
44
45
     MPI_SIZEOF(X, SIZE, IERROR)
46
         <type> X
47
         INTEGER SIZE, IERROR
48
```

Annex B

Change-Log

Annex B.1 summarizes changes from the previous version of the MPI standard to the version presented by this document. Only significant changes (i.e., clarifications and new features) that might either require implementation effort in the MPI libraries or change the understanding of MPI from a user's perspective are presented. Editorial modifications, formatting, typo corrections and minor clarifications are not shown. If not otherwise noted, the section and page references refer to the locations of the change or new functionality in this version of the standard. Changes in Annexes B.2–B.6 were already introduced in the corresponding sections in previous versions of this standard.

B.1	Changes from Version 3.2 to Version 4.0	24
B.1.1	1 Changes in MPI-4.0	25
D.1	Changes in wir 1-4.0	26
1.	Section 15.3 on page 701.	27
	MPI_SIZEOF was deprecated.	28
2	Section $6.4.2$ on page 269 .	29
۷.	MPI_COMM_TYPE_HW_UNGUIDED was added as a new possible value for the split_	30
	parameter of the MPI_COMM_SPLIT_TYPE function.	
		32
3.	Section 14.3.8.	33
	A callback-driven event interface was added to the MPI tool information interface	
4	Section $6.4.2$ on page 269 .	35
4.	MPI_COMM_TYPE_HW_GUIDED was added as a new possible value for the split_	36 type 37
	parameter of the MPI_COMM_SPLIT_TYPE function, as well as a new info key	38 s
	mpi_hw_resource_type. A specific value associated with this new info key is also define	
	mpi_inv_resource_type. It specific varue associated with this new fillow fills also done mpi_shared_memory.	40
		41
5.	Chapter 10, Sections 2.8, 3.2.3, 6.2.4, 6.3.2, 6.4.2, 6.6.2, 7.4, 7.5.1, 7.5.3, 7.5.4, 8.	42
	8.1.2, 8.3, 8.3.4, 8.5, 10.6, 13.2.1, 13.2.7, 13.7, 14.3.4, 18.2.4, 18.2.6, and Annex A	43
	pages 419, 20, 29, 257, 260, 269, 299, 330, 332, 334, 336, 383, 385, 390, 398, 402,	$448, _{44}$
	569, 575, 634, 650, 756, 761, and 771	45
	The Sessions Model was added to the standard.	46
6.	Section $10.2.1$ on page 420 .	47
	A new function MPI_INFO_CREATE_ENV was added.	48
	Unofficial Draft for Comment Only	913

1	B.2	Changes from Version 3.1 to Version 3.2
2 3	B.2.1	Changes in MPI-3.2
4 5 3 7	1.	Section 3.8.4 on page 80. Cancelling a send request by calling MPI_CANCEL has been deprecated and may be removed in a future version of the MPI specification.
8 9 0	2.	Sections 3.7.3, 3.9, 5.13, 7.8, and 7.9 on pages 60, 82, 226, 371, and 377. Persistent collective communication and persistent neighborhood communication were added to the standard.
1 2 3 4 5	3.	Section 6.4.2 on page 269, and MPI-3.1 Section 6.4.2 on page 237. The functions MPI_COMM_DUP and MPI_COMM_IDUP were updated to no longer propagate info hints. This change may affect backward compatibility.
6 7 8 9	4.	Sections 6.4.4, 11.2.7, and 13.2.8 on pages 286, 496, and 577, and MPI-3.1 Sections 6.4.4, 11.2.7, and 13.2.8 on pages 248, 415, and 500. The definition of info hints was updated to allow applications to provide assertions regarding their usage of MPI objects and operations.
20 21 22 23 24	5.	Section 6.4.4 on page 286. 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 commu- nicators.
5 6 7	6.	Section 6.4.2 on page 269. The MPI_COMM_IDUP_WITH_INFO function was added.
8 9 0 1	7.	Sections 6.4.4, 11.2.7, and 13.2.8 on pages 286, 496, and 577. The semantics of the MPI_COMM_SET_INFO, MPI_COMM_GET_INFO, MPI_WIN_SET_INFO, MPI_WIN_GET_INFO, MPI_FILE_SET_INFO, and MPI_FILE_GET_INFO were clarified.
	8.	Section 14.3.10 and Table 14.7 on pages 693 and 695. MPI_T_ERR_INVALID_ITEM is deprecated. MPI routines should return MPI_T_ERR_INVALID_INDEX instead of MPI_T_ERR_INVALID_ITEM.
3 7 3	9.	Section 7.5. 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.
) 1 2 3 4	10.	Sections 2.8, 8.3, 8.5, and 10.2.1 on pages 20, 390, 402, and 420. MPI calls that are not related to any objects are considered to be attached to the communicator MPI_COMM_SELF instead of MPI_COMM_WORLD. The definition of MPI_ERRORS_ARE_FATAL was clarified to cover all connected processes, and a new error handler, MPI_ERRORS_ABORT, was created to limit the scope of aborting.
5 6 7 8	11.	Sections 8.2, 11.2.2, and 11.2.3 on pages 387, 485, and 487. 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.

- 12. Section 8.4 on page 403. The error class MPI_ERR_PROC_ABORTED has been added.
- 13. Section 12.3 on page ?? 4 The mpi_f08 binding incorrectly had the dummy parameter flag in the MPI F08 5binding for MPI_STATUS_SET_CANCELLED marked as INTENT(OUT). It has been 6 fixed to be INTENT(IN). 7 8 14. Sections 8.3 and 8.4 on pages 390 and 401. 9 Clarified definition of errors to say that MPI should continue whenever possible and 10 allow the user to recover from errors. 11 15. Section ??. 12Added a new function MPI_INFO_GET_STRING that takes a buffer length argument 13 for returning info value strings. This function returns the required buffer length for 1415the requested string and guarantees null termination for C strings where buffer size is greater than 0. 161716. Section ?? on page ?? and Section 15.4 on page ??. 18 MPI_INFO_GET and MPI_INFO_GET_VALUELEN were deprecated. 19 2017. Section 8.4 on page 401. 21Added text to clarify what is implied about the status of MPI and user visible buffers 22 when MPI functions return MPI_SUCCESS or other error codes. 2318. Section 10.2.1 on page 420. Section 10.10.4 on page 477. 24 Clarified the semantic of failure and error reporting before (and during) MPI_INIT 2526and after MPI_FINALIZE. 2719. Section 10.8.4 on page 462. Section 10.8.4 on page 462. 28Added the mpi_initial_errhandler reserved info key with the reserved values 29 mpi_errors_abort, mpi_errors_are_fatal, and mpi_errors_return to the launch keys in 30 MPI_COMM_SPAWN, MPI_COMM_SPAWN_MULTIPLE, and mpiexec 3132 20. Section 3.9 and 3.7 on pages 82 and 54. 33 Addition of MPI_ISENDRECV and MPI_ISENDRECV_REPLACE. 34 35 **B.3** Changes from Version 3.0 to Version 3.1 36 37 B.3.1 Fixes to Errata in Previous Versions of MPI 38 39 1. Chapters 3–18, Annex A.3 on page 821, and Example 5.21 on page 197, and MPI-3.0 40 Chapters 3–17, Annex A.3 on page 707, and Example 5.21 on page 187. 41 Within the mpi_f08 Fortran support method, BIND(C) was removed from all 42SUBROUTINE, FUNCTION, and ABSTRACT INTERFACE definitions. 43 2. Section 3.2.5 on page 32, and MPI-3.0 Section 3.2.5 on page 30. 44The three public fields MPI_SOURCE, MPI_TAG, and MPI_ERROR of the Fortran derived 45type TYPE(MPI_Status) must be of type INTEGER. 4647

1

2

3

1 2 3	3.	Section 3.8.2 on page 76, 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 6.4.2 on page 269, 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 6.4.4 on page 286, 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 15 16	6.	Section 7.6 on page 355, 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).
17 18 19 20	7.	Section 8.1.1 on page 383, 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 25 26	8.	Sections 8.2 (MPI_ALLOC_MEM and MPI_ALLOC_MEM_CPTR), 11.2.2 (MPI_WIN_ALLOCATE and MPI_WIN_ALLOCATE_CPTR), 11.2.3 (MPI_WIN_ALLOCATE_SHARED and MPI_WIN_ALLOCATE_SHARED_CPTR), 11.2.3 (MPI_WIN_SHARED_QUERY and MPI_WIN_SHARED_QUERY_CPTR), 14.2.1 and 14.2.7 (Profiling interface), and corresponding sections in MPI-3.0. The linker name concept was substituted by defining specific procedure names.
27 28 29 30 31	9.	Section 11.2.1 on page 483, 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.
32 33 34 35	10.	Section 11.3.4 on page 504, 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 39	11.	Section 11.3.4 on page 504, 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.
40 41 42	12.	Section 14.3 on page 647, 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	13.	Section 14.3.3 on page 649, and MPI-3.0 Section 14.3.3 on page 563. New paragraph clarifying variable name equivalence in the tools information interface.
46 47 48	14.	Sections 14.3.6, 14.3.7, and 14.3.9 on pages 654, 661, and 688, and MPI-3.0 Sections 14.3.6, 14.3.7, and 14.3.8 on pages 567, 573, and 584. In functions $MPI_T_CVAR_GET_INFO$, $MPI_T_PVAR_GET_INFO$, and

MPI_T_CATEGORY_GET_INFO, clarification of parameters that must be identical for equivalent control variable / performance variable / category names across connected processes.
15. Section 14.3.7 on page 661, and MPI-3.0 Section 14.3.7 on page 573. Clarify return code of MPI_T_PVAR_{START,STOP,RESET} routines.
16. Section 14.3.7 on page 661, and MPI-3.0 Section 14.3.7 on page 579, line 7. Clarify the return code when bad handle is passed to an MPI_T_PVAR_* routine.

17. Section 18.1.4 on page 713, 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.

Section 18.1.5 on page 714, and MPI-3.0 Section 17.1.5 on page 605. The section was fully rewritten. The linker name concept was substituted by defining specific procedure names.

- 19. Section 18.1.6 on page 719, and MPI-3.0 Section 17.1.6 on page 611. The requirements on BIND(C) procedure interfaces were removed.
- 20. Annexes A.2, A.3, and A.4 on pages 795, 821, and 878, and MPI-3.0 Annexes A.2, A.3, and A.4 on pages 685, 707, and 756. The predefined callback MPI_CONVERSION_FN_NULL was added to all three annexes.
- 21. Annex A.3.4 on page 842, and MPI-3.0 Annex A.3.4 on page 724. In the mpi_f08 binding of MPI_{COMM|TYPE|WIN}_{DUP|NULL_COPY|NULL_DELETE}_FN, all INTENT(...) information was removed.

B.3.2 Changes in MPI-3.1

- 1. Sections 2.6.4 and 4.1.5 on pages 20 and 108. The use of the intrinsic operators "+" and "-" for absolute addresses is substituted by MPI_AINT_ADD and MPI_AINT_DIFF. In C, they can be implemented as macros.
- 2. Sections 8.1.1, 10.2.1, and 10.6 on pages 383, 420, and 448.
 - The routines MPI_INITIALIZED, MPI_FINALIZED, MPI_QUERY_THREAD, MPI_IS_THREAD_MAIN, MPI_GET_VERSION, and MPI_GET_LIBRARY_VERSION are callable from threads without restriction (in the sense of MPI_THREAD_MULTIPLE), irrespective of the actual level of thread support provided, in the case where the implementation supports threads.
- Section 11.2.1 on page 483. The same_disp_unit info key was added for use in RMA window creation routines.
- 4. Sections 13.4.2 and 13.4.3 on pages 586 and 591. Added MPI_FILE_IREAD_AT_ALL, MPI_FILE_IWRITE_AT_ALL, MPI_FILE_IREAD_ALL, and MPI_FILE_IWRITE_ALL

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

33

34

35

36

37

38

39

40 41

42

43 44

45

46

1 2 3	 Sections 14.3.6, 14.3.7, and 14.3.9 on pages 654, 661, and 688. Clarified that NULL parameters can be provided in MPI_T_{CVAR PVAR CATEGORY}_GET_INFO routines.
4 5 6 7 8 9	6. Sections 14.3.6, 14.3.7, 14.3.9, and 14.3.10 on pages 654, 661, 688, and 693. New routines MPI_T_CVAR_GET_INDEX, MPI_T_PVAR_GET_INDEX, MPI_T_CATEGORY_GET_INDEX, were added to support retrieving indices of variables and categories. The error codes MPI_T_ERR_INVALID and MPI_T_ERR_INVALID_NAME were added to indicate invalid uses of the interface.
11 12	B.4 Changes from Version 2.2 to Version 3.0
13	B.4.1 Fixes to Errata in Previous Versions of MPI
14 15 16 17	 Sections 2.6.2 and 2.6.3 on pages 18 and 18, 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.
18 19	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.
20 21 22 23 24 25 26 27 28 29 30 31 32	2. Sections 3.2.2, 5.9.2, 13.5.2 Table 13.2, and Annex A.1.1 on pages 27, 186, 618, and 771, and MPI-2.2 Sections 3.2.2, 5.9.2, 13.5.2 Table 13.2, 16.1.16 Table 16.1, and Annex A.1.1 on pages 27, 164, 433, 472 and 513 This is an MPI-2.2 erratum: New named predefined datatypes MPI_CXX_BOOL, MPI_CXX_FLOAT_COMPLEX, MPI_CXX_DOUBLE_COMPLEX, and MPI_CXX_LONG_DOUBLE_COMPLEX were added in C and Fortran corresponding to the C++ types bool, std::complex <float>, std::complex<double>, and std::complex<long double="">. These datatypes also correspond to the deprecated C++ predefined datatypes MPI::BOOL, MPI::COMPLEX, MPI::DOUBLE_COMPLEX and MPI::LONG_DOUBLE_COMPLEX, which were removed in MPI-3.0. The non standard C++ types Complex<> were substituted by the standard types std::complex<></long></double></float>
33 34 35	 Sections 5.9.2 on pages 186 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.
36 37 38 39 40	 4. Section 7.5.5 on page 343, 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.
41 42 43 44 45	 5. Section 13.5.2, Table 13.2 on page 618, and MPI-2.2, Section 13.5.3, Table 13.2 or page 433. This was an MPI-2.2 erratum: The MPI_C_BOOL "external32" representation is corrected to a 1-byte size.
46 47 48	 MPI-2.2 Section 16.1.16 on page 471, line 45. This is an MPI-2.2 erratum: The constant MPI::_LONG_LONG should be MPI::LONG_LONG.

7.	Annex A.1.1 on page 771, Table "Optional datatypes (Fortran)," and	1
	MPI-2.2, Annex A.1.1, Table on page 517, lines 34, and 37–41.	2
	This is an MPI-2.2 erratum: The C++ datatype handles MPI::INTEGER16,	3
	MPI::REAL16, MPI::F_COMPLEX4, MPI::F_COMPLEX8, MPI::F_COMPLEX16,	4
	MPI::F_COMPLEX32 were added to the table.	5
		6
B.4.2	2 Changes in MPI-3.0	7
		8
1.	Section 2.6.1 on page 17, Section 16.2 on page 704 and all other chapters.	9
	The C++ bindings were removed from the standard. See errata in Section B.4.1 on	10
	page 918 for the latest changes to the MPI C++ binding defined in MPI-2.2.	11
	This change may affect backward compatibility.	12
2	Section 2.6.1 on page 17, Section 15.1 on page 697 and Section 16.1 on page 703.	13
۷.	The deprecated functions MPI_TYPE_HVECTOR, MPI_TYPE_HINDEXED,	14
	MPI_TYPE_STRUCT, MPI_ADDRESS, MPI_TYPE_EXTENT, MPI_TYPE_LB,	15
	MPI_TYPE_UB, MPI_ERRHANDLER_CREATE (and its callback function prototype	16
	MPI_Handler_function), MPI_ERRHANDLER_SET, MPI_ERRHANDLER_GET, the dep-	17
	recated special datatype handles MPI_LB, MPI_UB, and the constants	18
	MPI_COMBINER_HINDEXED_INTEGER, MPI_COMBINER_HVECTOR_INTEGER,	19
	MPI_COMBINER_STRUCT_INTEGER were removed from the standard.	20
	This change may affect backward compatibility.	21
	This change may anect backward compatibility.	22
3.	Section 2.3 on page 10.	23
	Clarified parameter usage for IN parameters. C bindings are now const-correct where	24
	backward compatibility is preserved.	25
		26
4.	Section $2.5.4$ on page 15 and Section $7.5.4$ on page 336 .	27
	The recommended C implementation value for MPI_UNWEIGHTED changed from NULL	28
	to non-NULL. An additional weight array constant (MPI_WEIGHTS_EMPTY) was in-	29
	troduced.	30
-		31
5.	Section 2.5.4 on page 15 and Section 8.1.1 on page 383.	32
	Added the new routine MPI_GET_LIBRARY_VERSION to query library specific ver-	33
	sions, and the new constant MPI_MAX_LIBRARY_VERSION_STRING.	34
6	Sections 2.5.8, 3.2.2, 3.3, 5.9.2, on pages 17, 27, 29, 186, Sections 4.1, 4.1.7, 4.1.8,	35
0.	4.1.11, 12.3 on pages 89, 114, 116, 119, 564, and Annex A.1.1 on page 771.	36

4.1.11, New inquiry functions, MPI_TYPE_SIZE_X, MPI_TYPE_GET_EXTENT_X, 37 MPI_TYPE_GET_TRUE_EXTENT_X, and MPI_GET_ELEMENTS_X, return their re-3839sults as an MPI_Count value, which is a new type large enough to represent ele-40ment counts in memory, file views, etc. A new function, 41MPI_STATUS_SET_ELEMENTS_X, modifies the opaque part of an MPI_Status object 42so that a call to MPI_GET_ELEMENTS_X returns the provided MPI_Count value (in 43Fortran, INTEGER (KIND=MPI_COUNT_KIND)). The corresponding predefined datatype is MPI_COUNT. 4445

7. Chapter 3 on page 25 through Chapter 18 on page 707. 46In the C language bindings, the array-arguments' interfaces were modified to consis-47tently use use [] instead of *. 48

1 2 3		Exceptions are MPI_INIT, which continues to use char ***argv (correct because of subtle rules regarding the use of the & operator with char *argv []), and MPI_INIT_THREAD, which is changed to be consistent with MPI_INIT.
4 5 6 7 8 9 10 11	8.	Sections 3.2.5, 4.1.5, 4.1.11, 4.2 on pages 32, 108, 119, 139. The functions MPI_GET_COUNT and MPI_GET_ELEMENTS were defined to set the count argument to MPI_UNDEFINED when that argument would overflow. The functions MPI_PACK_SIZE and MPI_TYPE_SIZE were defined to set the size argument to MPI_UNDEFINED when that argument would overflow. In all other MPI-2.2 routines, the type and semantics of the count arguments remain unchanged, i.e., int or INTEGER.
12 13 14 15	9.	Section 3.2.6 on page 34, and Section 3.8 on page 73. MPI_STATUS_IGNORE can be also used in MPI_IPROBE, MPI_PROBE, MPI_IMPROBE, and MPI_MPROBE.
16 17 18 19 20	10.	Section 3.8 on page 73 and Section 3.10 on page 88. The use of MPI_PROC_NULL in probe operations was clarified. A special predefined message MPI_MESSAGE_NO_PROC was defined for the use of matching probe (i.e., the new MPI_MPROBE and MPI_IMPROBE) with MPI_PROC_NULL.
20 21 22 23 24 25 26 27 28	11.	Sections 3.8.2, 3.8.3, 18.2.4, A.1.1 on pages 76, 78, 756, 771. Like MPI_PROBE and MPI_IPROBE, the new MPI_MPROBE and MPI_IMPROBE operations allow incoming messages to be queried without actually receiving them, except that MPI_MPROBE and MPI_IMPROBE provide a mechanism to receive the specific message with the new routines MPI_MRECV and MPI_IMRECV regardless of other intervening probe or receive operations. The opaque object MPI_Message, the null handle MPI_MESSAGE_NULL, and the conversion functions MPI_Message_c2f and MPI_Message_f2c were defined.
29 30 31 32	12.	Section 4.1.2 on page 91 and Section 4.1.13 on page 124. The routine MPI_TYPE_CREATE_HINDEXED_BLOCK and constant MPI_COMBINER_HINDEXED_BLOCK were added.
33 34	13.	Chapter 5 on page 151 and Section 5.12 on page 208. Added nonblocking interfaces to all collective operations.
35 36 37 38 39	14.	Sections 6.4.2, 6.4.4, 11.2.7, on pages 269, 286, 496. The new routines MPI_COMM_DUP_WITH_INFO, MPI_COMM_SET_INFO, MPI_COMM_GET_INFO, MPI_WIN_SET_INFO, and MPI_WIN_GET_INFO were added. The routine MPI_COMM_DUP must also duplicate info hints.
40 41 42	15.	Section 6.4.2 on page 269. Added MPI_COMM_IDUP.
43 44 45 46 47 48	16.	Section 6.4.2 on page 269. Added the new communicator construction routine MPI_COMM_CREATE_GROUP, which is invoked only by the processes in the group of the new communicator being constructed.

17.	Section 6.4.2 on page 269. Added the MPI_COMM_SPLIT_TYPE routine and the communicator split type con- stant MPI_COMM_TYPE_SHARED.	1 2 3
18.	Section 6.6.2 on page 299. 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.	4 5 6 7 8
19.	Section 6.8 on page 322. Section 6.8 on page 238. The constant MPI_MAX_OBJECT_NAME also applies for type and window names.	9 10 11 12
20.	Section 7.5.8 on page 353. MPI_CART_MAP can also be used for a zero-dimensional topologies.	13 14 15
21.	Section 7.6 on page 355 and Section 7.7 on page 365. 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_ALLTOALL, MPI_INEIGHBOR_ALLTOALL, 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.	16 17 18 19 20 21 22 23 24 25 26
22.	Section 10.2.1 on page 420 and Section 10.2.1 on page 423. 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.	27 28 29 30 31
23.	Section 10.2.1 on page 420. Allow calls to MPI_T routines before MPI_INIT and after MPI_FINALIZE.	32 33
24.	Chapter 11 on page 481. 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.	34 35 36 37 38
25.	Section 14.3 on page 647. A new MPI Tool Information Interface was added. The following changes are related to the Fortran language support.	39 40 41 42
26.	Section 2.3 on page 10, and Sections 18.1.1, 18.1.2, 18.1.7 on pages 707, 708, and 723. The new mpi_08 Fortran module was introduced.	43 44 45
27.	Section 2.5.1 on page 12, and Sections 18.1.2, 18.1.3, 18.1.7 on pages 708, 711, and 723. Handles to opaque objects were defined as named types within the mpi_08 Fortran	46 47 48

1 2 3 4	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.
	. Sections 2.5.4, 2.5.5 on pages 15, 16, Sections 18.1.1, 18.1.10, 18.1.11, 18.1.12, 18.1.13 on pages 707, 733, 735, 735, 738, and Sections 18.1.2, 18.1.3, 18.1.7 on pages 708, 711, 723.
8 9 10 11 12 13	Within the mpi_08 Fortran module, choice buffers were defined as assumed-type and assumed-rank according to Fortran 2008 TS 29113 [43], 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
$^{14}_{15}$ 29	Section 2.6.2 on page 18, Section 18.1.2 on page 708, and Section 18.1.7 on page 723. The ierror dummy arguments are OPTIONAL within the mpi_08 Fortran module.
17 30 18 19 20 21 21 22 23 24 25 26	 Section 3.2.5 on page 32, Sections 18.1.2, 18.1.3, 18.1.7, on pages 708, 711, 723, and Section 18.2.5 on page 758. 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.
	. Section 3.6 on page 49. 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.
30 31 32 33 34 35 36 37	Section 4.1 on page 89, Section 4.1.6 on page 112, and Section 18.1.15 on page 739. 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.
38 35 39 40 41 42 43 44 45 46 47 48	. Sections 4.1.10, 5.9.5, 5.9.7, 6.7.4, 6.8, 8.3.1, 8.3.2, 8.3.3, 15.1, 18.1.9 on pages 119, 193, 199, 317, 322, 393, 395, 397, 697, and 725. In some routines, the dummy argument names were changed because they were identical to the Fortran keywords TYPE and FUNCTION. The new dummy argument names must be used because the mpi and mpi_08 modules guarantee keyword-based actual argument lists. The argument name type was changed in MPI_TYPE_DUP, the Fortran USER_FUNCTION of MPI_OP_CREATE, MPI_TYPE_SET_ATTR, MPI_TYPE_GET_ATTR, MPI_TYPE_DELETE_ATTR, MPI_TYPE_SET_NAME, MPI_TYPE_GET_NAME, MPI_TYPE_GET_NAME, MPI_TYPE_GET_NAME, MPI_TYPE_GET_NAME, MPI_TYPE_GET_NAME, MPI_TYPE_GET_NAME, MPI_TYPE_MATCH_SIZE, the callback prototype definition MPI_TyPE_NULL_DELETE_FN; function was changed in MPI_OP_CREATE,

MPI_COMM_CREATE_ERRHANDLER, MPI_WIN_CREATE_ERRHANDLER, MPI_FILE_CREATE_ERRHANDLER, and MPI_ERRHANDLER_CREATE. For consistency reasons, INOUBUF was changed to INOUTBUF in MPI_REDUCE_LOCAL, and intracomm to newintracomm in MPI_INTERCOMM_MERGE.

534. Section 6.7.2 on page 307. 6 It was clarified that in Fortran, the flag values returned by a comm_copy_attr_fn 7 callback, including MPI_COMM_NULL_COPY_FN and MPI_COMM_DUP_FN, are 8 .FALSE. and .TRUE.; see MPI_COMM_CREATE_KEYVAL. 9 10 35. Section 8.2 on page 387. 11 With the mpi and mpi_f08 Fortran modules, MPI_ALLOC_MEM now also supports 12TYPE(C_PTR) C-pointers instead of only returning an address-sized integer that may 13 be usable together with a non-standard Cray-pointer. 141536. Section 18.1.15 on page 739, and Section 18.1.7 on page 723. 16Fortran SEQUENCE and BIND(C) derived application types can now be used as buffers 17in MPI operations. 18 37. Section 18.1.16 on page 741 to Section 18.1.19 on page 750, Section 18.1.7 on page 723, 19 and Section 18.1.8 on page 724. 20The sections about Fortran optimization problems and their solutions were partially 21rewritten and new methods are added, e.g., the use of the ASYNCHRONOUS attribute. 22 The constant MPI_ASYNC_PROTECTS_NONBLOCKING tells whether the semantics of 23the ASYNCHRONOUS attribute is extended to protect nonblocking operations. The For- 24 tran routine MPI_F_SYNC_REG is added. MPI-3.0 compliance for an MPI library 25together with a Fortran compiler is defined in Section 18.1.7. 262738. Section 18.1.2 on page 708. 28Within the mpi_08 Fortran module, dummy arguments are now declared with 29 INTENT=IN, OUT, or INOUT as defined in the mpi_08 interfaces. 30 39. Section 18.1.3 on page 711, and Section 18.1.7 on page 723. 31The existing mpi Fortran module must implement compile-time argument checking. 32 33 40. Section 18.1.4 on page 713. 34 The use of the mpif.h Fortran include file is now strongly discouraged. 35 36 41. Section A.1.1, Table "Predefined functions" on page 779, Section A.1.3 on page 786, 37 and Section A.3.4 on page 842. 38 Within the new mpi_f08 module, all callback prototype definitions are now defined 39 with explicit interfaces PROCEDURE(MPI_...) that have the BIND(C) attribute; user-40 written callbacks must be modified if the mpi_f08 module is used. 41 4242. Section A.1.3 on page 786. In some routines, the Fortran callback prototype names were changed from \dots FN to 43 ..._FUNCTION to be consistent with the other language bindings. 4445464748

1 2

3

1	B.5	Changes from Version 2.1 to Version 2.2
3 4 5 6	1.	Section 2.5.4 on page 15. It is now guaranteed that predefined named constant handles (as other constants) can be used in initialization expressions or assignments, i.e., also before the call to MPI_INIT.
7 8 9 10	2.	Section 2.6 on page 17, and Section 16.2 on page 704. The C++ language bindings have been deprecated and may be removed in a future version of the MPI specification.
11 12 13 14 15 16 17	3.	Section 3.2.2 on page 27. MPI_CHAR for printable characters is now defined for C type char (instead of signed char). This change should not have any impact on applications nor on MPI libraries (except some comment lines), because printable characters could and can be stored in any of the C types char, signed char, and unsigned char, and MPI_CHAR is not allowed for predefined reduction operations.
18 19 20 21	4.	Section 3.2.2 on page 27. MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_BOOL, MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, and MPI_C_LONG_DOUBLE_COMPLEX are now valid predefined MPI datatypes.
22 23 24 25 26 27	5.	Section 3.4 on page 42, Section 3.7.2 on page 54, Section 3.9 on page 82, and Section 5.1 on page 151. 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.
28 29 30	6.	Section 3.7 on page 52. The Advice to users for IBSEND and IRSEND was slightly changed.
31 32 33	7.	Section 3.7.3 on page 60. The advice to free an active request was removed in the Advice to users for MPI_REQUEST_FREE.
34 35 36	8.	Section 3.7.6 on page 72. MPI_REQUEST_GET_STATUS changed to permit inactive or null requests as input.
37 38 39 40	9.	Section 5.8 on page 178. "In place" option is added to MPI_ALLTOALL, MPI_ALLTOALLV, and MPI_ALLTOALLW for intracommunicators.
40 41 42 43 44	10.	Section 5.9.2 on page 186. 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.
45 46 47 48	11.	Section 5.9.2 on page 186. $MPI_(U)INT\{8,16,32,64\}_T$ are all considered C integer types for the purposes of the predefined reduction operators. MPI_AINT and MPI_OFFSET are considered Fortran

integer types. MPI_C_BOOL is considered a Logical type. $\mathbf{2}$ MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, and MPI_C_LONG_DOUBLE_COMPLEX are considered Complex types. 4 12. Section 5.9.7 on page 199. 5 The local routines MPI_REDUCE_LOCAL and MPI_OP_COMMUTATIVE have been 6 added. 8 13. Section 5.10.1 on page 201. 9 The collective function MPI_REDUCE_SCATTER_BLOCK is added to the MPI stan-10 dard. 11 14. Section 5.11.2 on page 205. 12Added in place argument to MPI_EXSCAN. 13 1415. Section 6.4.2 on page 269, and Section 6.6 on page 296. 15Implementations that did not implement MPI_COMM_CREATE on intercommuni-16cators will need to add that functionality. As the standard described the behav-17 ior of this operation on intercommunicators, it is believed that most implementa-18 tions already provide this functionality. Note also that the C++ binding for both 19 MPI_COMM_CREATE and MPI_COMM_SPLIT explicitly allow Intercomms. 2016. Section 6.4.2 on page 269. 21MPI_COMM_CREATE is extended to allow several disjoint subgroups as input if comm 22 is an intracommunicator. If comm is an intercommunicator it was clarified that all 23processes in the same local group of comm must specify the same value for group. 242517. Section 7.5.4 on page 336. 26New functions for a scalable distributed graph topology interface has been added. 27In this section, the functions MPI_DIST_GRAPH_CREATE_ADJACENT and 28 MPI_DIST_GRAPH_CREATE, the constants MPI_UNWEIGHTED, and the derived C++ 29 class Distgraphcomm were added. 30 18. Section 7.5.5 on page 343. 31For the scalable distributed graph topology interface, the functions 32 MPI_DIST_GRAPH_NEIGHBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS and 33 the constant MPI_DIST_GRAPH were added. 34 35 19. Section 7.5.5 on page 343. 36 Remove ambiguity regarding duplicated neighbors with MPI_GRAPH_NEIGHBORS 37 and MPI_GRAPH_NEIGHBORS_COUNT. 38 20. Section 8.1.1 on page 383. 39 The subversion number changed from 1 to 2. 40 41 21. Section 8.3 on page 390, Section 15.2 on page 700, and Annex A.1.3 on page 786. 42Changed function pointer typedef names MPI_{Comm,File,Win}_errhandler_fn to 43 MPI_{Comm,File,Win}_errhandler_function. Deprecated old "_fn" names. 4422. Section 10.2.4 on page 430. 4546Attribute deletion callbacks on MPI_COMM_SELF are now called in LIFO order. Imple-47mentors must now also register all implementation-internal attribute deletion callbacks

Unofficial Draft for Comment Only

on MPI_COMM_SELF before returning from MPI_INIT/MPI_INIT_THREAD.

1 2 3 4 5 6 7 8		Section 11.3.4 on page 504. 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.
9 10 11 12	24.	Section 12.2 on page 557. 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.
13 14 15 16 17 18	25.	Section 13.5.2 on page 617, and Table 13.2 on page 618. 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.
19 20 21 22 23 24 25	26.	Section 18.2.7 on page 763. 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 18.13, 18.14, and 18.15 on pages 764–766 explicitly detailing cross-language attribute behavior. Implementations that matched the behavior of the old example will need to be updated.
26 27	27.	Annex A.1.1 on page 771. Removed type MPI::Fint (compare MPI_Fint in Section A.1.2 on page 785).
28 29 30 31 32 33	28.	Annex A.1.1 on page 771. Table Named Predefined Datatypes. Added MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_BOOL, MPI_C_FLOAT_COMPLEX, MPI_C_COMPLEX, MPI_C_DOUBLE_COMPLEX, and MPI_C_LONG_DOUBLE_COMPLEX are added as predefined datatypes.
34	B.6	Changes from Version 2.0 to Version 2.1
35 36 37 38	1.	Section 3.2.2 on page 27, and Annex A.1 on page 771. In addition, the MPI_LONG_LONG should be added as an optional type; it is a syn- onym for MPI_LONG_LONG_INT.
39 40 41 42 43	2.	Section 3.2.2 on page 27, and Annex A.1 on page 771. 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.
44 45 46 47 48	3.	Section 3.2.5 on page 32. MPI_GET_COUNT with zero-length datatypes: The value returned as the count argument of MPI_GET_COUNT for a datatype of length zero where zero bytes have been transferred is zero. If the number of bytes transferred is greater than zero, MPI_UNDEFINED is returned.

4.	Section 4.1 on page 89.	1
	General rule about derived datatypes: Most datatype constructors have replication	2
	count or block length arguments. Allowed values are non-negative integers. If the	3
	value is zero, no elements are generated in the type map and there is no effect on	4
	datatype bounds or extent.	5
F	Castion 4.2 on name 146	6
э.	Section 4.3 on page 146. MPI_BYTE should be used to send and receive data that is packed using	7
	MPI_PACK_EXTERNAL.	8
	MFI_FACK_EXTERNAL.	9
6.	Section 5.9.6 on page 198.	10
	If comm is an intercommunicator in MPI_ALLREDUCE, then both groups should pro-	11 12
	vide count and datatype arguments that specify the same type signature (i.e., it is not	12
	necessary that both groups provide the same count value).	14
7	Section 6.3.1 on page 258.	15
	MPI_GROUP_TRANSLATE_RANKS and MPI_PROC_NULL: MPI_PROC_NULL is a valid	16
	rank for input to MPI_GROUP_TRANSLATE_RANKS, which returns MPI_PROC_NULL	17
	as the translated rank.	18
		19
8.	Section 6.7 on page 306.	20
	About the attribute caching functions:	21
	Advice to implementors. High-quality implementations should raise an er-	22
	ror when a keyval that was created by a call to MPI_XXX_CREATE_KEYVAL	23
	is used with an object of the wrong type with a call to	24
	MPI_YYY_GET_ATTR, MPI_YYY_SET_ATTR, MPI_YYY_DELETE_ATTR, or	25
	MPI_YYY_FREE_KEYVAL. To do so, it is necessary to maintain, with each key-	26
	val, information on the type of the associated user function. (End of advice to	27
	implementors.)	28 29
0	Castion 6.8 on none 200	29 30
9.	Section 6.8 on page 322. In MPI_COMM_GET_NAME: In C, a null character is additionally stored at	31
	name[resultlen]. resultlen cannot be larger then MPI_MAX_OBJECT_NAME-1. In For-	32
	tran, name is padded on the right with blank characters. resulten cannot be larger	33
	then MPI_MAX_OBJECT_NAME.	34
		35
10.	Section 7.4 on page 330.	36
	About MPI_GRAPH_CREATE and MPI_CART_CREATE: All input arguments must	37
	have identical values on all processes of the group of comm_old.	38
11	Section 7.5.1 on page 332.	39
	In MPI_CART_CREATE: If ndims is zero then a zero-dimensional Cartesian topology	40
	is created. The call is erroneous if it specifies a grid that is larger than the group size	41
	or if ndims is negative.	42
		43
12.	Section 7.5.3 on page 334.	44
	In MPI_GRAPH_CREATE: If the graph is empty, i.e., nnodes $== 0$, then	45
	MPI_COMM_NULL is returned in all processes.	46 47
		47
		10

1 2	13.	Section 7.5.3 on page 334. In MPI_GRAPH_CREATE: A single process is allowed to be defined multiple times
3		in the list of neighbors of a process (i.e., there may be multiple edges between two
4		processes). A process is also allowed to be a neighbor to itself (i.e., a self loop in the
5		graph). The adjacency matrix is allowed to be non-symmetric.
6		
7		Advice to users. Performance implications of using multiple edges or a non-
8		symmetric adjacency matrix are not defined. The definition of a node-neighbor
9		edge does not imply a direction of the communication. (<i>End of advice to users.</i>)
10	14.	Section 7.5.5 on page 343.
11		In MPI_CARTDIM_GET and MPI_CART_GET: If comm is associated with a zero-
12		dimensional Cartesian topology, MPI_CARTDIM_GET returns ndims=0 and
13		MPI_CART_GET will keep all output arguments unchanged.
14	1 5	
15 16	15.	Section 7.5.5 on page 343.
17		In MPI_CART_RANK: If comm is associated with a zero-dimensional Cartesian topol-
18		ogy, coord is not significant and 0 is returned in rank.
19	16.	Section 7.5.5 on page 343.
20		In MPI_CART_COORDS: If comm is associated with a zero-dimensional Cartesian
21		topology, coords will be unchanged.
22	17	Section 7.5.6 on page 251
23	11.	Section 7.5.6 on page 351. In MPI_CART_SHIFT: It is erroneous to call MPI_CART_SHIFT with a direction that
24		is either negative or greater than or equal to the number of dimensions in the Cartesian
25		communicator. This implies that it is erroneous to call MPI_CART_SHIFT with a
26		comm that is associated with a zero-dimensional Cartesian topology.
27		
28	18.	Section 7.5.7 on page 353.
29		In MPI_CART_SUB: If all entries in remain_dims are false or comm is already associ-
30		ated with a zero-dimensional Cartesian topology then newcomm is associated with a
31 32		zero-dimensional Cartesian topology.
33	18.1.	Section 8.1.1 on page 383.
34		The subversion number changed from 0 to 1.
35	10	
36	19.	Section 8.1.2 on page 385.
37		In MPI_GET_PROCESSOR_NAME: In C, a null character is additionally stored at
38		name[resultlen]. resultlen cannot be larger then MPI_MAX_PROCESSOR_NAME-1. In
39		Fortran, name is padded on the right with blank characters. resultlen cannot be larger then MPI_MAX_PROCESSOR_NAME.
40		
41	20.	Section 8.3 on page 390.
42		$MPI_{COMM,WIN,FILE}_{GET_{ERRHANDLER}} \text{ behave as if a new error handler object}$
43		is created. That is, once the error handler is no longer needed,
44		$MPI_ERRHANDLER_FREE$ should be called with the error handler returned from
45		MPI_ERRHANDLER_GET or MPI_{COMM,WIN,FILE}_GET_ERRHANDLER to mark
46		the error handler for deallocation. This provides behavior similar to that of
47		MPI_COMM_GROUP and MPI_GROUP_FREE.
48		

B.6. CHANGES FROM VERSION 2.0 TO VERSION 2.1

- 21. Section 10.2.1 on page 420, see explanations to MPI_FINALIZE. MPI_FINALIZE is collective over all connected processes. If no processes were spawned, accepted or connected then this means over MPI_COMM_WORLD; otherwise it is collective over the union of all processes that have been and continue to be connected, as explained in Section 10.10.4 on page 477.
- 22. Section 10.2.1 on page 420. About MPI_ABORT:

Advice to users. Whether the errorcode is returned from the executable or from the MPI process startup mechanism (e.g., mpiexec), is an aspect of quality of the MPI library but not mandatory. (*End of advice to users.*)

Advice to implementors. Where possible, a high-quality implementation will try to return the errorcode from the MPI process startup mechanism (e.g. mpiexec or singleton init). (End of advice to implementors.)

23. Section 9 on page 411.

An implementation must support info objects as caches for arbitrary (key, value) pairs, regardless of whether it recognizes the key. Each function that takes hints in the form of an MPI_Info must be prepared to ignore any key it does not recognize. This description of info objects does not attempt to define how a particular function should react if it recognizes a key but not the associated value. MPI_INFO_GET_NKEYS, MPI_INFO_GET_NTHKEY, MPI_INFO_GET_VALUELEN, and MPI_INFO_GET must retain all (key,value) pairs so that layered functionality can also use the Info object.

24. Section 11.3 on page 498.
MPI_PROC_NULL is a valid target rank in the MPI RMA calls MPI_ACCUMULATE, MPI_GET, and MPI_PUT. The effect is the same as for MPI_PROC_NULL in MPI point-to-point communication. See also item 25 in this list.

- 25. Section 11.3 on page 498. After any RMA operation with rank MPI_PROC_NULL, it is still necessary to finish the RMA epoch with the synchronization method that started the epoch. See also item 24 in this list.
- Section 11.3.4 on page 504.
 MPI_REPLACE in MPI_ACCUMULATE, like the other predefined operations, is defined only for the predefined MPI datatypes.
- 27. Section 13.2.8 on page 577. About MPI_FILE_SET_VIEW and MPI_FILE_SET_INFO: When an info object that specifies a subset of valid hints is passed to MPI_FILE_SET_VIEW or MPI_FILE_SET_INFO, there will be no effect on previously set or defaulted hints that the info does not specify.
- 28. Section 13.2.8 on page 577. 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.

 24

Unofficial Draft for Comment Only

1	29.	Section 13.3 on page 580 .
2		If a file does not have the mode MPI_MODE_SEQUENTIAL, then
3		MPI_DISPLACEMENT_CURRENT is invalid as disp in MPI_FILE_SET_VIEW.
4		
	30	Section 13.5.2 on page 617.
5	50.	* 0
6		The bias of 16 byte doubles was defined with 10383. The correct value is 16383.
7	91	MDI 2.2 Section 16.1.4 (Section was removed in MDI 2.0)
8	51.	MPI-2.2, Section 16.1.4 (Section was removed in MPI-3.0).
9		In the example in this section, the buffer should be declared as const void* buf.
10	20	
	32.	Section 18.1.9 on page 725.
11		About MPI_TYPE_CREATE_F90_XXX:
12		
13		Advice to implementors. An application may often repeat a call to
14		$MPI_TYPE_CREATE_F90_XXX$ with the same combination of (XXX,p,r) . The
15		application is not allowed to free the returned predefined, unnamed datatype
16		handles. To prevent the creation of a potentially huge amount of handles, the
17		MPI implementation should return the same datatype handle for the same (
		REAL/COMPLEX/INTEGER,p,r) combination. Checking for the combination (
18		p,r) in the preceding call to MPI_TYPE_CREATE_F90_XXX and using a hash-
19		
20		table to find formerly generated handles should limit the overhead of finding
21		a previously generated datatype with same combination of (XXX,p,r) . (End of
22		advice to implementors.)
23	<u></u>	Casting A 1.1 and a 771
24	აა.	Section A.1.1 on page 771.
25		MPI_BOTTOM is defined as void * const MPI::BOTTOM.
26		
27		
28		
29		
30		
31		
32		
33		
34		
35		
36		
37		
38		
39		
40		
41		
42		
43		
44		
45		
46		
47		

Bibliography

[1]	Reverse domain name notation convention. https://docs.oracle.com/javase/tutorial, 6.4.2	/java/package
[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, 1.2	13 14 15 16 17
[3]	scalable, and portable collective communication libraries. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint. 1.2	18 19 20
[4]	Purushotham V. Bangalore, Nathan E. Doss, and Anthony Skjellum. MPI++: Issues and Features. In <i>OON-SKI '94</i> , page in press, 1994. 6.1	21 22 23
[5]	A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam.	24 25 26 27
[6]	portable parallel programming and their implementation on the Intel	28 29 30
	Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. <i>IJHPCN</i> , 1(1/2/3):9199, 2004, 11.7	31 32 33 34
[8]	and evaluation of primitives for parallel I/O. In Proceedings of Supercomputing '93, pages 452461, 1993. 13.1	35 36 37 38
[9]	R. Butler and E. Lusk. User's guide to the p4 programming system.	39 40
[10]] Ralph Butler and Ewing Lusk. Monitors, messages, and clusters: The p4 parallel programming system. <i>Parallel Computing</i> , 20(4):547564, April 1994. Also Argonne National Laboratory Mathematics and Computer Science Division preprint P362-0493. 1.2	41 42 43 44 45
[11]] Robin Calkin, Rolf Hempel, Hans-Christian Hoppe, and Peter Wypior. Portable programming with the PARMACS message-passing library. <i>Parallel</i> <i>Computing</i> , 20(4):615632, April 1994. 1.2	46 47 48

BIBLIOGRAPHY

1 2 3 4	[12] S. Chittor and R. J. Enbody. Performance evaluation of mesh-connected wormhole-routed networks for interprocessor communication in multicomputers. In Proceedings of the 1990 Supercomputing Conference, pages 647656, 1990. 7.1
5 6 7 8 9	[13] S. Chittor and R. J. Enbody. Predicting the effect of mapping on the communication performance of large multicomputers. In Proceedings of the 1991 International Conference on Parallel Processing, vol. II (Software), pages II1II4, 1991. 7.1
10 11 12	<pre>[14] Parasoft Corporation. Express version 1.0: A communication environment for parallel computers, 1988. 1.2</pre>
13 14 15 16 17	 [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. 17, 37
18 19 20 21 22 23	[16] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In IPPS '93 Workshop on Input/Output in Parallel Computer Systems, pages 5670, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 3138. 13.1
24 25 26 27 28	[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 282291. 6.4.2
29 30 31	<pre>[18] J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. framework supports heterogeneous network computing. Physics, 7(2):16675, April 1993. 1.2</pre> Integrated PVM Computers in
32 33 34 35 36	[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. 1.2
37 38 39	[20] Edinburgh Parallel Computing Centre, University of Edinburgh. CHIMP Concepts, June 1991. 1.2
40 41	[21] Edinburgh Parallel Computing Centre, University of Edinburgh. CHIMP Version 1.0 Interface, May 1992. 1.2
42 43 44 45	[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. 6.1.2
46 47 48	[23] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. The International Journal of Supercomputer Applications and High Performance Computing, 8, 1994. 1.3

1 [24] Message Passing Interface Forum. MPI: A Message-Passing $\mathbf{2}$ Interface standard (version 1.1). Technical report, 1995. 3 http://www.mpi-forum.org. 1.3 4 [25] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Bob Manchek, 5and Vaidy Sunderam. PVM: Parallel Virtual Machine---A User's Guide and 6 Tutorial for Network Parallel Computing. MIT Press, 1994. 10.1 7 8 [26] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. PICL: A 9 portable instrumented communications library, C reference manual. 10 Technical Report TM-11130, Oak Ridge National Laboratory, Oak Ridge, 11 TN, July 1990. 1.2 12[27] D. Gregor, T. Hoefler, B. Barrett, and A. Lumsdaine. Fixing probe 13 for multi-threaded MPI applications. Technical Report 674, Indiana 14University, Jan. 2009. 3.8.2 1516[28] William D. Gropp and Barry Smith. Chameleon parallel programming tools 17users manual. Technical Report ANL-93/23, Argonne National Laboratory, 18 March 1993. 1.2 19 [29] Michael Hennecke. A Fortran 90 interface to MPI version 1.1. Technical 2021Report Internal Report 63/96, Rechenzentrum, Universität Karlsruhe, 22 D-76128 Karlsruhe, Germany, June 1996. 18.1.3 23[30] T. Hoefler, G. Bronevetsky, B. Barrett, B. R. de Supinski, and 24 Efficient MPI support for advanced hybrid programming A. Lumsdaine. 25models. In Recent Advances in the Message Passing Interface 26(EuroMPI'10), volume LNCS 6305, pages 50--61. Springer, Sep. 2010. 273.8.1, 3.8.2 2829[31] T. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm. Optimizing 30 a conjugate gradient solver with non-blocking collective operations. 31Elsevier Journal of Parallel Computing (PARCD), 33(9):624--633, Sep. 32 2007. 5.12 33 [32] T. Hoefler, F. Lorenzen, and A. Lumsdaine. Sparse non-blocking 34 collectives in quantum mechanical calculations. In Recent Advances in 35 Parallel Virtual Machine and Message Passing Interface, 15th European 36 PVM/MPI Users' Group Meeting, volume LNCS 5205, pages 55--63. Springer, 37 Sep. 2008. 7.6 38 39 [33] T. Hoefler and A. Lumsdaine. Message progression in parallel computing 40 --- to thread or not to thread? In Proceedings of the 2008 IEEE 41 International Conference on Cluster Computing. IEEE Computer Society, 42Oct. 2008. 5.12 43 44[34] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and performance 45analysis of non-blocking collective operations for MPI. In Proceedings 46of the 2007 International Conference on High Performance Computing, 47Networking, Storage and Analysis, SCO7. IEEE Computer Society/ACM, Nov. 48 2007. 5.12

Unofficial Draft for Comment Only

934	BIBLIOGRAPHY
(]]	T. Hoefler, M. Schellmann, S. Gorlatch, and A. Lumsdaine. Communication optimization for medical image reconstruction algorithms. In Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting, volume LNCS 5205, pages 7583. Springer, Sep. 2008. 5.12
]	T. Hoefler and J. L. Traeff. Sparse collective operations for MPI. In Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium, HIPS'09 Workshop, May 2009. 7.6
c	Torsten Hoefler and Marc Snir. Writing parallel libraries with MPI common practice, issues, and extensions. In Cotronis et al. [15], pages 345355. 6.4.2
E E	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. 5.13
2	Institute of Electrical and Electronics Engineers, New York. IEEE Standard for Binary Floating-Point Arithmetic, IEEE Standard 754-2008, 2008. 13.5.2
ع ب	International Organization for Standardization, Geneva. ISO 8859-1:1987: Information processing 8-bit single-byte coded graphic character sets Part 1: Latin alphabet No. 1, February 1987. 13.5.2

- [41] International Organization for Standardization, Geneva. ISO/IEC 9945-1:1996: Information technology --- Portable Operating System Interface (POSIX) --- Part 1: System Application Program Interface (API) [C Language], December 1996. 10.6, 13.2.1
 - [42] International Organization for Standardization, Geneva. ISO/IEC 1539-1:2010: Information technology --- Programming languages ---Fortran --- Part 1: Base language, November 2010. 18.1.1, 18.1.2
- ISO/IEC TS [43] International Organization for Standardization, Geneva. 29113:2012: Information technology --- Further interoperability of Fortran with C, December 2012. 18.1.1, 18.1.1, 18.1.2, 18.1.7, 28
- [44] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy 39 L. Steele Jr., and Mary E. Zosel. The High Performance Fortran 40Handbook. MIT Press, 1993. 4.1.4 41
- [45] David Kotz. Disk-directed I/O for MIMD multiprocessors. In Proceedings of the 1994 Symposium on Operating Systems Design and Implementation, 44 pages 61--74, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994. 13.1
- 47[46] O. Krämer and H. Mühlenbein. Mapping strategies in message-based 48 multiprocessor systems. Parallel Computing, 9:213--225, 1989. 7.1

1

 $\mathbf{2}$

11

1213

1415

16

1718

19

202122

27

28

29

30 31

32

33

3435

36

37

38

42

43

45

- [47] 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. Avialable online: https://docs.freebsd.org/44doc/psd/21.ipc/paper.pdf. 10.10.5
- [48] Bradley Morgan, Daniel J. Holmes, Anthony Skjellum, Purushotham Bangalore, and Srinivas Sridharan. Planning for performance: Persistent collective operations for MPI. In Proceedings of the 24th European MPI Users' Group Meeting, EuroMPI '17, pages 4:1--4:11, New York, NY, USA, 2017. ACM. 5.13
- [49] nCUBE Corporation. nCUBE 2 Programmers Guide, r2.0, December 1990. 1.2
- Performance of the iPSC/860 Concurrent File System. [50] Bill Nitzberg. Technical Report RND-92-020, NAS Systems Division, NASA Ames, December 1992. 13.1
- [51] William J. Nitzberg. Collective Parallel I/O. PhD thesis, Department of Computer and Information Science, University of Oregon, December 1995. 13.1
- [52] 4.4BSD Programmer's Supplementary Documents (PSD). O'Reilly and Associates, 1994. 10.10.5
- [53] Paul Pierce. The NX/2 operating system. In Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, pages 384--390. ACM Press, 1988. 1.2
- [54] 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. 14.2.8
- [55] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In Proceedings of Supercomputing '95, December 1995. 13.1
- [56] A. Skjellum and A. Leung. Zipcode: a portable multicomputer communication library atop the reactive kernel. In D. W. Walker and Q. F. Stout, editors, Proceedings of the Fifth Distributed Memory Concurrent Computing Conference, pages 767--776. IEEE Press, 1990. 1.2, 6.1.2
- [57] A. Skjellum, S. Smith, C. Still, A. Leung, and M. Morari. The Zipcode message passing system. Technical report, Lawrence Livermore National Laboratory, September 1992. 1.2
- 45[58] Anthony Skjellum, Nathan E. Doss, and Purushotham V. Bangalore. Writing Libraries in MPI. In Anthony Skjellum and Donna S. Reese, editors, Proceedings of the Scalable Parallel Libraries Conference, 48 pages 166--173. IEEE Computer Society Press, October 1993. 6.1

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

56

7

8

9

10

11

12

13 14

15

16

17

18 19

20

21

22

23

2425

26

27

28

29

30

3132

33

34

35

36

37

38

39

40 41

42

43

44

46

1	[59] Anthony Skjellum, Nathan E. Doss, and Kishore Viswanathan.
2	Inter-communicator extensions to MPI in the MPIX (MPI eXtension)
3	Library. Technical Report MSU-940722, Mississippi State
4	University Dept. of Computer Science, August 1994. Archived at
5	http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.6283. 5.2.2
6	
7	[60] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P. Leung,
8	and Manfred Morari. The Design and Evolution of Zipcode. Parallel
9	Computing, 20(4):565596, April 1994. 6.1.2, 6.5.6
10	[C1] The Internet Contents NDD. Enternel Date Democratic Charles I Mar
11	[61] The Internet Society. XDR: External Data Representation Standard, May
12	2006. http://www.rfc-editor.org/pdfrfc/rfc4506.txt.pdf. 13.5.2
13	[62] Rajeev Thakur and Alok Choudhary. An extended two-phase method for
14	accessing sections of out-of-core arrays. Scientific Programming,
15	5(4):301317, Winter 1996. 13.1
16	0(1).001 017, WINDER 1000. 10.1
17	[63] The Unicode Standard, Version 2.0. Addison-Wesley, 1996. ISBN
18	0-201-48345-9. 13.5.2
19	
20	[64] D. Walker. Standards for message passing in a distributed memory
21	environment. Technical Report TM-12147, Oak Ridge National Laboratory,
22	August 1992. 1.2
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	

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, 16, 108, 743 access epoch, 518action in function names, $\underline{10}$ active, 60, 328 active target communication, 518 addresses, 123 absolute, 16, 108, 743 correct use, 123 relative displacement, 16, 108 alignment, 388, 487, 489, 914 all-reduce, 198 nonblocking, 222 persistent, 240 all-to-all, 178 nonblocking, 218 persistent, 236 array arguments, 14 assertions, **532** ASYNCHRONOUS Fortran attribute, 744 attribute, 255, 306, 763 caching, 254backward incompatibilies, 705 barrier synchronization, 157 nonblocking, 210 persistent, 228 blocking, <u>11</u>, <u>42</u>, 45, 583 I/O, 584 bounds of datatypes, 114 broadcast, 158 nonblocking, 210 persistent, 228 buffer allocation, 49 buffered, <u>42</u>, 52, <u>54</u> nonblocking, 53 buffered send, 45 \mathbf{C}

language binding, **18** caching, **253**, <u>254</u>, **306**

callback functions	12
language interoperability, 762	13
prototype definitions, 786	14
deprecated, 792	15
cancel, 19, 61, <u>73</u> , 80 , 428, 700, 914	16
canonical pack and unpack, 146	
Cartesian	17
topology, <u>330</u> , 332	18
change-log, 913	19
choice, 16	20
class	21
in function names, $\underline{10}$	22
clock synchronization, 386	23
collective, <u>11</u> , <u>583</u>	24
$\operatorname{split}, 624$	
collective communication, 151	25
correctness, 245	26
file data access operations, 602	27
neighborhood, 355	28
nonblocking, 208	29
commit, <u>117</u>	30
COMMON blocks, 747	31
communication, 481	32
collective, 151	33
modes, 42	
one-sided, 481	34
point-to-point, 25	35
RMA, 481	36
communicator, <u>29</u> , 253 , <u>254</u>	37
hidden, 154, 247, 526	38
completes	39
operation, $\underline{11}$	40
completion, 60	41
multiple, 65	42
connected, $\frac{477}{15}$	
constants, 15 , 767 , 771	43
context, 253 , 254 , 256 control variables	44
tools interface, 654	45
	46
conversion, 40 counts, 17	47
	48

1 create $\mathbf{2}$ in function names, 10 3 data, 27 4 data conversion, 40 5datatypes, 89, 761 6 default file error, 635 7 delete in function names, 10 8 deprecated interfaces, 17, 697 9 derived datatype, 12, 89, 739 10 disconnected, 478 11 displacement, 567, 581 12distributed graph 13 topology, <u>330</u>, <u>336</u> 14Dynamic Process Model, 419 15dynamically attached memory, 490 16elementary datatype, 567, 581 17empty, 6018 end of file, 569 19envelope, 25, 29 20environmental inquiries, 385 21equivalent datatypes, 12 error handling, 20, 390 22 default file error handler, 570, 573, 635 23error codes and classes, 401, 402 24error handlers, 393, 402, 763 25finalize, 21, 429, 478 26I/O, 634, 635 initial error handler, <u>21</u>, 391, 421, 429, 462 2728one-sided communication, 534 process failure, 21, 478 29 program error, 21 30 resource error, 21 31 startup, 21, 421, 462 32 transmission failure, 2033 establishing communication, 464 34etype, 567, 581 35 events tools interface, 673 36 exception, 390 37 exclusive scan 38 nonblocking, 226 39 persistent, 244 40 explicit offsets, 584, 586 41 exposure epoch, 51842extent of datatypes, <u>90</u>, <u>113</u>, **114** 43 true extent, **116** external32 44 file data representation, 615 45extra-state, 767 46 47fairness, 47, 70 48 file, 567

data access, 583 collective operations, 602 explicit offsets, 586 individual file pointers, 591 seek, 604 shared file pointers, 599 split collective, 606 end of file, 569filetype, 568 handle, 569 interoperability, 613 manipulation, 569 offset, 16, 568 pointer, 569size, <u>569</u> view, 567, <u>568</u>, **580** file size. 630 finalize, **426** finished, 429 Fortran language binding, 18, 707 Fortran support, 707 gather, 159 nonblocking, 212 persistent, 229 gather-to-all, 175 nonblocking, 216 persistent, 234 general datatype, 89 generalized requests, 557, 557 get in function names, 10graph topology, <u>330</u>, **334** group, 253, 254, 256, 297 group objects, 256 handles, 12, 756 hardware resource type, 281, 282 host rank, 385 immediate, 54 inactive, 60 inclusive scan, **204**

nonblocking, 225

individual file pointers, 584, 591

persistent, 243

file info, 577

keys, 793

values, 794

initial error handler, 21

independent, 478

info object, **411**

initiation, 54

Unofficial Draft for Comment Only

General Index

inter-communication, 255, 296 inter-communicator, 255, 296 collective operations, 155, 156 interlanguage communication, 768 internal file data representation, $\underline{614}$ interoperability, 613 intra-communication, 255, 296 intra-communicator, 254, 296 collective operations, 154intra-communicator objects, 257 I/O, 567 IO rank, 386 isin function names, 10 language binding, 17, 707 interoperability, 755 summary, 771 lb_marker, 102, 106, $\underline{112}$, 112, 117 erased, $\underline{115}$ local, <u>11</u>, 42 local group, 270loosely synchronous model, 327 lower bound, 112lower-bound markers, 112 macros, 20 main thread, 449matched receives, 78 matching type, **38**, <u>120</u>, **629** matching probe, **76** memory alignment, 388, 487, 489, 914 allocation, 387, 485, 487 system, 12 memory model, 482, 517 separate, 482, 489 unified, 482, 489 message, 25data, 27 envelope, 29 modes, 42module variables, 747 mpi module Fortran support, **711** MPI process initialization, 419 Dynamic Process Model, 419 Sessions Model, 570 Sessions Model, 257, 260, 385, 419, 450 World Model, 257, 260, 385, 419, 449,

570, 635

mpi_f08 module

Fortran support, 708	1
MPI_SIZEOF and storage_size(), 19, 701,	2
731–733	3
mpiexec, 422, <u>446</u> , 446	4
mpif.h include file	5
Fortran support, 713	6
mpirun, 446	7
multiple completions, 65	
named datatype, $\underline{12}$	8
names, 465	9
names, 465 name publishing, 469	10
naming objects, 322	11
native	12
file data representation, 614	13
neighborhood collective communication, 355	14
nonblocking, 366	15
non-local, <u>11</u> , <u>42</u> , <u>43</u>	16
nonblocking, <u>11</u> , 52 , 498, 583	17
communication, $\underline{52}$	18
completion, 60	19
Fortran problems, 742	20
I/O, 584	20
initiation, 54	
request objects, 53	22
null handle, <u>60</u>	23
null processes, 88	24
offset, 16 , <u>568</u>	25
one-sided communication, 481	26
Fortran problems, 743	27
opaque objects, 12 , 761	28
operation completes, $\underline{11}$	29
origin, <u>482</u>	30
1 100	31
pack, 139	32
canonical, 146 packing unit, <u>142</u>	33
parallel procedure, <u>327</u>	34
parallel procedure, $\underline{521}$ passive target communication, $\underline{518}$	35
performance variables	36
tools interface, 661	
persistent communication requests, 63, 82	37
collective persistent, 226, 371	38
Fortran problems, 743	39
PMPI_, <u>641</u>	40
point-to-point communication, 25	41
portable datatype, $\underline{12}$	42
ports, 465	43
predefined datatype, $\underline{12}$	44
predefined reduction operations, 186	45
private window copy, 516	46
probe, 73	47
probe, matching, 76	48
process creation, 419	

General Index

940

1 process failures, 21 $\mathbf{2}$ process group, 30processes, 20 3 processor name, **386** 4 profiling interface, 641 $\mathbf{5}$ program error, 21 6 progress, 209 7 prototype definitions, 786 8 deprecated, 792 9 public window copy, 516 10 rank, 256 11 ready, 43, 52, 54 12nonblocking, 5213 ready send, 45 14receive, 25, 26, 30 15buffer, 2616complete, 52context, 297 17start call, 52 18 reduce, **184** 19 nonblocking, 221 20persistent, 239 21reduce-scatter. 201 22nonblocking, 223, 224 23persistent, 241, 242 reduction operations, 184, 763 24 predefined, 186 25process-local, 199 26scan, **204** 27user-defined, 193 28related, 142 29relative displacement, 16, 108 30 remote group, 270 31 Remote Memory Access, see RMA removed interfaces, 17, 703 32 request complete 33 I/O, 58434 request objects, 53 35 resource error, 2136 RMA, 481 37 communication calls, 498 38 request-based, 511 memory model, **516** 39 synchronization calls, 518 40 41 scan, 204 42inclusive, 204 43 scatter. 169 44 nonblocking, 214 persistent, 232 45seek, 604 46 semantics 47collective communications, 152 48

file collective, 628 collective access, 602 conflicting access, 625 consistency, 624 explicit offsets, 586 nonblocking collective, 628 shared file pointer, 600 split collective, 624 nonblocking communications, 64 point-to-point communication, 46 semantics and correctness one-sided communication, 534 send, <u>25</u>, **26** buffer, 25complete, 52context. 297 start, 52 send-receive, **35** nonblocking, 58, 59 separate memory model, 482, 489, 517 sequential storage, 123Sessions Model. 570 Sessions Model, 257, 260, 385, 419, 450 set in function names, 10shared file pointers, 584, 599 shared memory allocation, 487 signals, 22singleton init, 476 size changing I/O, 630 source, 297 split collective, 583, 606 standard, $\underline{42}$, 52nonblocking, 53 standard send, 45 started, 429 starting processes, 452, 454 startup, 420 portable, 446 state, 14 status, 32, 758 associating information, 564 ignore, 34 test, 72 strong synchronization, 519 synchronization, 481, 498 synchronization calls RMA, 518 synchronous, 42, 52, 54, 60 nonblocking, 53 synchronous send, 45system memory, $\underline{12}$

Unofficial Draft for Comment Only

tag values, 385	1
target, $\underline{482}$	2
thread compliant, $\underline{424}$, $\underline{448}$	3
threads, 448	4
timers and synchronization, 409	5
tool information interface, 647	6
tool support, 641	
topologies, 329	7
topology	8
Cartesian, <u>330</u> , 332	9
distributed graph, <u>330</u> , 336	10
graph, 330 , 334	11
virtual, 330	12
transmission failures, $\underline{20}$	13
true extent of datatypes, 116	14
type map, <u>90</u>	
type matching, 38 , <u>120</u>	15
type signature, $\underline{90}$	16
types, 785	17
	18
ub_marker, 102, 106, 107, <u>112</u> , 112, 117	19
erased, $\underline{115}$	20
unified memory model, 482, 489, <u>517</u>	21
universe size, 475	22
unnamed datatype, <u>12</u>	
unpack, 139	23
canonical, 146	24
upper bound, <u>112</u>	25
upper-bound markers, 112	26
user functions at process termination, 430	27
user-defined data representations, 620	28
user-defined reduction operations, 193	29
verbosity levels	30
tools interface, 648	
version inquiries, 383	31
view, 567, <u>568</u> , 580	32
virtual topology, 254, 255, 330	33
$\frac{1}{2}$	34
weak synchronization, 519	35
window	36
allocation, 485	37
creation, 483	38
dynamically attached memory, 490	39
shared memory allocation, 487	
World Model, 257, 260, 385, 419, 449, 570,	40
635	41
	42
	43
	44
	45
	46

Examples Index

10

11 12 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.

13	ASYNCHRONOUS, 551, 751, 753 Attributes between languages, 764
14	Attributes between languages, 704
15	Blocking/Nonblocking collectives do not
16	match, 249
17	,
18	C/Fortran handle conversion, 757
19	Cartesian virtual topologies, 377
20	Client-server code, 71
21	with blocking probe, 75
22	with blocking probe, wrong, 75
23	Datatype
24	3D array, 131
25	absolute addresses, 136
26	array of structures, 133
27	elaborate example, 144, 145
28	matching type, 120
29	matrix transpose, 132
30	union, 137
	Datatypes
31	matching, <u>38</u>
32	not matching, 39
33	untyped, 39 Deadlock
34	if not buffered, 48
35	with MPI_Bcast, 245
36	wrong message exchange, 48
37	
38	False matching of collective operations, 248
39	Fortran 90
40	copying and sequence problem, 735, 737,
41	738 derived tensor 720
42	derived types, 739
43	heterogeneous communication (unsafe), 732
44	invalid KIND, 728
45	MPI_TYPE_MATCH_SIZE
46	implementation, 731
47	overlapping communication and
48	computation, 751, 752, 754
10	

register optimization, 742-744

Independence of nonblocking operations, 251 Intercommunicator, 274, 278 Interlanguage communication, 768 Intertwined matching pairs, 46 Message exchange (ping-pong), 48

Mixing blocking and nonblocking collective operations, 247 Mixing collective and point-to-point requests, 250MPI_ACCUMULATE, 506 MPI_Accumulate, 549, 550, 553 MPI_Aint, 133 MPI_Aint_add, 553 MPI_Allgather, 178 MPI_ALLOC_MEM, 389, 390 MPI_Alloc_mem, 390, 553 MPI_ALLREDUCE, 199 MPI_Alltoall, 249 MPI_ASYNC_PROTECTS_NONBLOCKING, 551MPI_BARRIER, 442 MPI_Barrier, 428, 539-542, 548-550 MPI_Bcast, 159, 245-248 MPI_BSEND, 46 MPI_Buffer_attach, 50, 428 MPI_Buffer_detach, 50 MPI_BYTE, 39 MPI_Cancel, 428 MPI_CART_COORDS, 352 MPI_CART_GET, 377 MPI_CART_RANK, 352 MPI_CART_SHIFT, 352, 377 MPI_CART_SUB, 353 MPI_CHARACTER, 40 MPI_Comm_accept, 472, 473 MPI_Comm_connect, 472, 473 MPI_Comm_create, 274, 290, 291, 294

MPI_COMM_CREATE_FROM_GROUP, 438, 442 MPI_Comm_create_keyval, 320 MPI_Comm_dup, 293 MPI_Comm_get_attr, 320 MPI_Comm_get_parent, 463 MPI_Comm_group, 274, 294, 320 MPI_Comm_remote_size, 278 MPI_Comm_set_attr, 320 MPI_COMM_SPAWN, 456 MPI_Comm_spawn, 456, 463 MPI_COMM_SPAWN_MULTIPLE, 461 MPI_Comm_spawn_multiple, 461 MPI_Comm_split, 278, 303, 304 MPI_Comm_split_type, 282, 283 MPI_Compare_and_swap, 550, 553 MPI_DIMS_CREATE, 333, 377 MPI_DIST_GRAPH_CREATE, 341 MPI_Dist_graph_create, 342 MPI_DIST_GRAPH_CREATE_ADJACENT, 341 MPI_F_sync_reg, 551 MPI_FILE_CLOSE, 592, 595 MPI_FILE_GET_AMODE, 576 MPI_FILE_IREAD, 595 MPI_FILE_OPEN, 592, 595 MPI_FILE_READ, 592 MPI_FILE_SET_ATOMICITY, 630 MPI_FILE_SET_VIEW, 592, 595 MPI_FILE_SYNC, 631 MPI_Finalize, 427–429 MPI_FREE_MEM, 389, 390 MPI_Free_mem, 553 MPI_Gather, 145, 162, 163, 167 MPI_Gatherv, 145, 164–167 MPI_GET, 502, 503 MPI_Get, 539-542, 547, 548 MPI_Get_accumulate, 549, 550, 553 MPI_GET_ADDRESS, 109, 739, 740, 761 MPI_Get_address, 133, 136, 137, 144 MPI_GET_COUNT, 122 MPI_GET_ELEMENTS, 122 MPI_GRAPH_CREATE, 335, 348 MPI_GRAPH_NEIGHBORS, 348 MPI_GRAPH_NEIGHBORS_COUNT, 348 MPI_Grequest_complete, 562 MPI_Grequest_start, 562 MPI_Group_excl, 290 MPI_Group_free, 274, 290, 291 MPI_GROUP_FROM_SESSION_PSET, 438, 441.442 MPI_Group_incl, 274, 291, 294 MPI_Iallreduce, 250 MPI_Ialltoall, 249

$MPI_Ibarrier, 247-250$	1
MPI_Ibcast, 211, 250, 251	2
MPI_INFO_CREATE, 438	3
MPI_INFO_ENV, 423	4
MPI_INFO_SET, 438	
MPI_Intercomm_create, 303, 304	5
MPI_Iprobe, 428	6
MPI_IRECV, 62–64, 71	7
MPI_Irecv, 250	8
MPI_ISEND, 62–64, 71	9
MPI_Op_create, 196, 197, 206	1(
MPI_Open_port, 472, 473	11
MPI_Pack, 143–145	
MPI_Pack_size, 145	12
MPI_PROBE, 75	13
MPI_Publish_name, 473	14
MPI_Put, 524, 530, 540–542, 547, 548	15
MPI_RECV, 38–40, 46, 48, 64, 75, 120	16
MPI_Recv, 249	17
MPI_REDUCE, 188, 191	
MPI_Reduce, 191, 192, 196, 197	18
MPI_REQUEST_FREE, 63	19
MPI_Request_free, 428	20
MPI_Rget, 552	21
MPI_Rput, 552	22
	23
MPI_Scan, 206	24
MPI_Scatter, 172	
MPI_Scattery, 173, 174	25
MPI_SEND, 38–40, 48, 64, 75, 120 MPI_Send_122, 126, 127, 144, 240, 250	26
MPI_Send, 133, 136, 137, 144, 249, 250	27
MPI_SENDRECV, 131, 132	28
MPI_SENDRECV_REPLACE, 352	29
MPI_SESSION_FINALIZE, 438, 441, 442	30
MPI_SESSION_GET_NTH_PSET, 441, 442	31
MPI_SESSION_GET_NUM_PSETS, 441, 442	
MPI_SESSION_INIT, 438, 441, 442	32
MPI_SSEND, 46, 64	33
MPI_Test_cancelled, 428	34
MPI_TYPE_COMMIT, 118, 131, 132, 502,	35
739, 740	36
MPI_Type_commit, 133, 136, 137, 144,	37
163–167, 174, 206	38
MPI_TYPE_CONTIGUOUS, 92, 112, 120, 122	
MPI_Type_contiguous, 163	39
MPI_TYPE_CREATE_DARRAY, 108	40
MPI_TYPE_CREATE_HVECTOR, 131, 132	41
MPI_Type_create_hvector, 133, 136	42
MPI_TYPE_CREATE_INDEXED_BLOCK,	43
502	44
MPI_TYPE_CREATE_RESIZED, 739, 740	45
MPI_TYPE_CREATE_STRUCT, 100, 112,	
132, 739, 740	46
MPI_Type_create_struct, 133, 136, 137, 144,	47
166, 167, 206	48

1	MPI_TYPE_CREATE_SUBARRAY, 639
2	MPI_TYPE_EXTENT, 502
3	MPI_TYPE_FREE, 502
	MPI_Type_get_contents, 138
4	MPI_Type_get_envelope, 138
5	MPI_TYPE_GET_EXTENT, 131, 132, 503,
6	506
7	
8	MPI_Type_get_extent, 133
	MPI_TYPE_INDEXED, 96, 131
9	MPI_Type_indexed, 133, 136
10	MPI_TYPE_VECTOR, 93, 131, 132
11	MPI_Type_vector, 164, 165, 167, 174
12	MPI_Unpack, 144, 145
13	MPI_Unpublish_name, 473
14	MPI_User_function, 197
	MPI_WAIT, 62–64, 71, 595
15	MPI_Wait, 247–250
16	MPI_Waitall, 250, 552
17	MPI_WAITANY, 71
18	MPI_Waitany, 552
19	MPI_WAITSOME, 71
	MPI_Win_attach, 553
20	MPI_Win_complete, 524, 542, 548
21	MPI_WIN_CREATE, 502, 503, 506
22	MPI_Win_create_dynamic, 553
23	MPI_Win_detach, 553
24	MPI_WIN_FENCE, 502, 503, 506
25	MPI_Win_fence, 547
	MPI_Win_flush, 541, 549, 550, 553
26	MPI_Win_flush_all, 550
27	
28	MPI_Win_flush_local, 540
29	MPI_WIN_FREE, 503, 506
30	MPI_Win_lock, 530, 539–542
31	MPI_Win_lock_all, 551–553
	MPI_Win_post, 542, 548
32	MPI_Win_start, 524, 542, 548
33	MPI_Win_sync, 540, 541, 549, 550
34	shared memory windows, 551
35	MPI_Win_unlock, 530, 539–542
36	MPI_Win_unlock_all, 552, 553
37	MPI_Win_wait, 542, 548
	mpiexec, 423, 447, 448
38	
39	Neighborhood collective communication, 377
40	Non-deterministic program with MPI_Bcast,
41	246
42	Non-overtaking messages, 46
	Nonblocking operations, $62, 63$
43	message ordering, 64
44	progress, 64
45	
46	Overlapping Communicators, 250
47	Dipolining pophloching collective
48	Pipelining nonblocking collective operations,
	250

Profiling interface implementation using the C macro preprocessor, 645 implementation using weak symbols, 644 measurement wrapper, 644 Progression of nonblocking collective operations, 249

Shared memory windows MPI_Win_sync, 551

Threads and MPI, 449
Tool information interface basic usage of performance variables, 671 listing names of all control variables, 657 reading the value of a control variable, 660
Topologies, 377
Typemap, 91–93, 96, 100, 108

Virtual topologies, 377

MPI Constant and Predefined Handle Index

This index lists predefined MPI constants and handles. Underlined page numbers give the location of the primary definition or use of the indexed term.

MPI::_LONG_LONG, 918 MPI::BOOL, 918 MPI::COMPLEX, 918 MPI::DOUBLE_COMPLEX, 918 MPI::F_COMPLEX16, 919 MPI::F_COMPLEX32, 919 MPI::F_COMPLEX4, 919 MPI::F_COMPLEX8, 919 MPI::INTEGER16, 919 MPI::LONG_DOUBLE_COMPLEX, 918 MPI::LONG_LONG, 918 MPI::REAL16, 919 MPI_2DOUBLE_PRECISION, 190, 777 MPI_2INT, 190, 777 MPI_2INTEGER, 190, 777 MPI_2REAL, 190, 777 MPI_ADDRESS_KIND, 15, 16, 16, 29, 306, 734, 764, 774 MPI_AINT, 28, 29, 187, 492, 618, 775, 776, 924.926 MPI_ANY_SOURCE, 30, <u>31</u>, 31, 36, 37, 46, 57-60, 73-78, 85, 287, 328, 386, 773 MPI_ANY_TAG, 15, 30, <u>31</u>, 31, 33, 36, 37, 57-60, 73, 74, 76-80, 85, 88, 287, 773, 921 MPI_APPNUM, 477, 780 MPI_ARGV_NULL, 16, 456, 460, 734, 782 MPI_ARGVS_NULL, 16, 460, 734, 782 MPI_ASYNC_PROTECTS_NONBLOCKING, 15, 551, 708, 709, 711, 713, 716, 723, 725, 745, 774, 923 MPI_BAND, 186, 188, 778 MPI_BOR, 186, 188, 778 MPI_BOTTOM, 10, 15, 16, 34, 108, 123, 124, 154, 338, 340, 458, 491, 495, 710, 712, 719, 734, 739, 741, 743, 744, 746–748, 750, 761, 762, 768, 773, 930 MPI_BSEND_OVERHEAD, 52, 773 MPI_BXOR, 186, 188, 778

MPI_BYTE, <u>27</u> , 27, 28, 38, 39, 41, 146, 187,	14
$568, \overline{614}, 615, 618, 629, 768, 775, 776,$	15
927	16
MPI_C_BOOL, <u>28</u> , 187, 618, 775, 918, 924–926	17
MPI_C_COMPLEX, <u>28</u> , 187, 618, 775, 918,	18
924-926	19
MPI_C_DOUBLE_COMPLEX, <u>28</u> , 187, 618,	20
775, 924-926	20
MPI_C_FLOAT_COMPLEX, <u>28</u> , 187, 618,	
775, 924–926	22
MPI_C_LONG_DOUBLE_COMPLEX, <u>28</u> ,	23
187, 618, 775, 924–926	24
MPI_CART, 343, 779	25
MPI_CHAR, <u>28</u> , 41, 100, 189, 618, 652, 775,	26
924 MDI CILADACTED 27 20 41 120 612 776	27
MPI_CHARACTER, <u>27</u> , 39–41, 189, 618, 776 MPI_COMBINER_CONTIGUOUS, 125, 128,	28
781	29
MPI_COMBINER_DARRAY, 125, 130, 781	30
MPI_COMBINER_DUP, 125, 128, 781	31
MPI_COMBINER_F90_COMPLEX, 125, 130,	32
781	33
MPI_COMBINER_F90_INTEGER, 125, 130,	
781	34
MPI_COMBINER_F90_REAL, 125, 130, 781	35
MPI_COMBINER_HINDEXED, 19, 125, 129,	36
781	37
MPI_COMBINER_HINDEXED_BLOCK, 125,	38
129,781,920	39
MPI_COMBINER_HINDEXED_INTEGER,	40
19,704,919	41
MPI_COMBINER_HVECTOR, 19, 125, 129,	42
781	43
MPI_COMBINER_HVECTOR_INTEGER,	44
19, 704, 919	
MPI_COMBINER_INDEXED, 125, 129, 781	45
MPI_COMBINER_INDEXED_BLOCK, 125,	46
129, 781 MRI COMBINER NAMED 195, 198, 791	47
MPI_COMBINER_NAMED, 125, 128, 781	48

1 2

9 10 11

12

1 MPI_COMBINER_RESIZED, 125, 131, 781 $\mathbf{2}$ MPI_COMBINER_STRUCT, 19, 125, 130, 781 MPI_COMBINER_STRUCT_INTEGER, 19, 3 704, 919 4 MPI_COMBINER_SUBARRAY, 125, 130, 781 5MPI_COMBINER_VECTOR, 125, 128, 781 6 MPI_COMM_DUP_FN, 19, 309, 779, 923 $\overline{7}$ MPI_COMM_NULL, 257, 273, 274, 276-278, 8 281, 282, 285, 286, 302, 324, 332, 334, 9 458, 478–480, 756, 778, 927 10 MPI_COMM_NULL_COPY_FN, 19, 309, 710, 763, 779, 923 11MPI_COMM_NULL_DELETE_FN, 19, 310, 12779 13MPI_COMM_PARENT, 324 14MPI_COMM_SELF, 21, 257, 276, 288, 306, 15324, 391, 421, 426, 429, 431, 479, 570, 16705, 777, 914, 925 17MPI_COMM_TYPE_HW_GUIDED, 281, 281, 282, 777, 913 18 MPI_COMM_TYPE_HW_UNGUIDED, 282, 19283, 284, 777, 913 20MPI_COMM_TYPE_SHARED, 281, 281, 282, 21777, 921 22 MPI_COMM_WORLD, 15, 22, 30, 255, 23257-260, 268, 269, 282-284, 288, 291, 24 300, 324, 333, 385, 386, 394, 406, 419, 421, 422, 424–427, 429, 431, 445, 446, 25452, 453, 455, 459-461, 475-479, 613, 26634, 635, 658, 667, 705, 756, 767, 777, 27914, 929 28MPI_COMPLEX, 27, 187, 617, 618, 726, 776 29MPI_COMPLEX16, 187, 619, 776 30MPI_COMPLEX32, 187, 619, 776 31MPI_COMPLEX4, 187, 619, 776 MPI_COMPLEX8, 187, 619, 776 32 MPI_CONGRUENT, 269, 298, 777 33 MPI_CONVERSION_FN_NULL, 623, 779 34MPI_COUNT, 28, 29, 187, 618, 652, 775, 776, 35919 36 MPI_COUNT_KIND, 15, 17, 29, 774 37 MPI_CXX_BOOL, 29, 187, 618, 619, 776, 918 38MPI_CXX_DOUBLE_COMPLEX, <u>29</u>, 187, 39 618, 619, 776, 918 MPI_CXX_FLOAT_COMPLEX, 29, 187, 618, 40619, 776, 918 41 MPI_CXX_LONG_DOUBLE_COMPLEX, 29, 42187, 618, 619, 776, 918 43MPI_DATATYPE_NULL, 118, 778 44MPI_DISPLACEMENT_CURRENT, 581, 45782, 930 46MPI_DIST_GRAPH, 343, 779, 925 47MPI_DISTRIBUTE_BLOCK, 105, 782 MPI_DISTRIBUTE_CYCLIC, 105, 782 48

MPI_DISTRIBUTE_DFLT_DARG, 105, 782 MPI_DISTRIBUTE_NONE, 105, 782 MPI_DOUBLE, 28, 187, 618, 652, 662, 663, 725, 775 MPI_DOUBLE_COMPLEX, 27, 187, 617, 618, 726, 776 MPI_DOUBLE_INT, 190, 191, 777 MPI_DOUBLE_PRECISION, 27, 187, 618, 726, 776 MPI_DUP_FN, 19, 309, 698, 780 MPI_ERR_ACCESS, 404, 573, 636, 772 MPI_ERR_AMODE, 404, 571, 636, 772 MPI_ERR_ARG, 403, 771 MPI_ERR_ASSERT, 403, 535, 772 MPI_ERR_BAD_FILE, 404, 636, 772 MPI_ERR_BASE, 389, 403, 535, 772 MPI_ERR_BUFFER, 403, 771 MPI_ERR_COMM, 403, 771 MPI_ERR_CONVERSION, 404, 623, 636, 772 MPI_ERR_COUNT, 403, 771 MPI_ERR_DIMS, 403, 771 MPI_ERR_DISP, 403, 535, 772 MPI_ERR_DUP_DATAREP, 404, 621, 636, 772 MPI_ERR_FILE, 404, 636, 772 MPI_ERR_FILE_EXISTS, 404, 636, 772 MPI_ERR_FILE_IN_USE, 404, 573, 636, 772 MPI_ERR_GROUP, 403, 771 MPI_ERR_IN_STATUS, 33, 35, 60, 67, 69, 393, 401, 403, 561, 586, 772 MPI_ERR_INFO, 403, 772 MPI_ERR_INFO_KEY, 403, 413, 772 MPI_ERR_INFO_NOKEY, 403, 413, 772 MPI_ERR_INFO_VALUE, 403, 413, 772 MPI_ERR_INTERN, 392, 403, 771 MPI_ERR_IO, 404, 636, 772 MPI_ERR_KEYVAL, 320, 403, 772 MPI_ERR_LASTCODE, 402, 404-407, 693, 773 MPI_ERR_LOCKTYPE, 403, 535, 772 MPI_ERR_NAME, 403, 472, 772 MPI_ERR_NO_MEM, 388, 403, 772 MPI_ERR_NO_SPACE, 404, 636, 772 MPI_ERR_NO_SUCH_FILE, 404, 573, 636, 772 MPI_ERR_NOT_SAME, 404, 636, 772 MPI_ERR_OP, 403, 534, 771 MPI_ERR_OTHER, 402, 403, 771 MPI_ERR_PENDING, 67, 403, 771 MPI_ERR_PORT, 403, 469, 772 MPI_ERR_PROC_ABORTED, 403, 445, 772, 915 MPI_ERR_QUOTA, 404, 636, 772 MPI_ERR_RANK, 403, 534, 771

Unofficial Draft for Comment Only

MPI_ERR_READ_ONLY, 404, 636, 772
MPI_ERR_REQUEST, 403, 771
MPI_ERR_RMA_ATTACH, 404, 535, 772
MPI_ERR_RMA_CONFLICT, 403, 535, 772
MPI_ERR_RMA_FLAVOR, 404, 490, 535, 772
MPI_ERR_RMA_RANGE, 404, 535, 772
MPI_ERR_RMA_SHARED, 404, 535, 772
MPI_ERR_RMA_SHARED, 404, 535, 772
MPI_ERR_ROOT, 403, 771
MPI_ERR_KCO1, 403, 771 MPI_ERR_SERVICE, 403, 471, 772
MPI_ERR_SERVICE, 403, 471, 772 MPI_ERR_SESSION, 404
MPI_ERR_SIZE, 403, 535, 772
MPI_ERR_SPAWN, 403, 457, 458, 772
MPI_ERR_TAG, 403, 771
MPI_ERR_TOPOLOGY, 403, 771
MPI_ERR_TRUNCATE, 403, 771
MPI_ERR_TYPE, 403, 771
MPI_ERR_UNKNOWN, 402, 403, 771
MPI_ERR_UNSUPPORTED_DATAREP, 404,
636,772
MPI_ERR_UNSUPPORTED_OPERATION,
404,636,772
MPI_ERR_WIN, 403, 535, 772
MPI_ERRCODES_IGNORE, 16, 458, 734, 782
MPI_ERRHANDLER_NULL, 400, 778
MPI_ERROR, <u>32</u> , 32, 60, 208, 511, 774, 915,
922
MPI_ERRORS_ABORT, 391, 421, 422, 462,
914
MPI_ERRORS_ARE_FATAL, 391, 392, 408,
462, 534, 634, 774, 914
MPI_ERRORS_RETURN, 391, 392, 409, 445,
MPI_ERRORS_RETURN, 391, 392, 409, 445, 462, 635, 767, 774
462, 635, 767, 774
462, 635, 767, 774 MPI_F08_STATUS_IGNORE, 759, 783, 922
462, 635, 767, 774 MPI_F08_STATUS_IGNORE, 759, 783, 922 MPI_F08_STATUSES_IGNORE, 759, 783, 922
462, 635, 767, 774 MPI_F08_STATUS_IGNORE, 759, 783, 922 MPI_F08_STATUSES_IGNORE, 759, 783, 922 MPI_F_STATUS_IGNORE, 758, 783
462, 635, 767, 774 MPI_F08_STATUS_IGNORE, 759, 783, 922 MPI_F08_STATUSES_IGNORE, 759, 783, 922 MPI_F_STATUS_IGNORE, 758, 783 MPI_F_STATUSES_IGNORE, 758, 783
462, 635, 767, 774 MPI_F08_STATUS_IGNORE, 759, 783, 922 MPI_F08_STATUSES_IGNORE, 759, 783, 922 MPI_F_STATUS_IGNORE, 758, 783 MPI_F_STATUSES_IGNORE, 758, 783 MPI_FILE_NULL, 572, 635, 778
462, 635, 767, 774 MPI_F08_STATUS_IGNORE, 759, 783, 922 MPI_F08_STATUSES_IGNORE, 759, 783, 922 MPI_F_STATUS_IGNORE, 758, 783 MPI_F_STATUSES_IGNORE, 758, 783 MPI_FILE_NULL, 572, 635, 778 MPI_FLOAT, <u>28</u> , 100, 185, 187, 616, 618, 775
462, 635, 767, 774 MPI_F08_STATUS_IGNORE, 759, 783, 922 MPI_F08_STATUSES_IGNORE, 759, 783, 922 MPI_F_STATUS_IGNORE, 758, 783 MPI_F_STATUSES_IGNORE, 758, 783 MPI_FILE_NULL, 572, 635, 778 MPI_FLOAT, <u>28</u> , 100, 185, 187, 616, 618, 775 MPI_FLOAT_INT, 12, 190, 191, 777
462, 635, 767, 774 MPI_F08_STATUS_IGNORE, 759, 783, 922 MPI_F08_STATUSES_IGNORE, 759, 783, 922 MPI_F_STATUS_IGNORE, 758, 783 MPI_F_STATUSES_IGNORE, 758, 783 MPI_FILE_NULL, 572, 635, 778 MPI_FLOAT, <u>28</u> , 100, 185, 187, 616, 618, 775 MPI_FLOAT_INT, 12, 190, 191, 777 MPI_GRAPH, 343 , 779
462, 635, 767, 774 MPI_F08_STATUS_IGNORE, 759, 783, 922 MPI_F08_STATUSES_IGNORE, 759, 783, 922 MPI_F_STATUS_IGNORE, 758, 783 MPI_F_STATUSES_IGNORE, 758, 783 MPI_FILE_NULL, 572, 635, 778 MPI_FLOAT, <u>28</u> , 100, 185, 187, 616, 618, 775 MPI_FLOAT_INT, 12, 190, 191, 777 MPI_GRAPH, 343, 779 MPI_GROUP_EMPTY, 256 , 262, 263, 273,
462, 635, 767, 774 MPI_F08_STATUS_IGNORE, 759, 783, 922 MPI_F08_STATUSES_IGNORE, 759, 783, 922 MPI_F_STATUS_IGNORE, 758, 783 MPI_F_STATUSES_IGNORE, 758, 783 MPI_FILE_NULL, 572, 635, 778 MPI_FLOAT, <u>28</u> , 100, 185, 187, 616, 618, 775 MPI_FLOAT_INT, 12, 190, 191, 777 MPI_GRAPH, 343, 779 MPI_GROUP_EMPTY, 256, 262, 263, 273, 274, 276, 285, 302, 778
462, 635, 767, 774 MPI_F08_STATUS_IGNORE, 759, 783, 922 MPI_F08_STATUSES_IGNORE, 759, 783, 922 MPI_F_STATUS_IGNORE, 758, 783 MPI_F_STATUSES_IGNORE, 758, 783 MPI_FILE_NULL, 572, 635, 778 MPI_FLOAT, <u>28</u> , 100, 185, 187, 616, 618, 775 MPI_FLOAT_INT, 12, 190, 191, 777 MPI_GRAPH, 343, 779 MPI_GROUP_EMPTY, 256, 262, 263, 273, 274, 276, 285, 302, 778 MPI_GROUP_NULL, 256, 266, 267, 778
462, 635, 767, 774 MPI_F08_STATUS_IGNORE, 759, 783, 922 MPI_F08_STATUSES_IGNORE, 759, 783, 922 MPI_F_STATUS_IGNORE, 758, 783 MPI_F_STATUSES_IGNORE, 758, 783 MPI_FILE_NULL, 572, 635, 778 MPI_FLOAT, <u>28</u> , 100, 185, 187, 616, 618, 775 MPI_FLOAT_INT, 12, 190, 191, 777 MPI_GRAPH, 343, 779 MPI_GROUP_EMPTY, 256, 262, 263, 273, 274, 276, 285, 302, 778 MPI_GROUP_NULL, 256, 266, 267, 778 MPI_HOST, 385, 777
462, 635, 767, 774 MPI_F08_STATUS_IGNORE, 759, 783, 922 MPI_F08_STATUSES_IGNORE, 759, 783, 922 MPI_F_STATUS_IGNORE, 758, 783 MPI_F_STATUSES_IGNORE, 758, 783 MPI_FILE_NULL, 572, 635, 778 MPI_FLOAT, <u>28</u> , 100, 185, 187, 616, 618, 775 MPI_FLOAT_INT, 12, 190, 191, 777 MPI_GRAPH, 343, 779 MPI_GROUP_EMPTY, 256, 262, 263, 273, 274, 276, 285, 302, 778 MPI_GROUP_NULL, 256, 266, 267, 778 MPI_HOST, 385, 777 MPI_IDENT, 260, 269, 777
462, 635, 767, 774 MPI_F08_STATUS_IGNORE, 759, 783, 922 MPI_F08_STATUSES_IGNORE, 759, 783, 922 MPI_F_STATUS_IGNORE, 758, 783 MPI_F_STATUSES_IGNORE, 758, 783 MPI_FILE_NULL, 572, 635, 778 MPI_FLOAT, <u>28</u> , 100, 185, 187, 616, 618, 775 MPI_FLOAT_INT, 12, 190, 191, 777 MPI_GRAPH, 343, 779 MPI_GROUP_EMPTY, 256, 262, 263, 273, 274, 276, 285, 302, 778 MPI_GROUP_NULL, 256, 266, 267, 778 MPI_HOST, 385, 777 MPI_IDENT, 260, 269, 777 MPI_IN_PLACE, 16, 154, 181, 713, 734, 773
462, 635, 767, 774 MPI_F08_STATUS_IGNORE, 759, 783, 922 MPI_F08_STATUSES_IGNORE, 759, 783, 922 MPI_F_STATUS_IGNORE, 758, 783 MPI_F_STATUSES_IGNORE, 758, 783 MPI_FILE_NULL, 572, 635, 778 MPI_FLOAT, <u>28</u> , 100, 185, 187, 616, 618, 775 MPI_FLOAT_INT, 12, 190, 191, 777 MPI_GRAPH, 343, 779 MPI_GROUP_EMPTY, 256, 262, 263, 273, 274, 276, 285, 302, 778 MPI_GROUP_NULL, 256, 266, 267, 778 MPI_HOST, 385, 777 MPI_IDENT, 260, 269, 777 MPI_IN_PLACE, 16, 154, 181, 713, 734, 773 MPI_INFO, 227
462, 635, 767, 774 MPI_F08_STATUS_IGNORE, 759, 783, 922 MPI_F08_STATUSES_IGNORE, 759, 783, 922 MPI_F_STATUS_IGNORE, 758, 783 MPI_F_STATUSES_IGNORE, 758, 783 MPI_FILE_NULL, 572, 635, 778 MPI_FLOAT, <u>28</u> , 100, 185, 187, 616, 618, 775 MPI_FLOAT_INT, 12, 190, 191, 777 MPI_GRAPH, 343, 779 MPI_GROUP_EMPTY, 256, 262, 263, 273, 274, 276, 285, 302, 778 MPI_GROUP_NULL, 256, 266, 267, 778 MPI_HOST, 385, 777 MPI_IDENT, 260, 269, 777 MPI_IDENT, 260, 269, 777 MPI_INFO, 227 MPI_INFO_ENV, 418, 422, 423, 431, 777, 921
462, 635, 767, 774 MPI_F08_STATUS_IGNORE, 759, 783, 922 MPI_F08_STATUSES_IGNORE, 759, 783, 922 MPI_F_STATUS_IGNORE, 758, 783 MPI_F_STATUSES_IGNORE, 758, 783 MPI_FILE_NULL, 572, 635, 778 MPI_FLOAT, <u>28</u> , 100, 185, 187, 616, 618, 775 MPI_FLOAT_INT, 12, 190, 191, 777 MPI_GRAPH, 343, 779 MPI_GROUP_EMPTY, 256, 262, 263, 273, 274, 276, 285, 302, 778 MPI_GROUP_NULL, 256, 266, 267, 778 MPI_HOST, 385, 777 MPI_IDENT, 260, 269, 777 MPI_IDENT, 260, 269, 777 MPI_IN_PLACE, 16, 154, 181, 713, 734, 773 MPI_INFO, 227 MPI_INFO_ENV, 418, 422, 423, 431, 777, 921 MPI_INFO_NULL, 341, 388, 417, 457, 467,
462, 635, 767, 774 MPI_F08_STATUS_IGNORE, 759, 783, 922 MPI_F08_STATUSES_IGNORE, 759, 783, 922 MPI_F_STATUS_IGNORE, 758, 783 MPI_F_STATUSES_IGNORE, 758, 783 MPI_FILE_NULL, 572, 635, 778 MPI_FLOAT, $\underline{28}$, 100, 185, 187, 616, 618, 775 MPI_FLOAT_INT, 12, 190, 191, 777 MPI_GRAPH, 343, 779 MPI_GROUP_EMPTY, 256, 262, 263, 273, 274, 276, 285, 302, 778 MPI_GROUP_NULL, 256, 266, 267, 778 MPI_HOST, 385, 777 MPI_IDENT, 260, 269, 777 MPI_IDENT, 260, 269, 777 MPI_INFO, 227 MPI_INFO, 227 MPI_INFO_ENV, 418, 422, 423, 431, 777, 921 MPI_INFO_NULL, 341, 388, 417, 457, 467, 571, 573, 582, 778
$\begin{array}{c} 462,635,767,774\\ \\ MPI_F08_STATUS_IGNORE,759,783,922\\ \\ MPI_F08_STATUSES_IGNORE,759,783,922\\ \\ MPI_F_STATUS_IGNORE,758,783\\ \\ MPI_F_STATUSES_IGNORE,758,783\\ \\ MPI_FILE_NULL,572,635,778\\ \\ MPI_FLOAT,\underline{28},100,185,187,616,618,775\\ \\ MPI_FLOAT_INT,12,190,191,777\\ \\ MPI_GRAPH,343,779\\ \\ MPI_GROUP_EMPTY,256,262,263,273,$
462, 635, 767, 774 MPI_F08_STATUS_IGNORE, 759, 783, 922 MPI_F08_STATUSES_IGNORE, 759, 783, 922 MPI_F_STATUS_IGNORE, 758, 783 MPI_F_STATUSES_IGNORE, 758, 783 MPI_FILE_NULL, 572, 635, 778 MPI_FLOAT, $\underline{28}$, 100, 185, 187, 616, 618, 775 MPI_FLOAT_INT, 12, 190, 191, 777 MPI_GRAPH, 343, 779 MPI_GROUP_EMPTY, 256, 262, 263, 273, 274, 276, 285, 302, 778 MPI_GROUP_NULL, 256, 266, 267, 778 MPI_HOST, 385, 777 MPI_IDENT, 260, 269, 777 MPI_IDENT, 260, 269, 777 MPI_INFO, 227 MPI_INFO, 227 MPI_INFO_ENV, 418, 422, 423, 431, 777, 921 MPI_INFO_NULL, 341, 388, 417, 457, 467, 571, 573, 582, 778

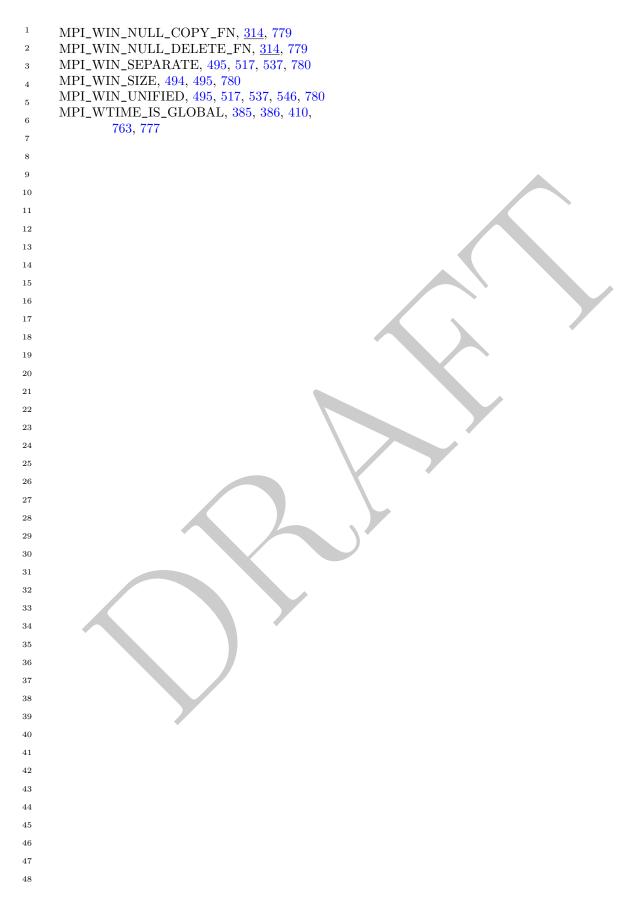
MPI_INT32_T, <u>28</u> , 187, 618, 652, 775, 924, 926	1
MPI_INT64_T, <u>28</u> , 187, 618, 652, 775, 924, 926	2
MPI_INT8_T, <u>28</u> , 187, 618, 775, 924, 926	3
MPI_INTEGER, <u>27</u> , 38, 187, 618, 725, 726,	4
769,776	5
MPI_INTEGER1, <u>27</u> , 187, 619, 776	6
MPI_INTEGER16, 187, 619, 776	7
MPI_INTEGER2, 27, 187, 617, 619, 776	
MPI_INTEGER4, <u>27</u> , 187, 619, 776	8
MPI_INTEGER8, 187, 619, 730, 776	9
MPI_INTEGER_KIND, 15, 774	10
MPI_IO, 385, 386, 777	11
MPI_KEYVAL_INVALID, 310–312, 773	12
MPI_LAND, 186, 188, 778	13
MPI_LASTUSEDCODE, 405, 406, 780	14
MPI_LB, 19, 704, 919	15
MPI_LOCK_EXCLUSIVE, 527, 773	16
MPI_LOCK_SHARED, 527, 528, 773	
MPI_LOGICAL, <u>27</u> , 187, 618, 776	17
MPI_LONG, <u>28</u> , 187, 618, 775	18
MPI_LONG_DOUBLE, <u>28</u> , 187, 618, 775	19
MPI_LONG_DOUBLE_INT, 190, 777	20
MPI_LONG_INT, 190, 191, 777	21
MPI_LONG_LONG, <u>28</u> , 187, 775, 926	22
MPI_LONG_LONG_INT, <u>28</u> , 187, 618, 775,	23
926	
MPI_LOR, 186, 188, 778	24
MPI_LXOR, 186, 188, 778	25
MPI_MAX, 185–187, 206, 778	26
MPI_MAX_DATAREP_STRING, 15, 583,	27
621, 774	28
MPI_MAX_ERROR_STRING, 15, 401, 407,	29
	30
MPI_MAX_FROM_GROUP_STRINGTAG,	31
MPI_MAX_FROM_GROUP_TAG, 285, 774	32
MPI_MAX_INFO_KEY, 15, 403, 411, 414,	33
415, 774	34
MPI_MAX_INFO_VAL, 15, 403, 411, 774	35
MPI_MAX_LIBRARY_VERSION_STRING,	36
15, 384, 774, 919 MDL MAX, OD IECT, NAME, 15, 202, 206	37
MPI_MAX_OBJECT_NAME, 15, 323–326,	38
774, 921, 927 MDI MAX DOPT NAME 15 466 774	39
MPI_MAX_PORT_NAME, 15, 466, 774	
MPI_MAX_PROCESSOR_NAME, 15, 387,	40
774, 928 MPI_MAX_PSET_NAME_LEN, 437, 774	41
	42
MPI_MAXLOC, 186, 189, 190, 193, 778 MPI_MESSACE_NO_PROC_77_77_70_80	43
MPI_MESSAGE_NO_PROC, <u>77</u> , 77, 79, 80, 88, 773, 920	44
MPI_MESSAGE_NULL, <u>77</u> , 79, 80, 778, 920	45
MPI_MESSAGE_NOLL, <u>11</u> , 19, 80, 118, 920 MPI_MESSAGE_PROC_NULL, <u>77</u>	46
MPI_MESSAGE_FROC_NOLL, <u>11</u> MPI_MIN, 186, 187, 778	47
MPI_MIN, 180, 187, 778 MPI_MINLOC, 186, 189, 190, 193, 778	
$10111_10100, 100, 109, 190, 190, 190, 100$	48

1 MPI_MODE_APPEND, 570, 571, 781 $\mathbf{2}$ MPI_MODE_CREATE, 570, 571, 579, 781 MPI_MODE_DELETE_ON_CLOSE, 570-572, 3 781 4 MPI_MODE_EXCL, 570, 571, 781 5MPI_MODE_NOCHECK, 528, 533, 534, 781 6 MPI_MODE_NOPRECEDE, 522, 533, 534, 7 781 8 MPI_MODE_NOPUT, 533, 781 9 MPI_MODE_NOSTORE, 533, 781 10 MPI_MODE_NOSUCCEED, 533, 534, 781 MPI_MODE_RDONLY, 570, 571, 576, 781 11 MPI_MODE_RDWR, 570, 571, 781 12MPI_MODE_SEQUENTIAL, 570, 571, 574, 13581, 586, 591, 604, 627, 781, 930 14MPI_MODE_UNIQUE_OPEN, 570, 571, 781 15MPI_MODE_WRONLY, 570, 571, 781 16MPI_NO_OP, 484, 508, 510, 778, 916 17MPI_NULL_COPY_FN, 19, 309, 698, 780 MPI_NULL_DELETE_FN, 19, 310, 698, 780 18 MPI_OFFSET, 28, 29, 187, 618, 775, 776, 924, 19926 20MPI_OFFSET_KIND, 15, 17, 29, 629, 734, 774 21MPI_OP_NULL, 196, 778 22MPI_ORDER_C, 15, 102, 105, 106, 782 23MPI_ORDER_FORTRAN, 15, 102, 105, 782 24 MPI_PACKED, 12, 27, 28, 38, 140, 142, 146, 617, 618, 768, 775, 776 25MPI_PROC_NULL, 26, 30-32, 74, 77, 79, 80, 2688, 88, 156, 158, 160, 162, 170, 172, 27186, 259, 352, 356, 385, 386, 489, 490, 28 498, 773, 920, 927, 929 29MPI_PROD, 186, 188, 778 30MPI_REAL, 27, 38, 187, 617, 618, 725, 726, 31 732, 776 MPI_REAL16, 187, 619, 776 32MPI_REAL2, 27, 187, 619, 776 33 MPI_REAL4, 27, 187, 619, 725, 730, 776 34MPI_REAL8, 27, 187, 619, 725, 776, 924 35 MPI_REPLACE, 506-508, 510, 550, 778, 926, 36 929 37 MPI_REQUEST_NULL, 60-63, 65-69, 560, 38 77839 MPI_ROOT, 156, 773 MPI_SEEK_CUR, 598, 605, 782 40MPI_SEEK_END, 598, 605, 782 41 MPI_SEEK_SET, 598, 599, 605, 782 42MPI_SESSION_NULL, 434, 778 43MPI_SHORT, 28, 187, 618, 775 44MPI_SHORT_INT, 190, 777 45MPI_SIGNED_CHAR, 28, 187, 189, 618, 775, 46926 47MPI_SIMILAR, 260, 269, 298, 777 MPI_SOURCE, <u>32</u>, 32, 208, 774, 915, 922 48

MPI_STATUS_IGNORE, 10, 15, 34, 34, 35, 559, 586, 712, 734, 758-760, 768, 782, 783, 920 MPI_STATUS_SIZE, 15, <u>32</u>, 32, 714, 774, 922 MPI_STATUSES_IGNORE, 14, 15, 34, 34, 35, 559, 561, 734, 758, 760, 782, 783 MPI_SUBARRAYS_SUPPORTED, 15, 708, 709, 712-716, 720-723, 735-737, 774, 922 MPI_SUBVERSION, 15, 384, 783 MPI_SUCCESS, 18, 19, 60, 67, 69, 309, 310, 312-315, 317, 318, 333, 401-403, 407-409, 422, 458, 623, 648, 657, 666, 668, 670, 675, 679, 680, 684, 691, 693, 695, 698, 771, 914, 915 MPI_SUM, 186, 188, 506, 767, 778 MPI_T_ERR_NOT_ACCESSIBLE, 676, 677 MPI_T_BIND_MPI_COMM, 649, 784 MPI_T_BIND_MPI_DATATYPE, 649, 784 MPI_T_BIND_MPI_ERRHANDLER, 649, 784MPI_T_BIND_MPI_FILE, 649, 784 MPI_T_BIND_MPI_GROUP, 649, 784 MPI_T_BIND_MPI_INFO, 649, 784 MPI_T_BIND_MPI_MESSAGE, 649, 784 MPI_T_BIND_MPI_OP, 649, 784 MPI_T_BIND_MPI_REQUEST, 649, 784 MPI_T_BIND_MPI_SESSION, 649 MPI_T_BIND_MPI_WIN, 649, 784 MPI_T_BIND_NO_OBJECT, 649, 656, 658, 665, 667, 679, 681, 784 MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE, 676, 785 MPI_T_CB_REQUIRE_MPI_RESTRICTED, 676, 785 MPI_T_CB_REQUIRE_NONE, 676, 785 MPI_T_CB_REQUIRE_THREAD_SAFE, 676, 785 MPI_T_CVAR_HANDLE_NULL, 659, 783 MPI_T_ENUM_NULL, 655, 665, 679, 783 MPI_T_ERR_CANNOT_INIT, 695, 773 MPI_T_ERR_CVAR_SET_NEVER, 660, 695, 773MPI_T_ERR_CVAR_SET_NOT_NOW, 660, 695.773 MPI_T_ERR_INVALID, 695, 773, 918 MPI_T_ERR_INVALID_HANDLE, 667, 684, 695, 773 MPI_T_ERR_INVALID_INDEX, 19, 675, 679, 695, 701, 773, 914 MPI_T_ERR_INVALID_ITEM, 19, 695, 701, 773, 914 MPI_T_ERR_INVALID_NAME, 657, 666, 680, 691, 695, 773, 918

MPI_T_ERR_INVALID_SESSION, 695, 773	MPI_
MPI_T_ERR_MEMORY, 695, 773	
MPI_T_ERR_NOT_ACCESSIBLE, 695, 773	MPI_
MPI_T_ERR_NOT_INITIALIZED, 695, 773	MPI_
MPI_T_ERR_NOT_SUPPORTED, 675, 695,	MPI_
773	MPI_
MPI_T_ERR_OUT_OF_HANDLES, 695, 773	-
MPI_T_ERR_OUT_OF_SESSIONS, 695, 773	MPI_
MPI_T_ERR_PVAR_NO_ATOMIC, 670, 695,	MPI_
773	MPI_
MPI_T_ERR_PVAR_NO_STARTSTOP, 668,	MPI_
695, 773	MPI_
MPI_T_ERR_PVAR_NO_WRITE, 669, 670,	WII I_
	MDI
695, 773	MPI_
MPI_T_PVAR_ALL_HANDLES, 668–671, 784	MPI_
MPI_T_PVAR_CLASS_AGGREGATE, 662,	MPI_
663, 784	MPI_
MPI_T_PVAR_CLASS_COUNTER, 662, 784	MPI_
MPI_T_PVAR_CLASS_GENERIC, 663, 784	MPI_
MPI_T_PVAR_CLASS_HIGHWATERMARK,	
662, 784	MPI_
MPI_T_PVAR_CLASS_LEVEL, 661, 784	
MPI_T_PVAR_CLASS_LOWWATERMARK,	MPI_
662, 784	MPI_
MPI_T_PVAR_CLASS_PERCENTAGE, 662,	
784	
MPI_T_PVAR_CLASS_SIZE, 662, 784	MPI_
MPI_T_PVAR_CLASS_STATE, 661, 784	MPI_
MPI_T_PVAR_CLASS_TIMER, 663, 784	MPI_
MPI_T_PVAR_HANDLE_NULL, 668, 783	
MPI_T_PVAR_SESSION_NULL, 666, 783	MPI_
MPI_T_SCOPE_ALL, 656, 784	
MPI_T_SCOPE_ALL_EQ, 656, 660, 784	MPI_
MPI_T_SCOPE_CONSTANT, 656, 784	
MPI_T_SCOPE_GROUP, 656, 784	MPI_
MPI_T_SCOPE_GROUP_EQ, 656, 660, 784	
MPI_T_SCOPE_LOCAL, 656, 784	MPI_
MPI_T_SCOPE_READONLY, 656, 784	MPI_
MPI_T_SOURCE_ORDERED, 675, 785	WII 1_
MPI_T_SOURCE_UNORDERED, 675, 785	MPI_
MPI_T_VERBOSITY_MPIDEV_ALL, 648,	MPI_
	_
649, 783	MPI_
MPI_T_VERBOSITY_MPIDEV_BASIC, 649,	MPI_
783	MDI
MPI_T_VERBOSITY_MPIDEV_DETAIL,	MPI_
649, 783	MPI_
MPI_T_VERBOSITY_TUNER_ALL, 649, 783	MPI_
MPI_T_VERBOSITY_TUNER_BASIC, 649,	MPI_
783	MPI_
MPI_T_VERBOSITY_TUNER_DETAIL, 649,	MPI_
783	MPI_
MPI_T_VERBOSITY_USER_ALL, 649, 783	
	MPI_
MPI_T_VERBOSITY_USER_BASIC, 648,	MPI_ MPI_
MPI_T_VERBOSITY_USER_BASIC, 648, 649, 783	

API_T_VERBOSITY_USER_DETAIL, 648,	1
649,783	2
MPI_TAG, <u>32</u> , 32, 208, 774, 915, 922	3
MPI_TAG_UB, 29, 385, 763, 767, 777	4
API_THREAD_FUNNELED, 424, 781	5
API_THREAD_MULTIPLE, 424, 426, 781,	6
917 ADI TUDEAD CEDIALIZED 494 781	7
ЛРІ_THREAD_SERIALIZED, 424, 781 ЛРІ_THREAD_SINGLE, 424, 425, 433, 781	8
API_TYPE_DUP_FN, <u>317</u> , 779	9
API_TYPE_NULL_COPY_FN, <u>317</u> , 779	10
$MPI_TYPE_NULL_DELETE_FN, 317, 779,$	11
922	12
API_TYPECLASS_COMPLEX, 731, 782	13
API_TYPECLASS_INTEGER, 731, 782	
MPI_TYPECLASS_REAL, 731, 782	14
MPI_UB, 4, 19, 704, 919	15
MPI_UINT16_T, <u>28</u> , 187, 618, 775, 924, 926	16
API_UINT32_T, <u>28</u> , 187, 618, 652, 775, 924,	17
926	18
API_UINT64_T, <u>28</u> , 187, 618, 652, 775, 924,	19
926 ADI UINTE T 28 187 618 775 024 026	20
API_UINT8_T, <u>28</u> , 187, 618, 775, 924, 926 API_UNDEFINED, <u>33</u> , <u>34</u> , 65–67, 69, 70, 112,	21
115, 117, 122, 143, 258, 259, 277, 278,	22
343, 354, 355, 727, 773, 920, 926	23
API_UNEQUAL, 260, 269, 298, 777	24
API_UNIVERSE_SIZE, 453, 475, 476, 780	25
MPI_UNSIGNED, <u>28</u> , 187, 618, 652, 661–663,	26
775	27
API_UNSIGNED_CHAR, <u>28</u> , 187, 189, 618,	28
775	29
MPI_UNSIGNED_LONG, <u>28</u> , 187, 618, 652,	30
662, 663, 775	
API_UNSIGNED_LONG_LONG, <u>28</u> , 187,	31
618, 652, 662, 663, 775, 926	32
API_UNSIGNED_SHORT, <u>28</u> , 187, 618, 775	33
API_UNWEIGHTED, 16, 338, 340–342, 349, 351, 734, 782, 919, 925	34
API_VAL, 12, 756	35
MPI_VERSION, 15, 384, 783	36
MPI_WCHAR, <u>28</u> , 189, 326, 617, 618, 775, 926	37
MPI_WEIGHTS_EMPTY, 16, 338, 340, 734,	38
782, 919	39
API_WIN_BASE, 494, 495, 767, 780	40
MPI_WIN_CREATE_FLAVOR, 494, 495, 780	41
API_WIN_DISP_UNIT, 494, 495, 780	42
API_WIN_DUP_FN, <u>314</u> , 779	43
API_WIN_FLAVOR_ALLOCATE, 495, 780	44
API_WIN_FLAVOR_CREATE, 495, 780	45
API_WIN_FLAVOR_DYNAMIC, 495, 780	46
API_WIN_FLAVOR_SHARED, 490, 495, 780	40
ЛРІ_WIN_MODEL, 495, 517, 780 ЛРІ_WIN_NULL, 494, 778	
$\mathbf{T} \mathbf{T} \mathbf{T} \mathbf{V} \mathbf{T} \mathbf{V} \mathbf{T} \mathbf{V} \mathbf{T} \mathbf{V} \mathbf{T} \mathbf{T} \mathbf{V} \mathbf{T} \mathbf{T} \mathbf{T} \mathbf{T} \mathbf{T} \mathbf{T} \mathbf{T} T$	48



MPI Declarations Index

This index refers to declarations needed in C, such as address kind integers, handles, etc. The underlined page numbers is the "main" reference (sometimes there are more than one when key concepts are discussed in multiple areas).

MPI_Aint, <u>16</u>, 16, 17, 28, <u>91</u>, 91, 94, 96, 99, 109-111, 114-116, 125, 146-148, 483, 485, 487, 491, 499, 501, 505, 507, 509-511, 513-515, 616, 621, 675, 734, 764.785 MPI_Comm, 12, 26, 260, 267, 268, 270-272, 277, 280, 286-288, 298-302, 308, 311, 312, 436, 777, 778, 785, 786 MPI_Count, 17, 17, 28, 673, 675, 785, 919 MPI_Datatype, 91, 743, 775–778, 785, 786 MPI_Errhandler, 393, 394-400, 757, 774, 778, 785.786 MPI_F08_status, 759, 783, 785, 922 MPI_File, 397, 398, 569, 572-575, 577, 578, 580, 582, 586-591, 593-605, 607-612, 616, 625–627, 757, 778, 785, 786 MPI_Fint, 756, 756, 783, 785, 926 MPI_Group, 258, 259, 259-264, 266, 299, 436, 495, 523, 524, 575, 757, 778, 785, 786 MPI_Info, 387, 411, 411-417, 435, 438, 454, 457, 459, 466, 469, 471, 479, 496, 497, 569, 572, 577, 578, 580, 700, 757, 777, 778, 785, 786, 929 MPI_Message, <u>76</u>, 757, 773, 778, 785, 786, 920 MPI_Offset, 17, 17, 28, 573-575, 580, 582, 586-590, 598, 599, 604, 605, 607, 608, 621, 629, 629, 675, 755, 785 MPI_Op, 184, 193, 196, 198-202, 204-206, 221-226, 239-244, 505, 507, 509, 514, 515, 757, 778, 785, 786 MPI_Request, 54–58, 60, 61, 62, 65–68, 70, 72, 80-86, 401, 558, 561, 588-590, 595-597, 601, 602, 736, 757, 778, 785, 786 MPI_Session, 399, 757, 778, 785, 786 MPI_Status, 30, 32-35, 37, 60, 61, 65-68, 70, 72-74, 76-78, 81, 121, 559, 564, 565, 586-588, 591, 593, 594, 600, 603, 607, 609-612, 711, 758-760, 782, 785, 786,

915, 919, 922

 $\begin{array}{l} {\rm MPI_T_cvar_handle, \underline{658}, 658, 659, 783} \\ {\rm MPI_T_enum, \underline{652}, 652-654, 664, 678, 783} \\ {\rm MPI_T_event_instance, \underline{683}, 683} \\ {\rm MPI_T_event_registration, \underline{680}, 680} \\ {\rm MPI_T_pvar_handle, \underline{666}, 666-670, 783} \\ {\rm MPI_T_pvar_session, \underline{666}, 666-670, 783} \\ {\rm MPI_Win, 314-316, 395, 396, \underline{483}, \underline{485}, \underline{487}, \\ \underline{491}, 494-497, 499, 501, 505, 507, \\ 509-511, 513-515, 520, 523-525, \\ 527-532, 757, 778, 785, 786 \\ \end{array}$

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

> 5 6

8

9

10

11 12

13

14

15

16

17

18

19

MPI Callback Function Prototype Index

11This index lists the C typedef names for callback routines, such as those used with attribute caching 12or user-defined reduction operations. Fortran example prototypes are given near the text of the C 13 name. 14MPI_Comm_copy_attr_function, 18, 19, 308, 15710, 779, 786 16MPI_Comm_delete_attr_function, 19, 308, 17779, 786 18 MPI_Comm_errhandler_fn, 700, 925 19MPI_Comm_errhandler_function, 19, 393, 700, 20704, 787, 925 21MPI_Copy_function, 19, 697, 780, 792 22 MPI_Datarep_conversion_function, 621, 779, 787 23

- MDI Deterror extent function 691 797
- ²⁴ MPI_Datarep_extent_function, 621, 787
- ²⁵ MPI_Delete_function, 19, 698, 780, 792
- ²⁶ MPI_File_errhandler_fn, 700, 925
- ²⁷ MPI_File_errhandler_function, 397, 700, 787, 925
- ²⁸ MPI_Grequest_cancel_function, 560, 787
- ²⁹ MPI_Grequest_free_function, 559, 787
- ³⁰ MPI_Grequest_query_function, 559, 787
- ³¹ MPI_Handler_function, 19, 704, 919
- 32 MPI_Session_errhandler_function, 399, 787
- 33 MPI_T_event_cb_function, 682, 787
- MPI_T_event_dropped_function, 685, 787
- MPI_T_event_free_cb_function, 684, 787
- ³⁵ MPI_Type_copy_attr_function, 317, 779, 786
- ³⁶ MPI_Type_delete_attr_function, 318, 779,
 ³⁷ 786, 922
- ³⁸ MPI_User_function, 194, 197, 786
- ³⁹ MPI_Win_copy_attr_function, **314**, **779**, **786**
- 40 MPI_Win_delete_attr_function, 314, 779, 786
- 41 MPI_Win_errhandler_fn, 700, 925
- ¹¹ MPI_Win_errhandler_function, 395, 700, 787, 925
- 43

5 6

- 44
- 46
- 47
- 48

MPI Function Index

The underlined page numbers refer to the function definitions.

MPI_ABORT, 195, 391, 421, 427, 444, 445, 478, 651, 756, 929 MPI_ACCUMULATE, 481, 498, 505, 506, 508, 515, 518, 543, 549, 550, 926, 929 MPI_ADD_ERROR_CLASS, 405, 405 MPI_ADD_ERROR_CODE, 406 MPI_ADD_ERROR_STRING, 406, 407 MPI_ADDRESS, 19, 703, 719, 919 MPI_AINT_ADD, 20, 108, <u>110</u>, 110, 491, 917 MPI_AINT_DIFF, 20, 108, 110, 110, 111, 491, 917 MPI_ALLGATHER, 151, 155, 156, 175, 176, 178, 179, 216 MPI_ALLGATHER_INIT, 234 MPI_ALLGATHERV, 151, 155, 156, 177, 177, 178, 217 MPI_ALLGATHERV_INIT, 235 MPI_ALLOC_MEM, 387, 388-390, 403, 485-490, 493, 500, 530, 720-722, 734, 916, 923 MPI_Alloc_mem, 914 MPI_ALLOC_MEM_CPTR, 388, 916 MPI_ALLREDUCE, 151, 154-156, 186, 194, 198, 198, 223, 927 MPI_ALLREDUCE_INIT, 240 MPI_ALLTOALL, 151, 155, 156, 178, 179, 181, 182, 218, 924 MPI_ALLTOALL_INIT, 236 MPI_ALLTOALLV, 151, 155, 156, 180, 181-183, 220, 924 MPI_ALLTOALLV_INIT, 237 MPI_ALLTOALLW, 151, 155, 156, <u>182</u>, 183, 184, 221, 924 MPI_ALLTOALLW_INIT, 238 MPI_ATTR_DELETE, 19, 320, 698, 699 MPI_ATTR_GET, 19, 320, 699, 763, 764 MPI_ATTR_PUT, 19, 320, 699, 763, 764, 767 MPI_BARRIER, 151, 155, <u>157</u>, 157, 210, 444, 540-542, 631 MPI_BARRIER_INIT, 228 MPI_BCAST, 151, 155, 158, 158, 159, 185, 211, 248

11 MPI_BCAST_INIT, 228 MPI_BSEND, 43, 52 12MPI_BSEND_INIT, 83, 86 13 MPI_BUFFER_ATTACH, 49, 61 14MPI_BUFFER_DETACH, 50, 922 15MPI_CANCEL, 19, 46, 61, 73, 80, 80-82, 209, 16227, 428, 511, 557, 560, 561, 700, 914 17 MPI_CART_COORDS, 331, 346, 347, 928 18 MPI_CART_CREATE, 297, 330, 331, 332, 332-334, 345, 353, 354, 356, 735, 916, 19 927 20MPI_CART_GET, 331, 345, 345, 928 21MPI_CART_MAP, 331, 354, 354, 921 22MPI_CART_RANK, 331, 346, 346, 928 23MPI_CART_SHIFT, 331, 351, 351, 352, 356, 24 928 25MPI_CART_SUB, 331, 353, 353, 354, 928 26MPI_CARTDIM_GET, 331, 344, 345, 928 MPI_CLOSE_PORT, 467, 467, 470 27MPI_COMM_ACCEPT, 465, 466, 467, 468, 28 469, 476, 477 29MPI_COMM_C2F, 756 30 MPI_COMM_CALL_ERRHANDLER, 407, 3140932 MPI_COMM_COMPARE, 269, 298, 432 33 MPI_COMM_CONNECT, 403, 468, 469, 476, 34 477MPI_COMM_CREATE, 267, 269, 273, 35273-278, 331, 925 36 MPI_COMM_CREATE_ERRHANDLER, 19, 37 392, 393, 394, 703, 789, 791, 923 38 MPI_COMM_CREATE_FROM_GROUP, 269, 39 285, 285, 286, 302, 385 40 MPI_COMM_CREATE_GROUP, 269, 275, 41 276-278, 285, 432, 920 MPI_COMM_CREATE_KEYVAL, 19, 306, 42308, 309, 310, 320, 697, 763, 788, 790, 43 923, 927 44MPI_COMM_DELETE_ATTR, 19, 306, 310, 45311, 313, 320, 699 46MPI_COMM_DISCONNECT, 320, 434, 444, 47458, 459, 477, 478, 478, 479 48

1 2 3

1	MPI_COMM_DUP, 260, 267, 269, <u>270</u> ,
2	270-272, 274, 300, 303, 307, 309, 313,
3	320, 328, 697, 705, 914, 920
4	MPI_COMM_DUP_FN, 19, <u>309</u> , 309, 310,
	715, 779, 917, 923
5	MPI_COMM_DUP_WITH_INFO, 269, 271,
6	271, 272, 286, 705, 920
7	MPI_COMM_F2C, 756
8	MPI_COMM_FREE, 267, 270, <u>286</u> , 300, 303,
9	310, 311, 313, 320, 427, 431, 434, 444,
10	458, 477, 479, 698
11	MPI_COMM_FREE_KEYVAL, 19, 306, <u>311</u> ,
	320, 698
12	MPI_COMM_GET_ATTR, 19, 306, 311, <u>312</u> ,
13	320, 385, 699, 716, 764, 767
14	MPI_COMM_GET_ERRHANDLER, 19, 392,
15	
16	<u>394,</u> 703, 928
	MPI_COMM_GET_INFO, 286, <u>288</u> , 288, 289,
17	914, 920
18	MPI_COMM_GET_NAME, 323, <u>324</u> , 324,
19	325, 927
20	MPI_COMM_GET_PARENT, 324, 421, 422,
21	$427, 455, \underline{458}, 458$
22	MPI_COMM_GROUP, 14, 258, <u>260</u> , 260, 261,
	267, 268, 298, 392, 928
23	MPI_COMM_IDUP, 267, 269, <u>271</u> , 272, 296,
24	307, 309, 313, 320, 705, 914, 916, 920
25	MPI_COMM_IDUP_WITH_INFO, 269, <u>272</u> ,
26	272, 286, 705, 914
27	MPI_COMM_JOIN, <u>479</u> , 479, 480
28	MPI_COMM_NULL_COPY_FN, 19 , <u>309</u> , 309,
29	310, 710, 763, 779, 917, 923
	MPI_COMM_NULL_DELETE_FN, 19, <u>310</u> ,
30	310, 779, 917
31	MPI_COMM_RANK, <u>268</u> , 268, 298, 717
32	MPI_COMM_RANK_F08, 717
33	MPI_COMM_REMOTE_GROUP, <u>299</u>
34	MPI_COMM_REMOTE_SIZE, <u>299</u> , 299
35	MPI_COMM_SET_ATTR, 19, 306, 310, <u>311</u> ,
	320,698,716,764,767
36	MPI_COMM_SET_ERRHANDLER, 19, 392,
37	394,703
38	MPI_COMM_SET_INFO, 228, 287, 287, 288,
39	914, 916, 920
40	MPI_COMM_SET_NAME, <u>323</u> , 323
41	MPI_COMM_SIZE, <u>267</u> , <u>268</u> , <u>298</u>
42	MPI_COMM_SPAWN, 421, 422, 446, 447, 452,
	$453, \underline{454}, 454-463, 475, 477, 915$
43	MPI_COMM_SPAWN_MULTIPLE, 421, 422,
44	447, 452, 453, 458, <u>459</u> , 460, 461, 477,
45	915
46	MPI_COMM_SPLIT, 269, 273, 274, <u>277</u> ,
47	277–279, 328, 331, 332, 334, 353–355,
48	925
-10	040

MPI_COMM_SPLIT_TYPE, 280, 284, 287, 913, 921 MPI_COMM_TEST_INTER, 297, 298 MPI_COMPARE_AND_SWAP, 481, 498, 510, 549 MPI_CONVERSION_FN_NULL, 623, 779, 917 MPI_CWIN_GET_ATTR, 716 MPI_DIMS_CREATE, 331, 333, 333, 334, 914 MPI_DIST_GRAPH_CREATE, 287, 330, 331, 336, 339, 340-342, 351, 356, 925 MPI_DIST_GRAPH_CREATE_ADJACENT, 287, 330, 331, 336, <u>337</u>, 338, 342, 351, 356, 921, 925 MPI_DIST_GRAPH_NEIGHBORS, 331, 349, 350, 351, 356, 921, 925 MPI_DIST_GRAPH_NEIGHBORS_COUNT, 331, <u>349</u>, 349, 351, 918, 925 MPI_DUP_FN, 19, 309, 698, 780 MPI_ERRHANDLER_C2F, 445, 757 MPI_ERRHANDLER_CREATE, 19, 703, 919, 923 MPI_ERRHANDLER_F2C, 445, 757 MPI_ERRHANDLER_FREE, 392, 400, 427, 434, 445, 928 MPI_ERRHANDLER_GET, 19, 703, 919, 928 MPI_ERRHANDLER_SET, 19, 703, 919 MPI_ERROR_CLASS, 402, 402, 405, 445, 693 MPI_ERROR_STRING, 401, 401, 402, 405, 407, 445 MPI_EVENT_CALLBACK_GET_INFO, 683 MPI_EVENT_HANDLE_GET_INFO, 681 MPI_EXSCAN, 152, 155, 186, 194, 205, 206, 226, 925 MPI_EXSCAN_INIT, 244 MPI_F_SYNC_REG, 109, 551, 708, 724, 724, 725, 744, 746, 747, 749, 923 MPI_FETCH_AND_OP, 481, 498, 506, 508, 509, 509 MPI_FILE_C2F, 757 MPI_FILE_CALL_ERRHANDLER, 408, 409 MPI_FILE_CLOSE, 479, 569, 570, 572, 572 MPI_FILE_CREATE_ERRHANDLER, 392, 397, 398, 789, 792, 923 MPI_FILE_DELETE, 571, 572, 572, 573, 577, 579,635 MPI_FILE_F2C, 757 MPI_FILE_GET_AMODE, 576, 576 MPI_FILE_GET_ATOMICITY, 626, 627 MPI_FILE_GET_BYTE_OFFSET, 591, 599, 599.605 MPI_FILE_GET_ERRHANDLER, 392, 398,

Unofficial Draft for Comment Only

635, 928

MPI_FILE_GET_GROUP, 575, 575

MPI_FILE_GET_INFO, 577, 578, 578, 579, 914, 929 MPI_FILE_GET_POSITION, 598, 599 MPI_FILE_GET_POSITION_SHARED, 604, 605, 605, 627 MPI_FILE_GET_SIZE, 575, 575, 630 MPI_FILE_GET_TYPE_EXTENT, 615, 616, 623MPI_FILE_GET_VIEW, 582, 583 MPI_FILE_IREAD, 583, 595, 595, 606, 624, 625MPI_FILE_IREAD_ALL, 583, 596, 596, 917 MPI_FILE_IREAD_AT, 583, 589, 589 MPI_FILE_IREAD_AT_ALL, 583, 589, 590, 917 MPI_FILE_IREAD_SHARED, 583, 601, 602 MPI_FILE_IWRITE, 583, 597, 597 MPI_FILE_IWRITE_ALL, 583, 597, 598, 917 MPI_FILE_IWRITE_AT, 583, 590, 590 MPI_FILE_IWRITE_AT_ALL, 583, 591, 591, 917 MPI_FILE_IWRITE_SHARED, 583, 602, 602 MPI_FILE_OPEN, 404, 451, 569, 569-571, 577, 579, 581, 599, 629, 630, 635, 636 MPI_FILE_PREALLOCATE, 573, 574, 574, 625, 630 MPI_FILE_READ, 583, 592, 592, 593, 595, 629, 630 MPI_FILE_READ_ALL, 583, 593, 593, 596, 606, 607 MPI_FILE_READ_ALL_BEGIN, 583, 606, 607, 609, 624, 749 MPI_FILE_READ_ALL_END, 583, 606, 607, 610, 624, 749 MPI_FILE_READ_AT, 583, 586, 586, 587, 589 MPI_FILE_READ_AT_ALL, 583, 587, 587, 590 MPI_FILE_READ_AT_ALL_BEGIN, 583, 607, 749 MPI_FILE_READ_AT_ALL_END, 583, 608, 749MPI_FILE_READ_ORDERED, 583, 603, 603 MPI_FILE_READ_ORDERED_BEGIN, 583, 611, 749 MPI_FILE_READ_ORDERED_END, 583, <u>612</u>, 749 MPI_FILE_READ_SHARED, 583, 600, 600, 602, 603 MPI_FILE_SEEK, 598, 598, 599 MPI_FILE_SEEK_SHARED, 604, 604, 605, 627 MPI_FILE_SET_ATOMICITY, 571, 625, 626, 626 MPI_FILE_SET_ERRHANDLER, 392, 397,

635 MPI_FILE_SET_INFO, 577, 577-579, 914, 929 $\mathbf{2}$ MPI_FILE_SET_SIZE, <u>573</u>, 573, 574, 625, 3 628, 630 MPI_FILE_SET_VIEW, 103, 404, 570, 5 577-579, 580, 581, 582, 599, 605, 614, 6 621, 629, 636, 929, 930 MPI_FILE_SYNC, 572, 584, 624, 625, 627, 627, 632 MPI_FILE_WRITE, 583, 584, 594, 594, 595, 9 597.629 10 MPI_FILE_WRITE_ALL, 583, 594, 595, 598 11 MPI_FILE_WRITE_ALL_BEGIN, 583, 610, 12736, 749 13 MPI_FILE_WRITE_ALL_END, 583, 611, 749 14MPI_FILE_WRITE_AT, 583, 584, 587, 588, 15590 MPI_FILE_WRITE_AT_ALL, 583, 588, 588, 1617591 MPI_FILE_WRITE_AT_ALL_BEGIN, 583, 18 608, 749 19MPI_FILE_WRITE_AT_ALL_END, 583, 609, 20749 21MPI_FILE_WRITE_ORDERED, 583, 603, 22604, 604 23MPI_FILE_WRITE_ORDERED_BEGIN, 24 583, 612, 749 MPI_FILE_WRITE_ORDERED_END, 583, 25<u>613</u>, 749 26MPI_FILE_WRITE_SHARED, 583, 584, 601, 27601 - 60428 MPI_FINALIZE, 15, 21, 22, 385, 391, 419, 426, 29427-431, 449, 478, 479, 570, 648, 658, 30 671, 673, 756, 758, 760, 915, 921, 929 31MPI_Finalize, 431 MPI_FINALIZED, 429, 430, 430, 431, 445, 32 756, 917 33 MPI_FREE_MEM, 389, 389, 403, 486, 488 34 MPI_GATHER, 151, 154, 155, 159, 162, 169, 35 170, 176, 185, 212 36 MPI_GATHER_INIT, 229 37 MPI_GATHERV, 151, 155, <u>161</u>, 162, 163, 171, 38 177, 178, 214 MPI_GATHERV_INIT, 230 39 MPI_GET, 481, 498, 501, 502, 508, 514, 518, 40 540, 542, 543, 551, 747, 929 41 MPI_GET_ACCUMULATE, 481, 498, 506, 42507, 508, 509, 516, 543, 549, 916 43 MPI_GET_ADDRESS, 19, 91, 108, 109, 44109-111, 123, 491, 703, 719, 738, 743, 45761.762 46MPI_GET_COUNT, <u>33</u>, 33, 34, 60, 122, 511, 565, 585, 920, 926 47MPI_GET_ELEMENTS, 60, 121-123, 565, 48

Unofficial Draft for Comment Only

1	566, 585, 920	ľ
2	MPI_GET_ELEMENTS_X, 60, 121–123, 565,	
3	586, 919	ľ
4	MPI_GET_LIBRARY_VERSION, <u>384</u> , <u>384</u> ,	
	445, 916, 917, 919	ľ
5	MPI_GET_PROCESSOR_NAME, <u>386</u> , 387,	ľ
6	928	
7	MPI_GET_VERSION, <u>383</u> , 384, 445, 723, 917	ľ
8	MPI_GRAPH_CREATE, 330, 331, <u>334</u> , 334,	-
9	336, 341, 344, 348, 355, 356, 927, 928	ľ
10	MPI_GRAPH_GET, 331, 344, 344	ľ
	MPI_GRAPH_MAP, 331, <u>355</u> , 355	ľ
11	MPI_GRAPH_NEIGHBORS, 331, <u>347</u> , 348,	l
12	356, 925	l
13		
14	MPI_GRAPH_NEIGHBORS_COUNT, 331,	1
15	<u>347</u> , 348, 925	I
16	MPI_GRAPHDIMS_GET, 331, <u>343</u> , 344	
	MPI_GREQUEST_COMPLETE, 558–560,	ľ
17	$\frac{561}{561}$, 561	_
18	MPI_GREQUEST_START, <u>558</u> , 558, 789, 792,	ľ
19	926	_
20	MPI_GROUP_C2F, <u>757</u>	ľ
21	MPI_GROUP_COMPARE, <u>259</u> , 263	
22	MPI_GROUP_DIFFERENCE, <u>262</u>	l
	MPI_GROUP_EXCL, <u>263</u> , 263, 265	
23	MPI_GROUP_F2C, <u>757</u>	1
24	MPI_GROUP_FREE, <u>266</u> , 267, 268, 392, 427,	
25	434, 928	l
26	MPI_GROUP_FROM_SESSION_PSET, <u>266</u> ,	1
27	266	
28	MPI_GROUP_INCL, <u>262</u> , 263, 264	
	MPI_GROUP_INTERSECTION, <u>261</u>	
29	MPI_GROUP_RANGE_EXCL, <u>265</u> , 265	
30	MPI_GROUP_RANGE_INCL, <u>264</u> , 264	1
31	MPI_GROUP_RANK, <u>258</u> , 268	
32	MPI_GROUP_SIZE, <u>258</u> , 267	
33	MPI_GROUP_TRANSLATE_RANKS, 259,	ľ
34	259, 927	
	MPI_GROUP_UNION, <u>261</u>	ľ
35	MPI_IALLGATHER, 151, 155, 156, 216	
36	MPI_IALLGATHERV, 151, 155, 156, 217	ľ
37	MPI_IALLREDUCE, 151, 155, 156, <u>222</u>	ľ
38	MPI_IALLTOALL, 151, 155, 156, <u>218</u>	-
39	MPI_IALLTOALLV, 151, 155, 156, <u>219</u>	ľ
40	MPI_IALLTOALLW, 151, 155, 156, <u>220</u>	1
	MPI_IBARRIER, 151, 155, 208, <u>210</u> , 210, 248	ľ
41	MPI_IBCAST, 151, 155, 200, 210, 210, 240 MPI_IBCAST, 151, 155, 210, 211, 252	ľ
42	MPI_IBSEND, <u>55</u> , 61, 86	
43		l
44	MPI_IEXSCAN, 152, 155, <u>226</u>	ľ
45	MPI_IGATHER, 151, 155, <u>212</u>	r
46	MPI_IGATHERV, 151, 155, <u>213</u>	I
	MPI_IMPROBE, 73, <u>76</u> , 76–78, 80, 450, 916,	
47	920	1
	NET THE CIT IS A SOLAR SOLAR	
48	MPI_IMRECV, 76–78, <u>79</u> , 80, 920	1

MPI_INEIGHBOR_ALLGATHER, 331, <u>366</u> ,
921 MPI_INEIGHBOR_ALLGATHERV, 331, <u>367</u> ,
921
MPI_INEIGHBOR_ALLTOALL, 331, <u>368</u> , 921
MPI_INEIGHBOR_ALLTOALLV, 331, <u>369</u> ,
921
MPI_INEIGHBOR_ALLTOALLW, 332, <u>370</u> ,
921
MPI_INFO_C2F, 445, <u>757</u>
MPI_INFO_CREATE, 412, 412, 445
MPI_INFO_CREATE_ENV, <u>417</u> , 445, 913
MPI_INFO_DELETE, 403, <u>413</u> , 413, 416, 445
MPI_INFO_DUP, <u>417</u> , 417, 445
MPI_INFO_F2C, 445, <u>757</u>
MPI_INFO_FREE, 289, <u>417</u> , 418, 427, 434,
$\begin{array}{c} 438, 445, 498, 578, 675, 679, 681, 684 \end{array}$
MPI_INFO_GET, 19, 411, <u>413</u> , 445, <u>700</u> , 915,
929
MPI_INFO_GET_NKEYS, 411, <u>416</u> , 416, 445,
929
MPI_INFO_GET_NTHKEY, 411, <u>416</u> , 445,
929
MPI_INFO_GET_STRING, 19, 411, <u>415</u> , 415,
700, 701, 915
MPI_INFO_GET_VALUELEN, 19, 411, <u>414</u> ,
445, <u>701</u> , 915, 929
MPI_INFO_SET, <u>412</u> , 413–416, 445
MPI_INIT, 15, 21, 22, 257, 385, 391, 418, 419,
$\underbrace{420}_{420}, 420, 421, 423, 424, 426, 429-431,$
$\underbrace{}_{444, 450, 455-458, 475, 476, 648, 650,}$
658, 671, 755, 756, 758, 760, 915, 920,
921, 924, 925
MPI_INIT_THREAD, 21, 257, 391, 418–421,
$\underbrace{423}_{424-426}, 429-431, 433, 450, 476,$
648, 650, 651, 658, 755, 920, 921, 925
MPI_INITIALIZED, 429, <u>430</u> , 430, 445, 756,
917
MPI_INTERCOMM_CREATE, 270, 276, 299,
<u>300</u> , 300, 303, 921
MPI_Intercomm_create, 302
MPI_INTERCOMM_CREATE_FROM_GROUPS,
269, 270, 299, <u>301</u> , 385
MPI_INTERCOMM_MERGE, 269, 276, 296,
299, 300, <u>302</u> , 303, 923
MPI_IPROBE, 34, <u>73</u> , 73–78, 80, 450, 920
MPI_IRECV, <u>57</u> , 80, 737, 738, 741, 742
MPI_IREDUCE, 151, 155, 156, <u>221</u> , 222
MPI_IREDUCE_SCATTER, <u>151</u> , <u>155</u> , <u>156</u> ,
<u>224</u>
MPI_IREDUCE_SCATTER_BLOCK, 151,
155, 156, 223

MPI_IRSEND, 56

 $\mathrm{MPI_IS_THREAD_MAIN},\, 424,\, \underline{426},\, 917$

MPI_ISCAN, 152, 155, 225 MPI_ISCATTER, 151, 155, 214 MPI_ISCATTERV, 151, 155, 215 MPI_ISEND, <u>54</u>, 86, 715, 716, 719, 736, 737, 742 MPI_ISENDRECV, 58, 915 MPI_ISENDRECV_REPLACE, 59, 915 $MPI_{ISSEND}, 55$ MPI_KEYVAL_CREATE, 19, 697, 699, 793 MPI_KEYVAL_FREE, 19, 320, 698 MPI_LOOKUP_NAME, 403, 465, 470, 471, 472MPI_MESSAGE_C2F, <u>757</u>, 920 MPI_MESSAGE_F2C, 757, 920 MPI_MPROBE, 73, 76, 77, <u>78</u>, 78, 80, 450, 920 MPI_MRECV, 76, 77, <u>78</u>, 78–80, 920 MPI_NEIGHBOR_ALLGATHER, 331, 357, 358, 360, 366, 921 MPI_NEIGHBOR_ALLGATHER_INIT, 371 MPI_NEIGHBOR_ALLGATHERV, 331, 359, 367, 921 MPI_NEIGHBOR_ALLGATHERV_INIT, 372 MPI_NEIGHBOR_ALLTOALL, 331, <u>360</u>, 362, 368, 921 MPI_NEIGHBOR_ALLTOALL_INIT, 373 MPI_NEIGHBOR_ALLTOALLV, 331, 362, 370.921 MPI_NEIGHBOR_ALLTOALLV_INIT, 374 MPI_NEIGHBOR_ALLTOALLW, 331, 363, 364, 371, 921 MPI_NEIGHBOR_ALLTOALLW_INIT, 376 MPI_NULL_COPY_FN, 18, 19, 309, 698, 780 MPI_NULL_DELETE_FN, 19, 310, 698, 780 MPI_OP_C2F, 757 MPI_OP_COMMUTATIVE, 200, 925 MPI_OP_CREATE, 193, 194, 196, 715, 787, 790, 922 MPI_OP_F2C, 757 MPI_OP_FREE, 196, 427, 434 MPI_OPEN_PORT, 465, 466, 468-470, 472MPI_PACK, 52, 140, 142, 143, 146, 617, 622 MPI_PACK_EXTERNAL, 8, 146, <u>147</u>, 729, 927 MPI_PACK_EXTERNAL_SIZE, 148 MPI_PACK_SIZE, 52, <u>143</u>, 143, 920 MPI_PCONTROL, 642, 643, 643, 644 MPI_PROBE, 31, 34, 35, 73, 74, 74-76, 78, 80, 450, 920 MPI_PUBLISH_NAME, 465, 466, 470, 470 - 472MPI_PUT, 481, 498, <u>499</u>, 501, 506, 512, 518, 524, 534, 536, 541, 542, 551, 736, 747, 929

MPI_QUERY_THREAD, <u>425</u> , 426, 450, 917	1
MPI_RACCUMULATE, 481, 498, 506, 508,	2
$\frac{514}{515}$	3
MPI_RECV, 26, <u>30</u> , 32, 34, 73, 76, 77, 90, 120,	4
121, 141, 152, 160, 249, 566, 631, 671,	
743, 746, 747	5
MPI_RECV_INIT, <u>85</u> , 86	6
MPI_REDUCE, 151, 155, 156, <u>184</u> , 185, 186,	7
194-196, 198, 202, 203, 205, 206, 222,	8
506, 508, 510, 926	9
MPI_REDUCE_INIT, 239	10
MPI_REDUCE_LOCAL, 185, 186, 194, 200,	11
923, 925	
MPI_REDUCE_SCATTER, 151, 155, 156,	12
186, 194, 202, <u>203</u> , 203, 224	13
MPI_REDUCE_SCATTER_BLOCK, 151,	14
	15
155, 156, 186, 194, 201, 201, 202, 224, 925	16
MPI_REDUCE_SCATTER_BLOCK_INIT,	17
241	18
MPI_REDUCE_SCATTER_INIT, <u>242</u>	
MPI_REGISTER_DATAREP, 404, <u>620</u> ,	19
621–623, 636, 790, 792	20
MPI_REQUEST_C2F, <u>757</u>	21
MPI_REQUEST_F2C, 757	22
$\operatorname{MII} \mathbf{I}_{1} \operatorname{MEQUESI}_{1} \mathbf{I}_{2} \mathbf{O}, 13 \mathbf{I}_{1}$	
	23
MPI_REQUEST_FREE, <u>62</u> , 63, 80, 87, 209,	23 24
MPI_REQUEST_FREE, <u>62</u> , 63, 80, 87, 209, 227, 427, 434, 511, 560, 561, 924	24
MPI_REQUEST_FREE, <u>62</u> , 63, 80, 87, 209, 227, 427, 434, 511, 560, 561, 924 MPI_REQUEST_GET_STATUS, 35, <u>72</u> , 72,	24 25
$\begin{array}{c} \text{MPI}_{\text{REQUEST}_{\text{FREE}}, \ \underline{62}, \ 63, \ 80, \ 87, \ 209, \\ & 227, \ 427, \ 434, \ 511, \ 560, \ 561, \ 924 \\ \text{MPI}_{\text{REQUEST}_{\text{GET}_{\text{STATUS}}, \ 35, \ \underline{72}, \ 72, \\ & 559, \ 924 \end{array}$	24
$\begin{array}{c} \text{MPI}_{\text{REQUEST}_{\text{FREE}}, \underline{62}, 63, 80, 87, 209, \\ & 227, 427, 434, 511, 560, 561, 924 \\ \text{MPI}_{\text{REQUEST}_{\text{GET}_{\text{STATUS}}, 35, \underline{72}, 72, \\ & 559, 924 \\ \text{MPI}_{\text{RGET}, 481, 498, \underline{513}, 514 \end{array}$	24 25
$\begin{array}{c} \text{MPI}_{\text{REQUEST}_{\text{FREE}}, \underline{62}, 63, 80, 87, 209, \\ & 227, 427, 434, 511, 560, 561, 924 \\ \text{MPI}_{\text{REQUEST}_{\text{GET}}, \text{STATUS}, 35, \underline{72}, 72, \\ & 559, 924 \\ \text{MPI}_{\text{RGET}, 481, 498, \underline{513}, 514 \\ \text{MPI}_{\text{RGET}_{\text{ACCUMULATE}, 481, 498, 506, } \end{array}$	24 25 26
$\begin{array}{c} \text{MPI_REQUEST_FREE, } \underline{62}, 63, 80, 87, 209, \\ 227, 427, 434, 511, 560, 561, 924 \\ \text{MPI_REQUEST_GET_STATUS, } 35, \underline{72}, 72, \\ 559, 924 \\ \text{MPI_RGET, } 481, 498, \underline{513}, 514 \\ \text{MPI_RGET_ACCUMULATE, } 481, 498, 506, \\ 508, \underline{515}, 516 \end{array}$	24 25 26 27
$\begin{array}{c} \text{MPI_REQUEST_FREE, \ \underline{62}, \ 63, \ 80, \ 87, \ 209, \\ & 227, \ 427, \ 434, \ 511, \ 560, \ 561, \ 924 \\ \\ \text{MPI_REQUEST_GET_STATUS, \ 35, \ \underline{72}, \ 72, \\ & 559, \ 924 \\ \\ \text{MPI_RGET, \ 481, \ 498, \ \underline{513}, \ 514 \\ \\ \text{MPI_RGET_ACCUMULATE, \ 481, \ 498, \ 506, \\ & 508, \ \underline{515}, \ 516 \\ \\ \text{MPI_RPUT, \ 481, \ 498, \ \underline{512}, \ 512, \ 513 \\ \end{array}$	24 25 26 27 28 28
$\begin{array}{c} \text{MPI}_{\text{REQUEST}_{\text{FREE}}, \underline{62}, 63, 80, 87, 209, \\ & 227, 427, 434, 511, 560, 561, 924 \\ \text{MPI}_{\text{REQUEST}_{\text{GET}}, \text{STATUS}, 35, \underline{72}, 72, \\ & 559, 924 \\ \text{MPI}_{\text{RGET}, 481, 498, \underline{513}, 514 \\ \text{MPI}_{\text{RGET}}, \text{ACCUMULATE}, 481, 498, 506, \\ & 508, \underline{515}, 516 \\ \text{MPI}_{\text{RPUT}, 481, 498, \underline{512}, 512, 513 \\ \text{MPI}_{\text{RSEND}, \underline{44}} \end{array}$	24 25 26 27 28 29 30
$\begin{array}{l} \text{MPI}_{\text{REQUEST}_{\text{FREE}}, \underline{62}, 63, 80, 87, 209, \\ 227, 427, 434, 511, 560, 561, 924 \\ \text{MPI}_{\text{REQUEST}_{\text{GET}}, \text{STATUS}, 35, \underline{72}, 72, \\ 559, 924 \\ \text{MPI}_{\text{RGET}, 481, 498, \underline{513}, 514 \\ \text{MPI}_{\text{RGET}}, \text{ACCUMULATE}, 481, 498, 506, \\ 508, \underline{515}, 516 \\ \text{MPI}_{\text{RPUT}, 481, 498, \underline{512}, 512, 513 \\ \text{MPI}_{\text{RSEND}, \underline{44}} \\ \text{MPI}_{\text{RSEND}_{\text{INIT}}, \underline{85} \\ \end{array}$	24 25 26 27 28 29 30 31
$\begin{array}{l} \text{MPI}_{\text{REQUEST}_{\text{FREE}}, \underline{62}, 63, 80, 87, 209, \\ 227, 427, 434, 511, 560, 561, 924 \\ \text{MPI}_{\text{REQUEST}_{\text{GET}}, \text{STATUS}, 35, \underline{72}, 72, \\ 559, 924 \\ \text{MPI}_{\text{RGET}, 481, 498, \underline{513}, 514 \\ \text{MPI}_{\text{RGET}}, \text{ACCUMULATE}, 481, 498, 506, \\ 508, \underline{515}, 516 \\ \text{MPI}_{\text{RPUT}}, 481, 498, \underline{512}, 512, 513 \\ \text{MPI}_{\text{RSEND}, \underline{44} \\ \text{MPI}_{\text{RSEND}_{\text{INIT}}, \underline{85} \\ \text{MPI}_{\text{SCAN}, 152, 155, 186, 194, \underline{204}, 205, 206, \\ \end{array}$	24 25 26 27 28 29 30 31 32
$\begin{array}{l} {\rm MPI_REQUEST_FREE,\ \underline{62},\ 63,\ 80,\ 87,\ 209, \\ 227,\ 427,\ 434,\ 511,\ 560,\ 561,\ 924 \\ {\rm MPI_REQUEST_GET_STATUS,\ 35,\ \underline{72},\ 72, \\ 559,\ 924 \\ {\rm MPI_RGET,\ 481,\ 498,\ \underline{513},\ 514 \\ {\rm MPI_RGET_ACCUMULATE,\ 481,\ 498,\ 506, \\ 508,\ \underline{515},\ 516 \\ {\rm MPI_RPUT,\ 481,\ 498,\ \underline{512},\ 512,\ 513 \\ {\rm MPI_RSEND,\ \underline{44} \\ {\rm MPI_RSEND_INIT,\ \underline{85} \\ {\rm MPI_SCAN,\ 152,\ 155,\ 186,\ 194,\ \underline{204},\ 205,\ 206, \\ 225 \\ \end{array}}$	24 25 26 27 28 29 30 31
$\begin{array}{l} \text{MPI}_{\text{REQUEST}_{\text{FREE}}, \underline{62}, 63, 80, 87, 209, \\ 227, 427, 434, 511, 560, 561, 924 \\ \text{MPI}_{\text{REQUEST}_{\text{GET}}, \text{STATUS}, 35, \underline{72}, 72, \\ 559, 924 \\ \text{MPI}_{\text{RGET}, 481, 498, \underline{513}, 514 \\ \text{MPI}_{\text{RGET}}, \text{ACCUMULATE}, 481, 498, 506, \\ 508, \underline{515}, 516 \\ \text{MPI}_{\text{RPUT}, 481, 498, \underline{512}, 512, 513 \\ \text{MPI}_{\text{RSEND}, \underline{44} \\ \text{MPI}_{\text{RSEND}_{\text{INIT}}, \underline{85} \\ \text{MPI}_{\text{SCAN}, 152, 155, 186, 194, \underline{204}, 205, 206, \\ 225 \\ \text{MPI}_{\text{SCAN}_{\text{INIT}}, \underline{243} \\ \end{array}$	24 25 26 27 28 29 30 31 32
$\begin{split} & \text{MPI}_\text{REQUEST}_\text{FREE}, \ \underline{62}, \ 63, \ 80, \ 87, \ 209, \\ & 227, \ 427, \ 434, \ 511, \ 560, \ 561, \ 924 \\ & \text{MPI}_\text{REQUEST}_\text{GET}_\text{STATUS}, \ 35, \ \underline{72}, \ 72, \\ & 559, \ 924 \\ & \text{MPI}_\text{RGET}, \ 481, \ 498, \ \underline{513}, \ 514 \\ & \text{MPI}_\text{RGET}_\text{ACCUMULATE}, \ 481, \ 498, \ 506, \\ & 508, \ \underline{515}, \ 516 \\ & \text{MPI}_\text{RPUT}, \ 481, \ 498, \ \underline{512}, \ 512, \ 513 \\ & \text{MPI}_\text{RSEND}, \ \underline{44} \\ & \text{MPI}_\text{RSEND}_\text{INIT}, \ \underline{85} \\ & \text{MPI}_\text{SCAN}, \ 152, \ 155, \ 186, \ 194, \ \underline{204}, \ 205, \ 206, \\ & 225 \\ & \text{MPI}_\text{SCATTER}, \ 151, \ 155, \ \underline{169}, \ 169, \ 171, \ 172, \\ \end{split}$	24 25 26 27 28 30 31 32 33
$\begin{split} & \text{MPI}_\text{REQUEST}_\text{FREE}, \ \underline{62}, \ 63, \ 80, \ 87, \ 209, \\ & 227, \ 427, \ 434, \ 511, \ 560, \ 561, \ 924 \\ & \text{MPI}_\text{REQUEST}_\text{GET}_\text{STATUS}, \ 35, \ \underline{72}, \ 72, \\ & 559, \ 924 \\ & \text{MPI}_\text{RGET}, \ 481, \ 498, \ \underline{513}, \ 514 \\ & \text{MPI}_\text{RGET}_\text{ACCUMULATE}, \ 481, \ 498, \ 506, \\ & 508, \ \underline{515}, \ 516 \\ & \text{MPI}_\text{RPUT}, \ 481, \ 498, \ \underline{512}, \ 512, \ 513 \\ & \text{MPI}_\text{RSEND}, \ \underline{44} \\ & \text{MPI}_\text{RSEND}_\text{INIT}, \ \underline{85} \\ & \text{MPI}_\text{SCAN}, \ 152, \ 155, \ 186, \ 194, \ \underline{204}, \ 205, \ 206, \\ & 225 \\ & \text{MPI}_\text{SCATTER}, \ 151, \ 155, \ \underline{169}, \ 169, \ 171, \ 172, \\ & 202, \ 214 \\ \end{split}$	24 25 26 27 28 30 31 32 33 34
$\begin{split} & \text{MPI}_\text{REQUEST}_\text{FREE}, \ \underline{62}, \ 63, \ 80, \ 87, \ 209, \\ & 227, \ 427, \ 434, \ 511, \ 560, \ 561, \ 924 \\ & \text{MPI}_\text{REQUEST}_\text{GET}_\text{STATUS}, \ 35, \ \underline{72}, \ 72, \\ & 559, \ 924 \\ & \text{MPI}_\text{RGET}, \ 481, \ 498, \ \underline{513}, \ 514 \\ & \text{MPI}_\text{RGET}_\text{ACCUMULATE}, \ 481, \ 498, \ 506, \\ & 508, \ \underline{515}, \ 516 \\ & \text{MPI}_\text{RPUT}, \ 481, \ 498, \ \underline{512}, \ 512, \ 513 \\ & \text{MPI}_\text{RSEND}_\text{MII}, \ \underline{85} \\ & \text{MPI}_\text{RSEND}_\text{INIT}, \ \underline{85} \\ & \text{MPI}_\text{SCAN}_\text{INIT}, \ \underline{243} \\ & \text{MPI}_\text{SCATTER}, \ 151, \ 155, \ \underline{169}, \ 169, \ 171, \ 172, \\ & \ 202, \ 214 \\ & \text{MPI}_\text{SCATTER}_\text{INIT}, \ \underline{232} \end{split}$	244 25 26 27 28 29 29 30 31 31 32 33 34 35 36
$\begin{split} & \text{MPI}_\text{REQUEST}_\text{FREE}, \ \underline{62}, \ 63, \ 80, \ 87, \ 209, \\ & 227, \ 427, \ 434, \ 511, \ 560, \ 561, \ 924 \\ & \text{MPI}_\text{REQUEST}_\text{GET}_\text{STATUS}, \ 35, \ \underline{72}, \ 72, \\ & 559, \ 924 \\ & \text{MPI}_\text{RGET}, \ 481, \ 498, \ \underline{513}, \ 514 \\ & \text{MPI}_\text{RGET}_\text{ACCUMULATE}, \ 481, \ 498, \ 506, \\ & 508, \ \underline{515}, \ 516 \\ & \text{MPI}_\text{RGET}_\text{ACCUMULATE}, \ 481, \ 498, \ 506, \\ & 508, \ \underline{515}, \ 516 \\ & \text{MPI}_\text{RPUT}, \ 481, \ 498, \ \underline{512}, \ 512, \ 513 \\ & \text{MPI}_\text{RSEND}, \ \underline{44} \\ & \text{MPI}_\text{RSEND}_\text{INIT}, \ \underline{85} \\ & \text{MPI}_\text{SCAN}, \ 152, \ 155, \ 186, \ 194, \ \underline{204}, \ 205, \ 206, \\ & 225 \\ & \text{MPI}_\text{SCATTER}, \ 151, \ 155, \ \underline{169}, \ 169, \ 171, \ 172, \\ & 202, \ 214 \\ & \text{MPI}_\text{SCATTER}_\text{INIT}, \ \underline{232} \\ & \text{MPI}_\text{SCATTERV}, \ 151, \ 155, \ \underline{171}, \ 171-173, \\ \end{split}$	24 25 26 27 28 29 30 31 31 32 33 34 35 36 37
$\begin{split} & \text{MPI}_\text{REQUEST}_\text{FREE}, \ \underline{62}, \ 63, \ 80, \ 87, \ 209, \\ & 227, \ 427, \ 434, \ 511, \ 560, \ 561, \ 924 \\ & \text{MPI}_\text{REQUEST}_\text{GET}_\text{STATUS}, \ 35, \ \underline{72}, \ 72, \\ & 559, \ 924 \\ & \text{MPI}_\text{RGET}, \ 481, \ 498, \ \underline{513}, \ 514 \\ & \text{MPI}_\text{RGET}_\text{ACCUMULATE}, \ 481, \ 498, \ 506, \\ & 508, \ \underline{515}, \ 516 \\ & \text{MPI}_\text{RGET}_\text{ACCUMULATE}, \ 481, \ 498, \ 506, \\ & 508, \ \underline{515}, \ 516 \\ & \text{MPI}_\text{RPUT}, \ 481, \ 498, \ \underline{512}, \ 512, \ 513 \\ & \text{MPI}_\text{RSEND}, \ \underline{44} \\ & \text{MPI}_\text{RSEND}_\text{INIT}, \ \underline{85} \\ & \text{MPI}_\text{SCAN}, \ 152, \ 155, \ 186, \ 194, \ \underline{204}, \ 205, \ 206, \\ & \ 225 \\ & \text{MPI}_\text{SCATTER}, \ 151, \ 155, \ \underline{169}, \ 169, \ 171, \ 172, \\ & \ 202, \ 214 \\ & \text{MPI}_\text{SCATTER}_\text{INIT}, \ \underline{232} \\ & \text{MPI}_\text{SCATTERV}, \ 151, \ 155, \ \underline{171}, \ 171-173, \\ & \ 203, \ 216 \\ \end{split}$	244 25 26 27 28 29 29 30 31 31 32 33 34 35 35 36 37 37 38
$\begin{split} & \text{MPI}_\text{REQUEST}_\text{FREE}, \ \underline{62}, \ 63, \ 80, \ 87, \ 209, \\ & 227, \ 427, \ 434, \ 511, \ 560, \ 561, \ 924 \\ & \text{MPI}_\text{REQUEST}_\text{GET}_\text{STATUS}, \ 35, \ \underline{72}, \ 72, \\ & 559, \ 924 \\ & \text{MPI}_\text{RGET}, \ 481, \ 498, \ \underline{513}, \ 514 \\ & \text{MPI}_\text{RGET}_\text{ACCUMULATE}, \ 481, \ 498, \ 506, \\ & 508, \ \underline{515}, \ 516 \\ & \text{MPI}_\text{RGET}_\text{ACCUMULATE}, \ 481, \ 498, \ 506, \\ & 508, \ \underline{515}, \ 516 \\ & \text{MPI}_\text{RDUT}, \ 481, \ 498, \ \underline{512}, \ 512, \ 513 \\ & \text{MPI}_\text{RSEND}, \ \underline{44} \\ & \text{MPI}_\text{RSEND}_\text{INIT}, \ \underline{85} \\ & \text{MPI}_\text{SCAN}, \ 152, \ 155, \ 186, \ 194, \ \underline{204}, \ 205, \ 206, \\ & \ 225 \\ & \text{MPI}_\text{SCATTER}, \ 151, \ 155, \ \underline{169}, \ 169, \ 171, \ 172, \\ & \ 202, \ 214 \\ & \text{MPI}_\text{SCATTER}_\text{INIT}, \ \underline{232} \\ & \text{MPI}_\text{SCATTERV}, \ 151, \ 155, \ \underline{171}, \ 171-173, \\ & \ 203, \ 216 \\ & \text{MPI}_\text{SCATTERV}_\text{INIT}, \ \underline{233} \\ \end{split}$	24 25 26 27 28 29 30 31 31 32 33 34 35 36 37
$\begin{split} & \text{MPI}_\text{REQUEST}_\text{FREE}, \ \underline{62}, \ 63, \ 80, \ 87, \ 209, \\ & 227, \ 427, \ 434, \ 511, \ 560, \ 561, \ 924 \\ & \text{MPI}_\text{REQUEST}_\text{GET}_\text{STATUS}, \ 35, \ \underline{72}, \ 72, \\ & 559, \ 924 \\ & \text{MPI}_\text{RGET}, \ 481, \ 498, \ \underline{513}, \ 514 \\ & \text{MPI}_\text{RGET}_\text{ACCUMULATE}, \ 481, \ 498, \ 506, \\ & 508, \ \underline{515}, \ 516 \\ & \text{MPI}_\text{RPUT}, \ 481, \ 498, \ \underline{512}, \ 512, \ 513 \\ & \text{MPI}_\text{RSEND}, \ \underline{44} \\ & \text{MPI}_\text{RSEND}_\text{INIT}, \ \underline{85} \\ & \text{MPI}_\text{SCAN}, \ 152, \ 155, \ 186, \ 194, \ \underline{204}, \ 205, \ 206, \\ & 225 \\ & \text{MPI}_\text{SCATTER}, \ 151, \ 155, \ \underline{169}, \ 169, \ 171, \ 172, \\ & 202, \ 214 \\ & \text{MPI}_\text{SCATTER}, \ 151, \ 155, \ \underline{171}, \ 171-173, \\ & 203, \ 216 \\ & \text{MPI}_\text{SCATTERV}, \ 1511, \ \underline{233} \\ & \text{MPI}_\text{SEND}, \ 25, \ \underline{26}, \ 27, \ 34, \ 39, \ 90, \ 119, \ 120, \\ \end{split}$	244 25 26 27 28 29 29 30 31 31 32 33 34 35 35 36 37 37 38
$\begin{split} & \text{MPI}_\text{REQUEST}_\text{FREE}, \ \underline{62}, \ 63, \ 80, \ 87, \ 209, \\ & 227, \ 427, \ 434, \ 511, \ 560, \ 561, \ 924 \\ & \text{MPI}_\text{REQUEST}_\text{GET}_\text{STATUS}, \ 35, \ \underline{72}, \ 72, \\ & 559, \ 924 \\ & \text{MPI}_\text{RGET}, \ 481, \ 498, \ \underline{513}, \ 514 \\ & \text{MPI}_\text{RGET}_\text{ACCUMULATE}, \ 481, \ 498, \ 506, \\ & 508, \ \underline{515}, \ 516 \\ & \text{MPI}_\text{RPUT}, \ 481, \ 498, \ \underline{512}, \ 512, \ 513 \\ & \text{MPI}_\text{RSEND}, \ \underline{44} \\ & \text{MPI}_\text{RSEND}_\text{INIT}, \ \underline{85} \\ & \text{MPI}_\text{SCAN}, \ 152, \ 155, \ 186, \ 194, \ \underline{204}, \ 205, \ 206, \\ & \ 225 \\ & \text{MPI}_\text{SCATTER}, \ 151, \ 155, \ \underline{169}, \ 169, \ 171, \ 172, \\ & \ 202, \ 214 \\ & \text{MPI}_\text{SCATTER}, \ 151, \ 155, \ \underline{171}, \ 171-173, \\ & \ 203, \ 216 \\ & \text{MPI}_\text{SCATTERV}, \ 151, \ 155, \ \underline{171}, \ 171-173, \\ & \ 203, \ 216 \\ & \text{MPI}_\text{SCATTERV}, \ 151, \ 155, \ \underline{171}, \ 171-173, \\ & \ 203, \ 216 \\ & \text{MPI}_\text{SEND}, \ 25, \ \underline{26}, \ 27, \ 34, \ 39, \ 90, \ 119, \ 120, \\ & \ 140, \ 249, \ 570, \ 631, \ 644, \ 743, \ 744, \ 746 \\ \end{split}$	24 25 26 27 28 30 31 31 32 33 34 35 36 37 38 36 37 38
$\begin{split} & \text{MPI}_\text{REQUEST}_\text{FREE}, \ \underline{62}, \ 63, \ 80, \ 87, \ 209, \\ & 227, \ 427, \ 434, \ 511, \ 560, \ 561, \ 924 \\ & \text{MPI}_\text{REQUEST}_\text{GET}_\text{STATUS}, \ 35, \ \underline{72}, \ 72, \\ & 559, \ 924 \\ & \text{MPI}_\text{RGET}, \ 481, \ 498, \ \underline{513}, \ 514 \\ & \text{MPI}_\text{RGET}_\text{ACCUMULATE}, \ 481, \ 498, \ 506, \\ & 508, \ \underline{515}, \ 516 \\ & \text{MPI}_\text{RGET}_\text{ACCUMULATE}, \ 481, \ 498, \ 506, \\ & 508, \ \underline{515}, \ 516 \\ & \text{MPI}_\text{RPUT}, \ 481, \ 498, \ \underline{512}, \ 512, \ 513 \\ & \text{MPI}_\text{RSEND}, \ \underline{44} \\ & \text{MPI}_\text{RSEND}_\text{INIT}, \ \underline{85} \\ & \text{MPI}_\text{SCAN}, \ 152, \ 155, \ 186, \ 194, \ \underline{204}, \ 205, \ 206, \\ & \ 225 \\ & \text{MPI}_\text{SCATTER}, \ 151, \ 155, \ \underline{169}, \ 169, \ 171, \ 172, \\ & \ 202, \ 214 \\ & \text{MPI}_\text{SCATTER}_\text{INIT}, \ \underline{232} \\ & \text{MPI}_\text{SCATTERV}, \ 151, \ 155, \ \underline{171}, \ 171-173, \\ & \ 203, \ 216 \\ & \text{MPI}_\text{SCATTERV}, \ 151, \ 155, \ \underline{171}, \ 171-173, \\ & \ 203, \ 216 \\ & \text{MPI}_\text{SEND}, \ 25, \ \underline{26}, \ 27, \ 34, \ 39, \ 90, \ 119, \ 120, \\ & \ 140, \ 249, \ 570, \ 631, \ 644, \ 743, \ 744, \ 746 \\ & \text{MPI}_\text{SEND}_\text{INIT}, \ \underline{83}, \ 86 \\ \end{split}$	24 25 26 27 28 30 31 31 32 33 34 35 36 37 38 36 37 38 36 37 38
$\begin{split} & \text{MPI}_\text{REQUEST}_\text{FREE}, \ \underline{62}, \ 63, \ 80, \ 87, \ 209, \\ & 227, \ 427, \ 434, \ 511, \ 560, \ 561, \ 924 \\ & \text{MPI}_\text{REQUEST}_\text{GET}_\text{STATUS}, \ 35, \ \underline{72}, \ 72, \\ & 559, \ 924 \\ & \text{MPI}_\text{RGET}, \ 481, \ 498, \ \underline{513}, \ 514 \\ & \text{MPI}_\text{RGET}_\text{ACCUMULATE}, \ 481, \ 498, \ 506, \\ & 508, \ \underline{515}, \ 516 \\ & \text{MPI}_\text{RGET}_\text{ACCUMULATE}, \ 481, \ 498, \ 506, \\ & 508, \ \underline{515}, \ 516 \\ & \text{MPI}_\text{RPUT}, \ 481, \ 498, \ \underline{512}, \ 512, \ 513 \\ & \text{MPI}_\text{RSEND}, \ \underline{44} \\ & \text{MPI}_\text{RSEND}_\text{INIT}, \ \underline{85} \\ & \text{MPI}_\text{SCAN}, \ 152, \ 155, \ 186, \ 194, \ \underline{204}, \ 205, \ 206, \\ & 225 \\ & \text{MPI}_\text{SCAN}_\text{INIT}, \ \underline{243} \\ & \text{MPI}_\text{SCATTER}, \ 151, \ 155, \ \underline{169}, \ 169, \ 171, \ 172, \\ & 202, \ 214 \\ & \text{MPI}_\text{SCATTERV}, \ 151, \ 155, \ \underline{171}, \ 171-173, \\ & 203, \ 216 \\ & \text{MPI}_\text{SCATTERV}, \ 151, \ 155, \ \underline{171}, \ 171-173, \\ & 203, \ 216 \\ & \text{MPI}_\text{SEND}, \ 25, \ \underline{26}, \ 27, \ 34, \ 39, \ 90, \ 119, \ 120, \\ & 140, \ 249, \ 570, \ 631, \ 644, \ 743, \ 744, \ 746 \\ & \text{MPI}_\text{SEND}_\text{INIT}, \ \underline{83}, \ 86 \\ & \text{MPI}_\text{SENDRECV}, \ \underline{36}, \ 351 \\ \end{split}$	24 25 20 27 28 29 30 31 31 32 33 34 35 36 37 38 39 30 37 38 39 40 41
$\begin{split} & \text{MPI}_\text{REQUEST}_\text{FREE}, \ \underline{62}, \ 63, \ 80, \ 87, \ 209, \\ & 227, \ 427, \ 434, \ 511, \ 560, \ 561, \ 924 \\ & \text{MPI}_\text{REQUEST}_\text{GET}_\text{STATUS}, \ 35, \ \underline{72}, \ 72, \\ & 559, \ 924 \\ & \text{MPI}_\text{RGET}, \ 481, \ 498, \ \underline{513}, \ 514 \\ & \text{MPI}_\text{RGET}_\text{ACCUMULATE}, \ 481, \ 498, \ 506, \\ & 508, \ \underline{515}, \ 516 \\ & \text{MPI}_\text{RGET}_\text{ACCUMULATE}, \ 481, \ 498, \ 506, \\ & 508, \ \underline{515}, \ 516 \\ & \text{MPI}_\text{RPUT}, \ 481, \ 498, \ \underline{512}, \ 512, \ 513 \\ & \text{MPI}_\text{RSEND}, \ \underline{44} \\ & \text{MPI}_\text{RSEND}_\text{INIT}, \ \underline{85} \\ & \text{MPI}_\text{SCAN}, \ 152, \ 155, \ 186, \ 194, \ \underline{204}, \ 205, \ 206, \\ & \ 225 \\ & \text{MPI}_\text{SCAN}_\text{INIT}, \ \underline{243} \\ & \text{MPI}_\text{SCATTER}, \ 151, \ 155, \ \underline{169}, \ 169, \ 171, \ 172, \\ & \ 202, \ 214 \\ & \text{MPI}_\text{SCATTER}, \ 151, \ 155, \ \underline{171}, \ 171-173, \\ & \ 203, \ 216 \\ & \text{MPI}_\text{SCATTERV}, \ 151, \ 155, \ \underline{171}, \ 171-173, \\ & \ 203, \ 216 \\ & \text{MPI}_\text{SCATTERV}_\text{INIT}, \ \underline{233} \\ & \text{MPI}_\text{SEND}, \ 25, \ \underline{26}, \ 27, \ 34, \ 39, \ 90, \ 119, \ 120, \\ & \ 140, \ 249, \ 570, \ 631, \ 644, \ 743, \ 744, \ 746 \\ & \text{MPI}_\text{SEND}_\text{INIT}, \ \underline{83}, \ 86 \\ & \text{MPI}_\text{SEND}\text{RECV}_\text{REPLACE}, \ \underline{37} \\ \end{array}$	24 25 26 27 28 30 31 31 32 33 34 35 36 37 38 36 37 38 36 40 41 41 42 43
$\begin{split} & \text{MPI}_\text{REQUEST}_\text{FREE}, \ \underline{62}, \ 63, \ 80, \ 87, \ 209, \\ & 227, \ 427, \ 434, \ 511, \ 560, \ 561, \ 924 \\ & \text{MPI}_\text{REQUEST}_\text{GET}_\text{STATUS}, \ 35, \ \underline{72}, \ 72, \\ & 559, \ 924 \\ & \text{MPI}_\text{RGET}, \ 481, \ 498, \ \underline{513}, \ 514 \\ & \text{MPI}_\text{RGET}_\text{ACCUMULATE}, \ 481, \ 498, \ 506, \\ & 508, \ \underline{515}, \ 516 \\ & \text{MPI}_\text{REPUT}, \ 481, \ 498, \ \underline{512}, \ 512, \ 513 \\ & \text{MPI}_\text{RSEND}, \ \underline{44} \\ & \text{MPI}_\text{RSEND}_\text{INIT}, \ \underline{85} \\ & \text{MPI}_\text{SCAN}, \ 152, \ 155, \ 186, \ 194, \ \underline{204}, \ 205, \ 206, \\ & \ 225 \\ & \text{MPI}_\text{SCAN}_\text{INIT}, \ \underline{243} \\ & \text{MPI}_\text{SCATTER}, \ 151, \ 155, \ \underline{169}, \ 169, \ 171, \ 172, \\ & \ 202, \ 214 \\ & \text{MPI}_\text{SCATTER}_\text{INIT}, \ \underline{232} \\ & \text{MPI}_\text{SCATTERV}, \ 151, \ 155, \ \underline{171}, \ 171-173, \\ & \ 203, \ 216 \\ & \text{MPI}_\text{SCATTERV}_\text{INIT}, \ \underline{233} \\ & \text{MPI}_\text{SEND}, \ 25, \ \underline{26}, \ 27, \ 34, \ 39, \ 90, \ 119, \ 120, \\ & \ 140, \ 249, \ 570, \ 631, \ 644, \ 743, \ 744, \ 746 \\ & \text{MPI}_\text{SENDRECV}_\text{REPLACE}, \ \underline{37} \\ & \text{MPI}_\text{SENDRECV}_\text{REPLACE}, \ \underline{37} \\ & \text{MPI}_\text{SESSION}_\text{C2F}, \ \underline{757} \\ \end{split}$	24 25 20 27 28 30 31 31 32 33 34 35 33 34 35 34 35 34 35 34 35 34 34 35 34 34 35 34 34 35 34 34 35 34 34 34 34 34 34 34 34 34 34 34 34 34
$\begin{split} & \text{MPI}_\text{REQUEST}_\text{FREE}, \ \underline{62}, \ 63, \ 80, \ 87, \ 209, \\ & 227, \ 427, \ 434, \ 511, \ 560, \ 561, \ 924 \\ & \text{MPI}_\text{REQUEST}_\text{GET}_\text{STATUS}, \ 35, \ \underline{72}, \ 72, \\ & 559, \ 924 \\ & \text{MPI}_\text{RGET}, \ 481, \ 498, \ \underline{513}, \ 514 \\ & \text{MPI}_\text{RGET}_\text{ACCUMULATE}, \ 481, \ 498, \ 506, \\ & 508, \ \underline{515}, \ 516 \\ & \text{MPI}_\text{REPUT}, \ 481, \ 498, \ \underline{512}, \ 512, \ 513 \\ & \text{MPI}_\text{RSEND}, \ \underline{44} \\ & \text{MPI}_\text{RSEND}_\text{INIT}, \ \underline{85} \\ & \text{MPI}_\text{SCAN}, \ 152, \ 155, \ 186, \ 194, \ \underline{204}, \ 205, \ 206, \\ & \ 225 \\ & \text{MPI}_\text{SCAN}_\text{INIT}, \ \underline{243} \\ & \text{MPI}_\text{SCATTER}, \ 151, \ 155, \ \underline{169}, \ 169, \ 171, \ 172, \\ & \ 202, \ 214 \\ & \text{MPI}_\text{SCATTER}_\text{INIT}, \ \underline{232} \\ & \text{MPI}_\text{SCATTERV}, \ 151, \ 155, \ \underline{171}, \ 171-173, \\ & \ 203, \ 216 \\ & \text{MPI}_\text{SCATTERV}_\text{INIT}, \ \underline{233} \\ & \text{MPI}_\text{SEND}, \ 25, \ \underline{26}, \ 27, \ 34, \ 39, \ 90, \ 119, \ 120, \\ & \ 140, \ 249, \ 570, \ 631, \ 644, \ 743, \ 744, \ 746 \\ & \text{MPI}_\text{SEND}_\text{INIT}, \ \underline{83}, \ 86 \\ & \text{MPI}_\text{SEND}\text{RECV}_\text{REPLACE}, \ \underline{37} \\ & \text{MPI}_\text{SESSION}_\text{C2F}, \ \underline{757} \\ & \text{MPI}_\text{SESSION}_\text{CALL}_\text{ERRHANDLER}, \ \underline{409}, \\ \end{split}$	244 25 26 27 28 30 31 32 33 34 35 35 36 37 38 39 30 40 41 41 42 43 44 44
$\begin{split} & \text{MPI}_\text{REQUEST}_\text{FREE}, \ \underline{62}, \ 63, \ 80, \ 87, \ 209, \\ & 227, \ 427, \ 434, \ 511, \ 560, \ 561, \ 924 \\ & \text{MPI}_\text{REQUEST}_\text{GET}_\text{STATUS}, \ 35, \ \underline{72}, \ 72, \\ & 559, \ 924 \\ & \text{MPI}_\text{RGET}, \ 481, \ 498, \ \underline{513}, \ 514 \\ & \text{MPI}_\text{RGET}_\text{ACCUMULATE}, \ 481, \ 498, \ 506, \\ & 508, \ \underline{515}, \ 516 \\ & \text{MPI}_\text{RPUT}, \ 481, \ 498, \ \underline{512}, \ 512, \ 513 \\ & \text{MPI}_\text{RSEND}, \ \underline{44} \\ & \text{MPI}_\text{RSEND}_\text{INIT}, \ \underline{85} \\ & \text{MPI}_\text{SCAN}, \ 152, \ 155, \ 186, \ 194, \ \underline{204}, \ 205, \ 206, \\ & 225 \\ & \text{MPI}_\text{SCATTER}, \ 151, \ 155, \ \underline{169}, \ 169, \ 171, \ 172, \\ & 202, \ 214 \\ & \text{MPI}_\text{SCATTER}_\text{INIT}, \ \underline{232} \\ & \text{MPI}_\text{SCATTERV}, \ 151, \ 155, \ \underline{171}, \ 171-173, \\ & 203, \ 216 \\ & \text{MPI}_\text{SCATTERV}, \ 151, \ 155, \ \underline{171}, \ 171-173, \\ & 203, \ 216 \\ & \text{MPI}_\text{SEND}, \ 25, \ \underline{26}, \ 27, \ 34, \ 39, \ 90, \ 119, \ 120, \\ & 140, \ 249, \ 570, \ 631, \ 644, \ 743, \ 744, \ 746 \\ & \text{MPI}_\text{SEND}_\text{INIT}, \ \underline{83}, \ 86 \\ & \text{MPI}_\text{SENDRECV}_\text{REPLACE}, \ \underline{37} \\ & \text{MPI}_\text{SESSION}_\text{C2F}, \ \underline{757} \\ & \text{MPI}_\text{SESSION}_\text{CALL}_\text{ERRHANDLER}, \ \underline{409}, \\ & 409, \ 445 \\ \end{array}$	24 25 20 27 28 30 31 31 32 33 34 35 33 34 35 34 35 34 35 34 35 34 34 35 34 34 35 34 34 35 34 34 35 34 34 34 34 34 34 34 34 34 34 34 34 34
$\begin{split} & \text{MPI}_\text{REQUEST}_\text{FREE}, \ \underline{62}, \ 63, \ 80, \ 87, \ 209, \\ & 227, \ 427, \ 434, \ 511, \ 560, \ 561, \ 924 \\ & \text{MPI}_\text{REQUEST}_\text{GET}_\text{STATUS}, \ 35, \ \underline{72}, \ 72, \\ & 559, \ 924 \\ & \text{MPI}_\text{RGET}, \ 481, \ 498, \ \underline{513}, \ 514 \\ & \text{MPI}_\text{RGET}_\text{ACCUMULATE}, \ 481, \ 498, \ 506, \\ & 508, \ \underline{515}, \ 516 \\ & \text{MPI}_\text{REPUT}, \ 481, \ 498, \ \underline{512}, \ 512, \ 513 \\ & \text{MPI}_\text{RSEND}, \ \underline{44} \\ & \text{MPI}_\text{RSEND}_\text{INIT}, \ \underline{85} \\ & \text{MPI}_\text{SCAN}, \ 152, \ 155, \ 186, \ 194, \ \underline{204}, \ 205, \ 206, \\ & \ 225 \\ & \text{MPI}_\text{SCAN}_\text{INIT}, \ \underline{243} \\ & \text{MPI}_\text{SCATTER}, \ 151, \ 155, \ \underline{169}, \ 169, \ 171, \ 172, \\ & \ 202, \ 214 \\ & \text{MPI}_\text{SCATTER}_\text{INIT}, \ \underline{232} \\ & \text{MPI}_\text{SCATTERV}, \ 151, \ 155, \ \underline{171}, \ 171-173, \\ & \ 203, \ 216 \\ & \text{MPI}_\text{SCATTERV}_\text{INIT}, \ \underline{233} \\ & \text{MPI}_\text{SEND}, \ 25, \ \underline{26}, \ 27, \ 34, \ 39, \ 90, \ 119, \ 120, \\ & \ 140, \ 249, \ 570, \ 631, \ 644, \ 743, \ 744, \ 746 \\ & \text{MPI}_\text{SEND}_\text{INIT}, \ \underline{83}, \ 86 \\ & \text{MPI}_\text{SEND}\text{RECV}_\text{REPLACE}, \ \underline{37} \\ & \text{MPI}_\text{SESSION}_\text{C2F}, \ \underline{757} \\ & \text{MPI}_\text{SESSION}_\text{CALL}_\text{ERRHANDLER}, \ \underline{409}, \\ \end{split}$	244 25 26 27 28 30 31 32 33 34 35 35 36 37 38 39 30 40 41 41 42 43 44 44

1	MPI_SESSION_CREATE_KEYVAL, 791
2	MPI_SESSION_F2C, 757
3	MPI_SESSION_FINALIZE, <u>434</u> , 434, 444
4	MPI_SESSION_GET_ERRHANDLER, 392,
5	400
	MPI_SESSION_GET_INFO, 433, 437, 438
6	MPI_SESSION_GET_NTH_PSET, 266, 436,
7	437
8	MPI_SESSION_GET_NUM_PSETS, <u>435</u> , 435,
9	436
10	$MPI_SESSION_GET_PSET_INFO, \underline{438}$
11	MPI_SESSION_INIT, 392, 427, <u>433</u> , 433, 444,
12	450, 476, 651
13	MPI_SESSION_SET_ERRHANDLER, 392,
14	399
15	MPI_SIZEOF, 19, 701, <u>702</u> , 913
	MPI_SSEND, <u>44</u>
16	MPI_SSEND_INIT, <u>84</u>
17	MPI_START, <u>86</u> , 86, 87, 227, 743
18	MPI_STARTALL, <u>87</u> , 87, 227, 743
19	MPI_STATUS_C2F, 758
20	MPI_STATUS_C2F08, <u>759</u> , 922
21	MPI_STATUS_F082C, <u>759</u> , 922
22	MPI_STATUS_F082F, <u>760</u> , 922
23	MPI_STATUS_F2C, 758
24	MPI_STATUS_F2F08, <u>760</u> , 922 MDI_STATUS_SET_CANCELLED_F6F_015
	MPI_STATUS_SET_CANCELLED, <u>565</u> , 915 MPI_STATUS_SET_ELEMENTS, <u>564</u> , 565
25	MPI_STATUS_SET_ELEMENTS_X, <u>565</u> ,
26	565, 919
27	MPI_T_CATEGORY_CHANGED, <u>693</u>
28	MPI_T_CATEGORY_GET_CATEGORIES,
29	692, 692, 693, 695
30	MPI_T_CATEGORY_GET_CVARS, 691, 691,
31	<u>693, 695</u>
32	MPI_T_CATEGORY_GET_EVENTS, <u>692</u> ,
33	692
34	MPI_T_CATEGORY_GET_INDEX, <u>691</u> , 691,
35	695, 918
	MPI_T_CATEGORY_GET_INFO, <u>690</u> , 690,
36	693,695,917,918
37	MPI_T_CATEGORY_GET_NUM, <u>689</u>
38	MPI_T_CATEGORY_GET_NUM_EVENTS,
39	$\underline{690}, 691$
40	MPI_T_CATEGORY_GET_PVARS, <u>692</u> , 692,
41	693, 695
42	MPI_T_CVAR_GET_INDEX, <u>656</u> , 657, 695,
43	918
44	MPI_T_CVAR_GET_INFO, 652, 654, <u>655</u> ,
45	655, 656, 658–660, 693, 695, 916, 918
	MPI_T_CVAR_GET_NUM, <u>654</u> , 659
46	MPI_T_CVAR_HANDLE_ALLOC, 652, 658,
47	659,660,695
48	

MPI_T_CVAR_HANDLE_FREE, <u>659</u> , 659,
695
MPI_T_CVAR_READ, <u>659</u> , 695
MPI_T_CVAR_WRITE, <u>659</u> , 695
MPI_T_ENUM_GET_INFO, <u>652</u> , <u>653</u> , <u>695</u>
MPI_T_ENUM_GET_ITEM, 653, 653, 695
MPI_T_EVENT_CALLBACK_GET_INFO,
683
MPI_T_EVENT_CALLBACK_SET_INFO,
683, 683
MPI_T_EVENT_COPY, 677, 679, 687, 687
MPI_T_EVENT_GET_INDEX, <u>680</u> , 680, 695
MPI_T_EVENT_GET_INFO, 652, <u>678</u> , 678,
681, 687, 695
MPI_T_EVENT_GET_NUM, <u>677</u>
MPI_T_EVENT_GET_SOURCE, 677, 688
MPI_T_EVENT_GET_TIMESTAMP, 675,
$677, \underline{687}, 687$
MPI_T_EVENT_HANDLE_ALLOC, <u>680</u> , 680,
683, 695
MPI_T_EVENT_HANDLE_FREE, <u>684</u> , <u>684</u> ,
695
MPI_T_EVENT_HANDLE_GET_INFO, <u>681</u> ,
681
MPI_T_EVENT_HANDLE_SET_INFO, 681,
681
MPI_T_EVENT_READ, 677, 679, <u>686</u> , 686
MPI_T_EVENT_REGISTER_CALLBACK,
$\frac{682}{682}, 682$
MPI_T_EVENT_SET_DROPPED_HANDLER,
<u>685, 685, 686</u>
MPI_T_FINALIZE, <u>651</u> , 651
MPI_T_INIT_THREAD, <u>650</u> , 650, 651
MPI_T_PVAR_GET_INDEX, <u>665</u> , 665, 695,
918
MPI_T_PVAR_GET_INFO, 652, 663, <u>664</u> ,
664,667,669,670,693,695,916,918
MPI_T_PVAR_GET_NUM, <u>663</u> , 667
MPI_T_PVAR_HANDLE_ALLOC, 652, 667,
667, 669, 695
MPI_T_PVAR_HANDLE_FREE, <u>667</u> , 668,
695, 917
MPI_T_PVAR_READ, <u>669</u> , 669, 670, 677,
695, 917
MPI_T_PVAR_READRESET, 665, <u>670</u> , 670,
677, 695, 917 MDI T DVAD DESET 670, 670, 677, 605
MPI_T_PVAR_RESET, <u>670</u> , 670, 677, 695,
917
MPI_T_PVAR_SESSION_CREATE, <u>666</u> , 695
MPI_T_PVAR_SESSION_FREE, <u>666</u> , 695
MPI_T_PVAR_START, <u>668</u> , 677, 695, 917
MPI_T_PVAR_STOP, <u>668</u> , 677, 695, 917
MPI_T_PVAR_WRITE, <u>669</u> , 669, 677, 695,

MPI_T_SOURCE_GET_INFO, <u>674</u> , 675, 688, 695
MPI_T_SOURCE_GET_NUM, <u>674</u> , 674, 695
$MPI_T_SOURCE_GET_NOM, 674, 674, 695$ $MPI_T_SOURCE_GET_TIMESTAMP, 675,$
675, 677, 695
MPI_TEST, 35, 60, <u>61</u> , 62–67, 72, 81, 87, 227, 427, 561, 584, 585
MPI_TEST_CANCELLED, 60–62, <u>81</u> , 82, 559, 566, 586
MPI_TESTALL, 65, <u>68</u> , 68, 432, 450, 559–561,
$\begin{array}{c} \text{MI1} \underline{1} \underline{1} \underline{1} \underline{2} \underline{3} \underline{1} \underline{1} \underline{1} \underline{1} \underline{1} \underline{2} \underline{3} \underline{1} \underline{1} \underline{1} \underline{1} \underline{1} \underline{1} \underline{1} 1$
MPI_TESTANY, 65, <u>66</u> , 66, 67, 70, 450, 559,
560, 564
MPI_TESTSOME, 65, <u>70</u> , 70, 71, 450,
559–561, 564
MPI_TOPO_TEST, 331, <u>343</u> , 343
MPI_TYPE_C2F, <u>756</u>
MPI_TYPE_COMMIT, 117, 118, 757
MPI_TYPE_CONTIGUOUS, 12, <u>91</u> , 91, 93,
112, 125, 568, 616
MPI_TYPE_CREATE_DARRAY, 12, 34, <u>104</u> ,
104, 105, 125
MPI_TYPE_CREATE_F90_COMPLEX, 12,
125, 127, 187, 617, 708, <u>727</u> , 729
MPI_TYPE_CREATE_F90_INTEGER, 12,
125, 127, 187, 617, 708, <u>727</u> , 729
MPI_TYPE_CREATE_F90_REAL, 12, 125,
127, 187, 617, 708, <u>726</u> , 727–729, 924
MPI_TYPE_CREATE_HINDEXED, 12, 19,
91, 96, 96, 99, 100, 125, 703
MPI_TYPE_CREATE_HINDEXED_BLOCK,
12, 91, <u>98</u> , 98, 125, 920
MPI_TYPE_CREATE_HVECTOR, 12, 19,
91, <u>94</u> , 94, 125, 703
MPI_TYPE_CREATE_INDEXED_BLOCK,
$12, \underline{98}, 98, 125$
MPI_TYPE_CREATE_KEYVAL, 306, <u>317</u> ,
320, 763, 788, 791, 927
MPI_TYPE_CREATE_RESIZED, 19, 91, 112,
115, 116, 125, 616, 704, 922
MPI_TYPE_CREATE_STRUCT, 12, 19, 91,
$\underline{99}, 99, 100, 113, 125, 183, 703$
MPI_TYPE_CREATE_SUBARRAY, 12, 15,
$\underline{101}, 103, 105, 125$
MPI_TYPE_DELETE_ATTR, 306, <u>320</u> , 320,
922
MPI_TYPE_DUP, 12, <u>119</u> , 119, 125, 922
MPI_TYPE_DUP_FN, <u>317</u> , 317, 779, 917
MPI_TYPE_EXTENT, 19, 703, 919
MPI_TYPE_F2C, <u>756</u>
MPI_TYPE_FREE, <u>118</u> , 127, 318, 427, 434
MPI_TYPE_FREE_KEYVAL, 306, <u>318</u> , 320
MPI_TYPE_GET_ATTR, 306, <u>319</u> , 320, 716,
764, 922
104, 522

MPI_TYPE_GET_CONTENTS, 125, <u>126</u> ,	1
127, 128	2
MPI_TYPE_GET_ELEMENTS, <u>121</u>	3
MPI_TYPE_GET_ELEMENTS_X, <u>121</u>	4
MPI_TYPE_GET_ENVELOPE, <u>124</u> , 125,	5
126, 128, 728	
MPI_TYPE_GET_EXTENT, <u>19</u> , <u>114</u> , 117,	6
703, 731, 761	7
MPI_TYPE_GET_EXTENT_X, <u>114</u> , 919	8
MPI_TYPE_GET_NAME, <u>326</u> , 922	9
MPI_TYPE_GET_TRUE_EXTENT, <u>116</u> , 116	10
MPI_TYPE_GET_TRUE_EXTENT_X, <u>116</u> ,	11
116, 919	
MPI_TYPE_HINDEXED, 19, 703, 919	12
MPI_TYPE_HVECTOR, 19, 703, 919	13
MPI_TYPE_INDEXED, 12, <u>95</u> , 95–97, 125	14
MPI_TYPE_LB, 19, 703, 919	15
MPI_TYPE_MATCH_SIZE, 708, <u>731</u> , 731, 922	16
MPI_TYPE_NULL_COPY_FN, <u>317</u> , 317, 779,	17
917 MPI_TYPE_NULL_DELETE_FN, <u>317</u> , 779,	18
	19
917, 922	20
MPI_TYPE_SET_ATTR, 306, <u>319</u> , 320, 716,	21
764, 767, 922	22
MPI_TYPE_SET_NAME, <u>325</u> , 922	23
MPI_TYPE_SIZE, <u>111</u> , 112, 644, 920	
MPI_TYPE_SIZE_X, <u>111</u> , 112, 919	24
MPI_TYPE_STRUCT, 19, 703, 919	25
MPI_TYPE_UB, 19, 703, 919	26
MPI_TYPE_VECTOR, 12, <u>92</u> , 92–94, 96, 125	27
MPI_UNPACK, <u>141</u> , 141, 142, 146, 622	28
MPI_UNPACK_EXTERNAL, 8, <u>148</u> , 729	29
MPI_UNPUBLISH_NAME, 403, <u>471</u> , 471	
MPI_WAIT, 33, 35, <u>60</u> , 60–64, 66, 67, 80, 81,	30
87, 208, 227, 249, 427, 450, 557, 561,	31
584, 585, 606, 624- $626, 736, 742, 743,$	32
746	33
MPI_WAITALL, 65, <u>67</u> , 67, 68, 208, 250, 450,	34
511, 559-561, 564	35
MPI_WAITANY, 46, <u>65</u> , 65, 66, 71, 450, 559,	
560, 564	36
MPI_WAITSOME, 65, <u>69</u> , 69–71, 450,	37
559-561, 564	38
MPI_WIN_ALLOCATE, 482, <u>485</u> , 486, 488,	39
494, 495, 500, 530, 720, 722, 916	40
MPI_Win_allocate, 914	41
MPI_WIN_ALLOCATE_CPTR, 486, 916	
MPI_WIN_ALLOCATE_SHARED, 281, 482,	42
<u>487</u> , 488, 490, 494, 495, 530, 722, 916	43
MPI_Win_allocate_shared, 914	44
MPI_WIN_ALLOCATE_SHARED_CPTR,	45
488, 916	46
MPI_WIN_ATTACH, 491, <u>492</u> , 492, 493, 530	47
MPI_WIN_C2F, <u>757</u>	48

	300
1	MPI_WIN_CALL_ERRHANDLER, <u>408</u> , 409
2	MPI_WIN_COMPLETE, 494, 519, <u>523</u> ,
3	524-526, 535, 542
4	MPI_WIN_CREATE, 451, 482, <u>483</u> , 485–488,
5	491, 493–495, 534
6	MPI_WIN_CREATE_DYNAMIC, 404, 482,
7	<u>491</u> , 491–495, 535 MPI_WIN_CREATE_ERRHANDLER, 392,
8	395, 396, 789, 791, 923
9	MPI_WIN_CREATE_KEYVAL, 306, <u>313</u> , 320,
10	763, 788, 791, 927
11	MPI_WIN_DELETE_ATTR, 306, <u>316</u> , 320
	MPI_WIN_DETACH, 491, <u>493</u> , 493, 494
12	MPI_WIN_DUP_FN, <u>314</u> , 314, 779, 917
13	MPI_WIN_F2C, <u>757</u>
14	MPI_WIN_FENCE, 494, 502, 518, 520, 521,
15	522, 532, 533, 535, 536, 539, 544, 746,
16	747
17	MPI_WIN_FLUSH, 489, 511, 513, <u>530</u> , 531,
18	535, 549, 551
19	MPI_WIN_FLUSH_ALL, 511, 513, <u>531</u> , 535
20	MPI_WIN_FLUSH_LOCAL, 511, <u>531</u> , 535
21	$MPI_WIN_FLUSH_LOCAL_ALL, 511, \underline{532},$
22	532, 535
23	MPI_WIN_FREE, 315, 427, 434, 479, <u>494</u> , 494
24	MPI_WIN_FREE_KEYVAL, 306, <u>315</u> , 320
	MPI_WIN_GET_ATTR, 306, <u>316</u> , 320, 495,
25	764, 767 MPI_WIN_GET_ERRHANDLER, <u>392</u> , <u>396</u> ,
26	928
27	MPI_WIN_GET_GROUP, <u>496</u> , 496
28	MPI_WIN_GET_INFO, 496, 497, 497, 914,
29	920
30	MPI_WIN_GET_NAME, <u>327</u>
31	MPI_WIN_LOCK, 484, 494, 519, <u>527</u> ,
32	528–530, 532, 534, 536, 539–541
33	MPI_WIN_LOCK_ALL, 484, 519, <u>528</u> , 528,
34	529, 532, 534, 536, 541, 549, 550
35	MPI_WIN_NULL_COPY_FN, <u>314</u> , 314, 779,
36	917
37	MPI_WIN_NULL_DELETE_FN, <u>314</u> , 779,
38	917 MDI WIN DOCT 101 510 502 504 504 505
	MPI_WIN_POST, 494, 519, 523, <u>524</u> , 524–527,

529, 532, 533, 535, 542, 544 MPI_WIN_SET_ATTR, 306, 315, 320, 495,

MPI_WIN_SET_ERRHANDLER, 392, 396

MPI_WIN_SET_INFO, 496, 496, 497, 914, 920

MPI_WIN_SHARED_QUERY, 487, 489, 722,

MPI_WIN_SHARED_QUERY_CPTR, 490,

MPI_WIN_START, 494, 519, 523, 523-527,

716, 764, 767

MPI_WIN_SET_NAME, 326

916

916

532, 533, 542, 549 MPI_WIN_SYNC, <u>532</u>, 532, 535–538, 542, 549 - 551MPI_WIN_TEST, <u>526</u>, 526 MPI_WIN_UNLOCK, 494, 513, 519, 528, 530, 535, 536, 539, 540 MPI_WIN_UNLOCK_ALL, 513, 519, 528, <u>529</u>, 535, 536, 539, 550 MPI_WIN_WAIT, 494, 519, 525, 525, 526, 529, 535, 536, 539, 542, 543 MPI_WTICK, 20, 410, 410, 676 MPI_WTIME, 20, 386, 409, 410, 644, 663, 676 mpiexec, 421, 422, 424, 446, 446, 915 mpirun, 446 PMPI_, 641, 716

PMPI_AINT_ADD, 20 PMPI_AINT_DIFF, 20 PMPI_ISEND, 716, 719 PMPI_T_EVENT_COPY, 677 PMPI_T_EVENT_GET_SOURCE, 677 PMPI_T_EVENT_GET_TIMESTAMP, 677 PMPI_T_EVENT_READ, 677 PMPI_T_PVAR_READ, 677 PMPI_T_PVAR_READRESET, 677 PMPI_T_PVAR_RESET, 677 PMPI_T_PVAR_START, 677 PMPI_T_PVAR_STOP, 677 PMPI_T_PVAR_WRITE, 677 PMPI_T_SOURCE_GET_TIMESTAMP, 677 PMPI_WTICK, 20 PMPI_WTIME, 20

39

40

 41

42

43

44

45

46

47