MPI-2: Extensions to the Message-Passing Interface

Message Passing Interface Forum

DRAFT processed on November 7, 1996
This work was supported in part by NSF and ARPA under NSF contract CDA-9115428 and Esprit under project HPC Standards (21111).

Abstract

This document describes the efforts of the MPI Forum to enhance the original MPI standard. Known as MPI-2, this effort began in March 1995. A draft version of the document is being released for public input at Supercomputing '96. The final version is anticipated in May 1997. The document has three parts: MPI-1.2, MPI-2, and MPI Journal of Development. Topics being explored for inclusion in MPI-2 are dynamic processes, one-sided communications, extended collective operations, external interfaces, additional language bindings, miscellaneous topics, and I/O.

The discussions of the MPI Forum for MPI-2 are all public. The following sources of information and input are available:

The minutes of the MPI Forum for MPI-2 are available from
http://www.mcs.anl.gov/Projects/mpi/mpi2/mpi2.html

Mailing lists are available of the electronic discussions held. The mailing lists are divided by chapter in the document. They are:

Chapter 4, Dynamic Processes conducts its discusions on `mpi-dynamic@mcs.anl.gov`.

Chapter 5, One-Sided Communications conducts its discusions on `mpi-1sided@mcs.anl.gov`.

Chapter 6, Extended Collective Operations conducts its discusions on `mpi-coll@mcs.anl.gov`.

Chapter 7, External Interfaces conducts its discusions on `mpi-external@mcs.anl.gov`.

Chapter 8, Language Binding conducts its discusions on `mpi-bind@mcs.anl.gov`.

Chapters 3 and 9, Miscellaneous conducts its discusions on `mpi-misc@mcs.anl.gov`.

Chapter 10, I/O conducts its discusions on `mpi-io@mcs.anl.gov`.

Chapter 11, Real-Time conducts its discusions on `mpi-realtime@mcs.anl.gov`.

To submit comments, send mail to the listed address. All these lists are maintained by Majordomo software. If you wish to subscribe to a list, send mail to Majordomo@mcs.anl.gov with the body of the message containing the line “subscribe ⟨address⟩”. General help can be gotten by sending mail to Majordomo@mcs.anl.gov with the line “help”.

Those wishing to receive information about meetings, etc. should subscribe to mpi-core@mcs.anl.gov.

Contents

Acknowledgments **xi**

1 Introduction to MPI-2 **1**
1.1 Background . . . 1
1.2 Organization of this Document . . . 2

2 MPI-2 Terms and Conventions **4**
2.1 Document Notation . . . 4
2.2 Procedure Specification . . . 4
2.3 Semantic Terms . . . 5
2.4 Data Types . . . 6
2.4.1 Opaque objects . . . 6
2.4.2 Array arguments . . . 7
2.4.3 State . . . 8
2.4.4 Named constants . . . 8
2.4.5 Choice . . . 8
2.4.6 Addresses . . . 8
2.5 Language Binding . . . 8
2.5.1 Fortran 77 Binding Issues . . . 9
2.5.2 A Fortran Problem with Register Optimization . . . 10
2.5.3 C Binding Issues . . . 12
2.5.4 C++ Binding Issues . . . 12
2.6 Processes . . . 14
2.7 Error Handling . . . 14
2.8 Implementation issues . . . 15
2.8.1 Independence of Basic Runtime Routines . . . 16
2.8.2 Interaction with signals in POSIX . . . 16

I MPI-1.2 **19**

3 Version 1.2 of MPI **21**
3.1 Version Number . . . 21
3.2 MPI-1.0 and MPI-1.1 Clarifications . . . 22
3.2.1 Clarification of MPI_FINALIZE . . . 22
3.2.2 Clarification of status after MPI_WAIT and MPI_TEST . . . 23
3.2.3 Clarification of MPI_INTERCOMM_CREATE . . . 24
3.2.4 Clarification of MPI_INTERCOMM_MERGE . . . 25

3.2.5 Clarification of Binding of MPI_Type_size 25
3.2.6 Clarification of MPI_REDUCE 25
3.2.7 Minor Corrections 25

II MPI-2 **35**

4 Process Creation and Management **37**
4.1 Introduction 37
4.2 The MPI-2 Process Model 38
4.2.1 Starting Processes 38
4.2.2 The Runtime Environment 38
4.3 Process Manager Interface 40
4.3.1 Processes in MPI 40
4.3.2 Starting Processes and Establishing Communication 40
4.3.3 Starting Multiple Executables and Establishing Communication . . 44
4.3.4 Nonblocking requests 47
4.3.5 Reserved keys 49
4.3.6 Spawn Examples 50
4.4 Establishing Communication 52
4.4.1 Names, addresses, ports, and all that 53
4.4.2 Server Routines 54
4.4.3 Client Routines 56
4.4.4 Name publishing 57
4.4.5 Non-blocking routines 59
4.4.6 Reserved key values 60
4.4.7 Client/Server Examples 60
4.5 Other routines 63
4.5.1 Info Management Routines 63
4.5.2 Universe Size 67
4.5.3 Singleton MPI_INIT 68
4.5.4 MPI_APPNUM 68
4.5.5 Releasing Connections 69

5 One-Sided Communications **71**
5.1 Introduction 71
5.2 Initialization 72
5.2.1 Memory Allocation 72
5.2.2 Window Creation 73
5.3 Communication Calls 76
5.3.1 Put 76
5.3.2 Get 78
5.3.3 Examples 79
5.3.4 Accumulate Functions 81
5.4 Synchronization calls 83
5.4.1 General 2-party synchronization 87
5.4.2 Barrier 90
5.4.3 Lock 91

5.4.4 Assertions 92
5.5 Examples 93
5.6 Semantics and Correctness 96
5.6.1 Atomicity 98
5.6.2 Progress 99
5.6.3 Registers and Compiler Optimizations 99
5.6.4 Implementation model 100

6 Extended Collective Operations 105
6.1 Introduction 105
6.2 Persistent collective operations 105
6.3 Limited Non-Blocking Collective Communication 107
6.3.1 Obtaining additional Non-Blocking Behavior in MPI-2 107
6.4 Intercommunicator Constructors 108
6.5 Extended Collective Operations 111
6.5.1 Intercommunicator Collective Operations 111
6.5.2 Operations that Move Data 112
6.5.3 Reductions 120
6.5.4 Generalized All-to-all Operation 122
6.5.5 Generalized All-to-all Function 123
6.6 Exclusive Scan 125
6.7 Channels 125

7 External Interfaces 127
7.1 Introduction 127
7.2 Generalized Requests 128
7.2.1 Introduction 128
7.2.2 Functionality 129
7.2.3 Failed User Functions and Cancelling Requests 135
7.2.4 Example of MPI_ALLGATHER 136
7.3 Blocking Instantiation of a Request 140
7.4 Associating Information with Status 141
7.5 Naming Objects 142
7.6 Adding New Error Classes and Error Codes 144
7.7 Accessing MPI Datatypes 146
7.7.1 Low Level Datatype Accessing 147
7.7.2 Datatype Encoding and Decoding 151
7.7.3 Canonical Datatype Representation 153
7.8 Identifying Requests 157
7.9 MPI and Threads 157
7.9.1 General 158
7.9.2 Clarifications 159
7.9.3 Initialization 160
7.9.4 User-defined Handlers 161

8 Language Bindings **166**
8.1 C++ Introduction . . . 166
8.1.1 Design . . . 166
8.1.2 C++ Classes for MPI . . . 167
8.1.3 Class Member Functions for MPI . . . 167
8.1.4 Semantics . . . 168
8.1.5 C++ Datatypes . . . 170
8.1.6 Mixed-language Inter-operability . . . 172
8.1.7 Profiling . . . 173
8.2 Fortran 90 Support . . . 175
8.2.1 Overview . . . 176
8.2.2 Basic Fortran Support . . . 176
8.2.3 Extended Fortran Support . . . 179

9 Miscellany **182**
9.1 Portable MPI Process Startup . . . 182
9.2 Convenient Form of MPI_Type_indexed_block . . . 184
9.3 Treatment of MPI_Status . . . 184
9.3.1 Passing MPI_STATUS_IGNORE for MPI_Status . . . 185
9.3.2 Non-destructive Test of MPI_Status . . . 185
9.4 Error Class for Invalid Keyval . . . 186
9.5 Determining Whether MPI Has Finished . . . 186
9.6 Handle Types . . . 186
9.7 Language Interoperability . . . 187
9.7.1 Introduction . . . 187
9.7.2 Assumptions . . . 187
9.7.3 Initialization . . . 188
9.7.4 Transfer of handles . . . 188
9.7.5 MPI Opaque objects . . . 192
9.7.6 Addresses . . . 193
9.7.7 Reduce operations . . . 196
9.7.8 Status . . . 196
9.7.9 Constants . . . 196
9.7.10 Interlanguage communication . . . 196
9.8 New datatype manipulation functions . . . 197
9.8.1 Type constructors with explicit addresses . . . 198
9.8.2 Extent and Bounds of Datatypes . . . 200
9.8.3 True Extent of Datatypes . . . 201
9.9 New attribute caching functions . . . 202
9.10 Continuable Errors . . . 203
9.11 New Elementary Datatypes . . . 204
9.11.1 Wide Characters . . . 204
9.11.2 Signed Characters and Reductions . . . 205

10 I/O **206**
10.1 Introduction . . . 206
10.1.1 Definitions . . . 206
10.2 File Manipulation . . . 208
10.2.1 Opening a File . . . 208
10.2.2 Nonblocking Open . . . 209
10.2.3 Closing a file . . . 210
10.2.4 Nonblocking Close . . . 210
10.2.5 Deleting a file . . . 211
10.2.6 Resizing a file . . . 211
10.2.7 Querying the size of a file . . . 212
10.2.8 Querying File Parameters . . . 212
10.2.9 File Info . . . 213
10.3 File Views . . . 216
10.4 Data Access . . . 218
10.4.1 Data Access Routines . . . 218
10.4.2 Data Access With Explicit Offsets . . . 220
10.4.3 Data Access With Individual File Pointers . . . 224
10.4.4 Data Access with Shared File Pointers . . . 228
10.5 File Interoperability . . . 234
10.5.1 Datatype Size and Extent in Files . . . 237
10.5.2 Canonical Data Representation . . . 237
10.6 Consistency and Semantics . . . 237
10.6.1 File Consistency . . . 237
10.6.2 Progress . . . 240
10.6.3 Type Matching . . . 240
10.6.4 Logical vs. Physical File Layout . . . 240
10.7 Filetype Constructors . . . 241
10.7.1 Introduction . . . 241
10.7.2 Scatter / Gather Type Constructors . . . 241
10.7.3 Local Array Filetype Constructor . . . 242
10.7.4 Distributed Array Filetype Constructors . . . 243
10.8 I/O Error Classes . . . 246
10.9 Examples . . . 247
10.9.1 Scatter / Gather Filetype Constructor Example . . . 247
10.9.2 Local Array Filetype Constructor Example . . . 248

III MPI Journal of Development **251**

11 Real-Time MPI **253**
11.1 Introduction . . . 253
11.2 Real-time Message-Passing Requirements . . . 255
11.3 RT Profiles . . . 255
11.3.1 Resource Constrained MPI 1.2 . . . 256
11.3.2 Other profiles . . . 256
11.4 RT Initialization/Termination . . . 257
11.5 Clocks . . . 257

11.5.1 Synchronization of Clocks 257
11.5.2 Description of the Clocks 258
11.5.3 Clock Synchronization Parameters 258
11.5.4 MPI/RT Clock Attributes 259
11.6 Real-time Buffer Pools 260
11.6.1 Buffer Pool Object Creation and Destruction 260
11.6.2 User Buffer Access 261
11.7 Channels 263
11.7.1 Point-to-point Channels 263
11.7.2 Collective Operations with Quality of Service 269
11.8 Quality of Service 271
11.9 Time-driven MPI/RT 272
11.9.1 Scheduling Message Transfers 272
11.9.2 Schedulable Time Intervals 273
11.9.3 MPI/RT Time Handle 273
11.9.4 Time-Driven Channel Calls 274
11.10 Event-Driven MPI/RT 276
11.10.1 Low-Level, Event-Driven Real-Time MPI 277
11.10.2 High-Level, Event-Driven Real-Time MPI 283
11.11 Priority-Driven MPI/RT 292
11.11.1 Message Priority 292
11.11.2 Process Priority 292
11.12 Ad Hoc MPI/RT 293
11.13 Soft MPI/RT 293
11.14 Issues in Resource Constrained Systems 293
11.14.1 Resource Constrained MPI 294
11.15 Instrumentation 294
11.15.1 MPI/RT Monitoring 295
11.16 Timing Correctness 298
11.17 Deadlock Avoidance/Recovery 298
11.18 Fault Tolerance 299
11.18.1 Replication Policies 299
11.18.2 MPI Communication Fault Tolerance 300
11.18.3 Timeouts for Traditional MPI Functionality 300
11.19 Communicator Semantics with Channels 300
11.20 Errors and Error Codes 301
11.21 APPENDIX: Quality of Service 301
11.22 APPENDIX: Deadlock Avoidance/Recovery 305

12 Spawning Independent Processes **306**
12.1 Starting Independent Processes 306
12.1.1 MPI processes 307
12.1.2 Non-MPI processes 307
12.2 Starting multiple independent processes 308
12.3 Nonblocking operations 309
12.4 Signalling Processes 310
12.5 Notification of change in state of a process 311

13 Threads and MPI **314**
13.1 Global thread functions . . . 314
13.1.1 Mutexes . . . 314
13.1.2 Condition variables . . . 316

14 Caching on MPI Handles **318**
14.1 Introduction . . . 318
14.2 Functions . . . 318
14.3 Example . . . 322

15 Miscellany **323**
15.1 New Datatype Constructors . . . 323
15.1.1 Simple Struct Types . . . 323
15.1.2 Contiguous Struct Types . . . 324
15.2 Two Process "Shared" Buffers . . . 327
15.3 Yet Another Send Mode . . . 333

16 Towards a Full Fortran-90 Interface **334**
16.1 A different way of handling buffer arguments . . . 334
16.1.1 MPI_F90_DESCRIBE_OBJECT and relatives . . . 334
16.2 Using a Fortran 90 - ANSI C interface . . . 335
16.3 Typed functions . . . 336
16.4 The possibility of optional arguments . . . 336
16.5 Derived types for MPI handles . . . 336

Bibliography **337**

A Language Binding **339**
A.1 Introduction . . . 339
A.2 Defined Constants, Error Codes, and Info Keys . . . 339
A.2.1 Constants . . . 339
A.2.2 Error Codes . . . 341
A.2.3 Info Keys . . . 341
A.3 MPI-1.2 C bindings . . . 342
A.4 MPI-1.2 Fortran bindings . . . 342
A.5 MPI-2 C bindings . . . 342
A.5.1 Process Creation and Management . . . 342
A.5.2 One-Sided Communications . . . 344
A.5.3 Extended Collective Operations . . . 344
A.5.4 External Interfaces . . . 345
A.5.5 Miscellany . . . 346
A.5.6 I/O . . . 347
A.6 MPI-2 C functions . . . 349
A.7 MPI-2 Fortran bindings . . . 350
A.7.1 Process Creation and Management . . . 350
A.7.2 One-Sided Communications . . . 352
A.7.3 Extended Collective Operations . . . 353
A.7.4 External Interfaces . . . 353

A.7.5 Miscellany 355
A.7.6 I/O 356
A.8 MPI-2 Fortran Subroutines 360

B MPI-1 C++ Language Binding - Proposal 1 **361**
B.1 C++ Classes 361
B.2 Defined Constants 361
B.3 Typedefs 365
B.4 C++ Bindings for Point-to-Point Communication 365
B.5 C++ Bindings for Collective Communication 368
B.6 C++ Bindings for Groups, Contexts, and Communicators 369
B.7 C++ Bindings for Process Topologies 370
B.8 C++ Bindings for Environmental Inquiry 371
B.9 C++ Bindings for Profiling 372
B.10 C++ Bindings for Status Access 372
B.11 C++ Bindings on all MPI Classes 372
B.11.1 Construction / Destruction 373
B.11.2 Copy / Assignment 373
B.11.3 Comparison 373
B.11.4 Inter-language Operability 373

C MPI-1 C++ Language Binding - Proposal 2 **374**
C.1 C++ Classes 374
C.2 Defined Constants 374
C.3 Typedefs 378
C.4 C++ Bindings for Point-to-Point Communication 379
C.5 C++ Bindings for Collective Communication 381
C.6 C++ Bindings for Groups, Contexts, and Communicators 382
C.7 C++ Bindings for Process Topologies 383
C.8 C++ Bindings for Environmental Inquiry 384
C.9 C++ Bindings for Profiling 385
C.10 C++ Bindings for Status Access 385
C.11 C++ Bindings on all MPI Classes 385
C.11.1 Construction / Destruction 386
C.11.2 Copy / Assignment 386
C.11.3 Comparison 386
C.11.4 Inter-language Operability 386

MPI Function Index **387**

Acknowledgments

This document represents the work of many people who have served on the MPI Forum. The meetings have been attended by dozens of people from many parts of the world. It is the hard and dedicated work of this group that has lead 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-2), many people helped with this effort. Those who served as the primary coordinators are:

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

The following list includes some of the active participants in the MPI-2 process not mentioned above.

Greg Astfalk	Robert Babb	Rajesh Bordawekar	Pete Bradley
Ron Brightwell	Maciej Brodowicz	Eric Brunner	Greg Burns
Margaret Cahir	Pang Chen	Ying Chen	Albert Cheng
Yong Cho	Joel Clark	Lyndon Clarke	Laurie Costello
Dennis Cottel	Jim Cownie	Raja Daoud	David DiNucci
Doug Doefler	Jack Dongarra	Terry Dontje	Nathan Doss
Anne Elster	Mark Fallon	Sam Fineberg	Stephen Fleischman
Ian Foster	Hubertus Franke	Richard Frost	Al Geist
Robert George	David Greenberg	Bill Gropp	Kei Harada
Leslie Hart	Shane Hebert	Rolf Hempel	Tom Henderson
Alex Ho	Hans-Christian Hoppe	Joefon Jann	Suresh Damondovan Kamal
Karl Kesselman	Koichi Konishi	Susan Kraus	Steve Landherr
Mario Lauria	Mark Law	Juan Leon	Lloyd Lewins
Ziyang Lu	Peter Madams	John May	Oliver McBryan
Brian McCandless	Thom McMahon	Harish Nag	Nick Nevin
Jarek Nieplocha	Ron Oldfield	Steve Otto	Pratap Pattnaik
Peter Pacheco	Yoonho Park	Perry Partow	Elsie Pierce
Paul Pierce	Heidi Poxon	JeanPierre Prost	James Pruyve
Rolf Rabenseifner	Joe Rieken	Peter Rigsbee	Anna Rounbehler
Nobutoshi Sagawa	Arindam Saha	Eric Salo	Eric Sharakan
Andrew Sherman	Fred Shirley	Lance Shuler	A. Gordon Smith
Jeff Squyres	David Taylor	Steven Taylor	Greg Tensa
Rajeev Thakur	Marydell Tholburn	Dick Treumann	Simon Tsang
Manuel Ujaldon	David Walker	Jerrell Watts	Klaus Wolf
Parkson Wong	Dave Wright		

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 ARPA under NSF contract CDA-9115428 for travel by U.S. academic participants and Esprit under project HPC Standards (21111) for European participants.

Chapter 1

Introduction to MPI-2

Current Status: No votes on entire chapter

1.1 Background

Beginning in March 1995, the MPI Forum began meeting to consider corrections and extensions to the original MPI Standard document [7]. The first product of these deliberations was Version 1.1 of the MPI specification, released in June of 1995 (See http://www.mcs.anl.gov/mpi for official MPI document releases). Since that time, effort has been focused in four types of 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, version identification, 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. 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. 0-copy semantics on shared-memory machines, real-time specifications.)

Opinions will vary about what category various sections and subsections belong to.

This document is a snapshot of the Forum's progress as of November, 1996. The precise state of individual sections is indicated by the "**Current Status:**" notation. The MPI Forum accepts sections into the Standard by a process that culminates in two votes of acceptance at two separate Forum meetings. Thus those sections that are annotated with a current status of "two votes" will be part of the MPI-2 Standard, although small changes may be made through amendments. Sections annotated with "one vote" have been approved by one of the required votes, and are awaiting second vote at an upcoming meeting. Sections annotated "no votes" are still under discussion and have not been presented for voting to the entire Forum by their associated chapter subcommittee.

The MPI Forum welcomes comments on all parts of this document. Comments should be sent to `mpi-comment@mcs.anl.gov`. Comments will have the most impact if received before the next MPI Forum meeting in late January, 1997.

It is expected that in May of 1997 the final version of this document will be released. It is expected to contain:

- MPI-1.2, containing items of category (1) and perhaps some of (2) above, which have been fully voted on (reading + two official votes).
- MPI-2.0, containing fully-voted-on (reading + two official votes) items in categories (2) and (3) above.
- MPI Journal of Development (JOD), containing items of categories (3) and (4) that the Forum believes at this point should not be included in the Standard.

The division into three documents will make it easier for vendors and users to have a discourse about what level of MPI compliance a given implementation has:

- MPI-1 compliance will mean compliance with MPI 1.2. This is a useful level of compliance. The idea is that the original message-passing model, as envisioned during the MPI-1 process, is fully supported, and there will be changes required from Version 1.1, but no significant new types of functionality.
- MPI-2 compliance will mean compliance with MPI 2.0.

Guidance for future work will be provided in the MPI Journal of Development.

It is to be emphasized that forward compatibility will be preserved. That is, a valid MPI-1.1 program will be both a valid MPI-1.2 program and a valid MPI-2 program, and a valid MPI-1.2 program will be a valid MPI-2 program.

1.2 Organization of this Document

This document is in three parts. The first contains proposed corrections, clarifications, and modifications to the MPI-1.1 Standard, which together will become MPI-1.2. The small number of new functions in MPI-1.2 are extensions to MPI-1.1 concepts, but do not add any new types of functionality. They are not anticipated to require extensive additions or changes to existing implementations. They are expected to require some minor modifications.

The second part of this document contains the MPI-2 Standard Specification. It adds substantial new types of functionality to MPI, in most cases specifying functions for an extended computational model (e.g., dynamic process creation and one-sided communication) or for a significant new capability (e.g., parallel I/O).

The following is a list of the chapters in MPI-2, along with a brief description of each.

- Chapter 2, MPI-2 Terms and Conventions, explains notational terms and conventions used throughout the MPI-2 document.
- Chapter 4, Dynamic Processes, discusses the extension of MPI to remove the static process model in MPI. It gives routines to allow for creation of processes.
- Chapter 5, One-Sided Communications, defines communication routines which are one directional. These include shared memory type operations (put/get), and remote accumulate operations.

- Chapter 6, Extended Collective Operations, gives the extensions to MPI to allow non-blocking collective operations, apply collective operations to inter-communicators, and additional features.
- Chapter 7, External Interfaces, defines those routines designed for allowing developers to layer on top of MPI. This includes generalized requests, routines to decode MPI opaque objects, and handlers and threads.
- Chapter 8, Language Binding, describes the C++ binding and discusses Fortran-90 issues.
- Chapter 9, Miscellaneous, discusses items that don't fit elsewhere, in particular language interoperability.
- Chapter 10, I/O, discusses MPI-2 support for parallel I/O.

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

- Chapter 11, Real Time, discusses MPI support for real time processing.
- Chapter 12, Spawning Independent Processes includes some elements of dynamic process management, in particular management of processes with which the spawning processe do not intend to communicate, that the Forum discussed at length but ultimately decided not to include in the MPI Standard.
- Chapter 13, Threads and MPI describes some of the expected interaction between an MPI implementation and a thread library in a multi-threaded environment.
- Chapter 14, Caching on MPI Objects, describes some elements of external interface definitions (in particular caching on MPI objects other than communicators) that the Forum decided not to include in the Standard.
- Chapter 15, Miscellany, discusses Miscellaneous topics in the MPI JOD, in particular single-copy routines for use in shared-memory environments and new datatype constructors.
- Chapter 16, Toward a Full Fortran 90 Interface, describes an approach to providing a more elaborate Fortran 90 interface.

This document does not address the issue of interoperability among various MPI implementations. The Forum is still considering this issue.

The MPI Function Index is a simple index showing the location of the precise definition of each MPI-2 function, together with C, C++, and Fortran bindings. The Appendices are:

- Annex A, Specific Language Bindings, gives bindings for MPI-2 functions, and lists constants, error codes, etc.
- Annex B and C, MPI-1 C++ Language Binding, give the two C++ bindings under consideration for MPI-1.

Chapter 2

MPI-2 Terms and Conventions

Current Status: No votes on entire chapter

This chapter explains notational terms and conventions used throughout the MPI-2 document, some of the choices that have been made, and the rationale behind those choices.

Missing:
This chapter will need changes as MPI-2 develops. Issue like thread interaction, signals, etc. will need to be updated as we make other decisions.

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 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 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:

- the call may use the input value but does not update an argument is marked IN,
- the call may update an argument but does not use its input value is marked OUT,

- the call may both use and update an argument is marked INOUT.

There is one special case — if an argument is a handle to an opaque object (these terms are defined in Section 2.4.1), and the object is updated by the procedure call, then the argument is marked OUT. It is marked this way even though the handle itself is not modified — we use the OUT attribute to denote that what the handle *references* is updated. Thus, in C++, IN arguments are either references or pointers to `const` objects.

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

A common occurrence for MPI functions is an argument that is used as IN by some processes and OUT by other processes. Such 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 argument.

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

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

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

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

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

All MPI functions are first specified in the language-independent notation. Immediately below this, the ANSI C version of the function is show followed by a version of the same function in Fortran 77 and then the C++ binding.

Missing:
F90 bindings

2.3 Semantic Terms

When discussing MPI procedures the following semantic terms are used. The first two are usually applied to communication operations.

nonblocking If the procedure may return before the operation completes, and before the user is allowed to re-use resources (such as buffers) specified in the call.

blocking If return from the procedure indicates the user is allowed to re-use resources specified in the call.

local If completion of the procedure depends only on the local executing process. Such an operation does not require communication with another user process.

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 If all processes in a process group need to invoke the procedure.

2.4 Data Types

2.4.1 Opaque objects

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

In Fortran, all handles have type INTEGER. In C and C++, a different handle type is defined for each category of objects. In addition, handles themselves are distinct objects in C++. The C and C++ types must support the use of the assignment and equality operators.

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

Missing:

what about F90?

(*End of advice to implementors.*)

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

A call to a deallocate routine invalidates the handle and marks the object for deallocation. The object is not accessible to the user after the call. However, MPI need not deallocate the object immediatly. 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.

Missing:

If we allow datatype transfers, we may need to add something like:

In some cases, e.g. datatypes, special functions exist which allow transfer of the "opaque" object.

MPI provides certain predefined opaque objects and predefined, static handles to these objects. Such objects may not be destroyed. In C++, this is enforced by declaring the handles to these predefined objects to be `static const`.

Rationale. This design hides the internal representation used for MPI data structures, thus allowing similar calls in C 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 separating of handles in user space and objects in system space, allows space-reclaiming and deallocation calls to be made at appropriate points in the user program. If the opaque objects were in user space, one would have to be very careful not to go out of scope before any pending operation requiring that object completed. The specified design allows an object to be marked for deallocation, the user program can then go out of scope, and the object itself still persists until any pending operations are complete.

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

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

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

2.4.2 Array arguments

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

2.4.3 State

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

2.4.4 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 can be queried using environmental inquiry functions (Chapter 7 of the MPI-1 document).

2.4.5 Choice

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

2.4.6 Addresses

Some MPI procedures use *address* arguments that represent an absolute address in the calling program. The datatype of such an argument is an integer of the size needed to hold any valid address in the execution environment.

Missing:
This may become MPI_Aint or other things depending on what we decide.

2.5 Language Binding

This section defines the rules for MPI language binding in general and for Fortran 77 and ANSI C in particular. Defined here are various object representations, as well as the naming conventions used for expressing this standard. The actual calling sequences are defined elsewhere.

It is expected that any Fortran 90 implementations use the Fortran 77 bindings. Although we consider it premature to define other bindings to Fortran 90, the current bindings are designed to encourage, rather than discourage, experimentation with better bindings that might be adopted later.

Missing:
If we do F90 then this needs to be changed.

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

2.5.1 Fortran 77 Binding Issues

All MPI names have an `MPI_` prefix, and all characters are capitals. Programs must not declare variables or functions with names beginning with the prefix, `MPI_`. This is mandated to avoid possible name collisions.

All MPI Fortran subroutines have a return code in the last argument. A few MPI operations which are functions do not have the return code argument. The return code value for successful completion is MPI_SUCCESS. Other error codes are implementation dependent; see the error codes in Chapter 7 of the MPI-1 document.

Missing:
where are the new error codes in MPI-2 document.

Handles are represented in Fortran as `INTEGER`s. Binary-valued variables are of type `LOGICAL`.

Array arguments are indexed from one.

Unless explicitly stated, the MPI F77 binding is consistent with ANSI standard Fortran 77. There are several points where this standard diverges from the ANSI Fortran 77 standard. These exceptions are consistent with common practice in the Fortran community. In particular:

- MPI identifiers are limited to thirty, not six, significant characters.

  Discussion:

  Did we want 30 or 31? Assume upper limit is 32 and need 1 for profiling??

- MPI identifiers may contain underscores after the first character.

- An MPI subroutine with a choice argument may be called with different argument types. An example is shown in Figure 2.1. This violates the letter of the Fortran standard, but such a violation is common practice. An alternative would be to have a separate version of MPI_SEND for each data type.

- Although not required, it is strongly suggested that named MPI constants (`PARAMETER`s) be provided in an include file, called `mpif.h`. On systems that do not support include files, the implementation should specify the values of named constants.

```
double precision a
integer b
...
call MPI_send(a,...)
call MPI_send(b,...)
```

Figure 2.1: An example of calling a routine with mismatched formal and actual arguments.

- Vendors are encouraged to provide type declarations in the `mpif.h` file on Fortran systems that support user-defined types. One should define, if possible, the type MPI_ADDRESS, which is an INTEGER of the size needed to hold an address in the execution environment. On systems where type definition is not supported, it is up to the user to use an INTEGER of the right kind to represent addresses (i.e., `INTEGER*4` on a 32 bit machine, `INTEGER*8` on a 64 bit machine, etc.).

2.5.2 A Fortran Problem with Register Optimization

Current Status: Passed once.

Advice to users.

MPI provides operations which may be hidden from the user code and run in parallel 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. This section discusses register optimization pitfalls.

When a variable is local to a Fortran subroutine (i.e. not in a `COMMON block`), the compiler will assume that it cannot be modified by a called subroutine unless it is an actual argument of the call. In the most common linkage convention, the subroutine is expected to save and restore certain registers. Thus, the optimizer will assume that a register which held a valid copy of such a variable before the call will still hold a valid copy on return.

Normally users are not afflicted with this. But the user should pay attention to this section if in his/her program a buffer argument to an MPI_SEND, MPI_RECV etc. uses a name which hides the actual variables involved. MPI_BOTTOM with an MPI_Datatype containing absolute addresses is one example. Creating a datatype which uses one variable as an anchor and brings along others by using MPI_ADDRESS to determine their offsets from the anchor is another. The anchor variable would be the only one mentioned in the call. Also attention must be payed if MPI operations are used that run in parallel with the user's application.

The following example shows, what Fortran compilers are allowed to do:

This source ...	can be compiled as:
call MPI_ADDRESS(buf,bufaddr,ierr)	call MPI_ADDRESS(buf,...)
call MPI_TYPE_STRUCT(1,1,bufaddr, MPI_REAL,typ,ierr)	call MPI_TYPE_STRUCT(...)
call MPI_TYPE_COMMIT(typ)	call MPI_TYPE_COMMIT(...)
val_old = buf	register = buf val_old = register
call MPI_RECV(MPI_BOTTOM,1,typ,...)	call MPI_RECV(MPI_BOTTOM,...)
val_new = buf	val_new = register

The compiler does not invalidate the register because it cannot see that MPI_RECV changes the value of buf. The access of buf is hidden by the use of MPI_ADDRESS and MPI_BOTTOM.

The next example shows extreme, but allowed possibilities!

Source	compiled as	or compiled as
call MPI_IRECV(buf,..req)	call MPI_IRECV(buf,..req)	call MPI_IRECV(buf,..req)
	register = buf	b1 = buf
call MPI_WAIT(req,..)	call MPI_WAIT(req,..)	call MPI_WAIT(req,..)
b1 = buf	b1 := register	

MPI_WAIT or a parallel thread modifies buf between the invocation of MPI_IRECV and the finish of MPI_WAIT. But the compiler cannot see any possibility that buf can be changed after MPI_IRECV has finished. And therefore it can schedule the load of buf earlier than typed in the source. It has no reason to avoid using a register to hold buf across the call to MPI_WAIT. It also can reorder the instructions as in the left case.

To prevent instruction reordering or the allocation of a variable or buffer in a register, there are three possibilities in portable Fortran code:

- One can put the variables and buffers into common blocks; then they will be allocated in the memory while any library subroutine (e.g. an MPI routine) is called provided that the compiler cannot analyze that the library subroutine does not use these common blocks, or
- one can declare them as VOLATILE, but this is not part of the standards Fortran 77 and 90, and your code will not be portable because there are compilers without VOLATILE, and normally it prevents any optimization, or
- one can call a dummy routine with the variable or buffer as an actual argument; if the dummy routine is not part of the application then the compiler must allocate the variable or buffer in the memory while that subroutine is executed; MPI_ADDRESS can be used as dummy routine.

High quality Fortran implementations may provide mechanisms to avoid the problem, without disabling optimizations. However, code that does not use one of the three solutions described above may not be portable.

In C, subroutines which modify variables that are not in the parameter list will not cause register optimization problems. This is because taking pointers to storage objects by using the & operator and later referencing the objects by way of the pointer is an integral part of the language. A C compiler understands the implications, so that the problem should not occur, in general. However, some compilers do offer optional aggressive optimization levels which may not be safe.

(*End of advice to users.*)

2.5.3 C Binding Issues

We use the ANSI C declaration format. All MPI names have an `MPI_` prefix, defined constants are in all capital letters, and defined types and functions have one capital letter after the prefix. Programs must not declare variables or functions with names beginning with the prefix, `MPI_`. This is mandated to avoid possible name collisions.

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. A few C functions do not return values, so that they can be implemented as macros.

Type declarations are provided for handles to each category of opaque objects. Either a pointer or an integer type is used.

Array arguments are indexed from zero.

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

Choice arguments are pointers of type `void*`.

Address arguments are of MPI defined type MPI_Aint. This is defined to be an int of the size needed to hold any valid address on the target architecture.

2.5.4 C++ Binding Issues

We use the ANSI C++ declaration format. All MPI names are declared within the scope of a namespace called `MPI` and therefore are referenced with an `MPI::` prefix. Defined constants are in all capital letters, and class names, defined types, and functions have only their first letter capitalized. Programs must not declare variables or functions in the `MPI` namespace. This is mandated to avoid possible name collisions.

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

C++ functions that create objects or return information usually place the object or information in the return value. Other C++ functions return `void`.

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

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

Array arguments are indexed from zero.

Logical flags are of type `bool`.

Choice arguments are pointers of type `void*`.

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

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

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

1. All functions, types, and constants are declared within the scope of a `namespace` called `MPI`.

2. Arrays of MPI handles are always left in the argument list (whether they are IN or OUT arguments).

3. If the argument list of an MPI function contains a scalar IN handle, and it makes sense to define the function as a method of the object corresponding to that handle, the function is made a member function of the corresponding MPI class. The member functions are named according to the corresponding MPI function name, but without the "MPI_" prefix and without the object name prefix (if applicable). In addition:

   (a) The scalar IN handle is dropped from the argument list, and `this` corresponds to the dropped argument.

   (b) The function is declared `const`.

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

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

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

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

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

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

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

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

9. Any IN pointer, reference, or array argument must be declared `const`.
10. Handles are passed by reference.
11. Array arguments are denoted with square brackets (`[]`), not pointers, as this is more semantically precise.

2.6 Processes

An MPI program consists of autonomous processes, executing their own code, in an MIMD style. The codes executed by each process need not be identical. The processes communicate via calls to MPI communication primitives. Typically, each process executes in its own address space, although shared-memory implementations of MPI are possible. This document specifies the behavior of a parallel program assuming that only MPI calls are used for communication. The interaction of an MPI program with other possible means of communication (e.g., shared memory) is not specified.

MPI does not specify the execution model for each process. A process can be sequential, or can be multi-threaded, with threads possibly executing concurrently.. Care has been taken to make MPI "thread-safe," by avoiding the use of implicit state. The desired interaction of MPI with threads is that concurrent threads be all allowed to execute MPI calls, and calls be reentrant; a blocking MPI call blocks only the invoking thread, allowing the scheduling of another thread.

2.7 Error Handling

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

Missing:
if we add dynamic processes that can go away then this needs to be modified.

Of course, MPI programs may still be erroneous. A **program error** can occur when an MPI call is called 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.

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. However, MPI provides mechanisms for users to change this default and to handle recoverable errors. The user may specify that no error is fatal, and handle error codes returned by MPI calls by himself or herself. Also, the user may provide his or her own error-handling routines, which will be invoked whenever an MPI call returns abnormally. The MPI error handling facilities are described in Chapter 7 of the MPI-1 document.

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

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

Discussion:
Do we really mean it must be fatal? What if error handlers are being used?

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

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

2.8 Implementation issues

Missing:
Once we decide on signals and threads this needs to be redone.

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 I/O or signal handling) be provided, it does strongly suggest the behavior to be provided if those services are available. This is an important point in achieving portability across platforms that provide the same set of services.

2.8.1 Independence of Basic Runtime Routines

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

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

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

The corresponding Fortran 77 and C++ programs are also expected to complete.

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

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

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

2.8.2 Interaction with signals in POSIX

MPI does not specify either the interaction of processes with signals, in a UNIX environment, or with other events that do not relate to MPI communication. That is, signals are not significant from the view point of MPI, and implementors should attempt to implement MPI so that signals are transparent: an MPI call suspended by a signal should resume and complete after the signal is handled. Generally, the state of a computation that is visible or significant from the view-point of MPI should only be affected by MPI calls.

The intent of MPI to be thread and signal safe has a number of subtle effects. For example, on Unix systems, a catchable signal such as SIGALRM (an alarm signal) must not cause an MPI routine to behave differently than it would have in the absence of the signal. Of course, if the signal handler issues MPI calls or changes the environment in which the MPI routine is operating (for example, consuming all available memory space), the MPI routine should behave as appropriate for that situation (in particular, in this case, the behavior should be the same as for a multithreaded MPI implementation).

A second effect is that a signal handler that performs MPI calls must not interfere with the operation of MPI. For example, an MPI receive of any type that occurs within a signal handler must not cause erroneous behavior by the MPI implementation. Note that an implementation is permitted to prohibit the use of MPI calls from within a signal handler, and is not required to detect such use.

It is highly desirable that MPI not use `SIGALRM`, `SIGFPE`, or `SIGIO`. An implementation is *required* to clearly document all of the signals that the MPI implementation uses; a good place for this information is a Unix '`man`' page on `MPI`.

Part I

MPI-1.2

Chapter 3

Version 1.2 of MPI

This section contains clarifications and minor corrections to Version 1.1 of the MPI Standard. The only new function here is one for identifying which version of the MPI Standard the implementation being used conforms to. There are small differences between MPI-1 and MPI-1.1. There are very few differences (only those discussed in this chapter) between MPI-1.1 and MPI-1.2, but large differences (the rest of this document) between MPI-1.2 and MPI-2.

3.1 Version Number

Current Status: Passed twice.

In order to cope with changes to the MPI Standard, there are both compile-time and run-time 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 and C++,

```
#define MPI_VERSION    1
#define MPI_SUBVERSION 2
```

in Fortran,

```
INTEGER MPI_VERSION, MPI_SUBVERSION
PARAMETER (MPI_VERSION    = 1)
PARAMETER (MPI_SUBVERSION = 2)
```

For runtime determination,

MPI_GET_VERSION(version, subversion)

OUT	version	version number
OUT	subversion	subversion number

```
int MPI_Get_version( int *version, int *subversion )
```

```
MPI_GET_VERSION( VERSION, SUBVERSION, IERROR )
    INTEGER VERSION, SUBVERSION, IERROR
```

3.2 MPI-1.0 and MPI-1.1 Clarifications

3.2.1 Clarification of MPI_FINALIZE

Current Status: Passed once.

This routine cleans up all MPI state. Each process must call MPI_FINALIZE before it exits.

A successful return from a blocking communication operation or from MPI_WAIT or MPI_TEST tells the user that the buffer can be reused and means that the communication is completed by the user, but does not guarantee that the local process has no more work to do. A successful return from MPI_REQUEST_FREE with a request handle generated by a non-blocking send operation only frees the request handle but also means that the communication is completed by the user provided that the communication can be done due to an appropriate receive operation, but slso does not guarantee that the local process has no more work to do.

MPI_FINALIZE guarantees that all local actions required by communications the user has completed will, in fact, occur before it returns. MPI_FINALIZE guarantees nothing about pending communications that have not been completed by MPI_WAIT, MPI_TEST, or MPI_REQUEST_FREE, or about pending receive operations that have been completed with MPI_REQUEST_FREE, and, in fact, the behavior of MPI_FINALIZE is undefined in this case.

Once this routine returns, no MPI routine (even not MPI_INIT) may be called. Each process must complete any pending communication it initiated before it calls MPI_FINALIZE. If the call returns, each process may continue local computations, or exit, without participating in further MPI communication with other processes. MPI_FINALIZE is collective on MPI_COMM_WORLD.

> *Advice to implementors.* Even though a process has completed all the communication it initiated, such communication may not yet be completed from the viewpoint of the underlying MPI system. E.g., a blocking send may have completed, even though the data is still buffered at the sender. The MPI implementation must ensure that a process has completed any involvement in MPI communication before MPI_FINALIZE returns. Thus, if a process exits after the call to MPI_FINALIZE, this will not cause an ongoing communication to fail. (*End of advice to implementors.*)

Although it is not required that all processes return from MPI_FINALIZE, it is required that at least process 0 in MPI_COMM_WORLD return, so that users can know that the MPI portion of the computation is over. In addition, in a POSIX environment, they may desire to supply an exit code for each process that returns from MPI_FINALIZE.

Example 3.1 *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);
```

Discussion: The alternative would be to have a predefined “survivor” constant. It seems simplest just to require that it be 0.

Allowing User Functions at Process Termination

Current Status: No votes

There are times in which it would be convenient to have actions happen when an MPI process finishes. For example, a routine may do initializations that are useful until the MPI job (or that part of the job that being terminated in the case of dynamically created processes) is finished. This can be accomplished in MPI-2 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 on all processes that are about to complete their participation in MPI. This will cause the delete callback function to be exectuted on all keys associated with MPI_COMM_FREE. If no key has been attached to MPI_COMM_SELF, then no callback is invoked. To allow for clean layering of callbacks, MPI will call the callback functions in the reverse order to which the keys were created. The “freeing” of MPI_COMM_SELF occurs before any other parts of MPI are effected. Thus, for example, calling MPI_FINALIZED will return false in any of these callback functions. Once done with MPI_COMM_SELF, the order and rest of the actions taken by MPI_FINALIZE is not specified.

Advice to implementors. Since attributes can be added from any supported language, the MPI implementation needs to remember the creating language so the correct callback is made. (*End of advice to implementors.*)

Callback functions on communicators are always called in the reverse order to which they were added to the communicator.

3.2.2 Clarification of status after MPI_WAIT and MPI_TEST

Current Status: Passed once.

A status returned by a call to MPI_WAIT, MPI_TEST, or any of the other derived functions, where the request corresponds to a send call, is undefined, with two exceptions: The error status field will contain valid information if the wait or test call returned with MPI_ERROR_IN_STATUS; and the returned status can be queried by the call MPI_TEST_CANCELLED.

3.2.3 Clarification of MPI_INTERCOMM_CREATE

Current Status: Passed twice.

The Problem: The MPI 1.1 standard says, in the discussion of MPI_INTERCOMM_CREATE, both that

> The groups must be disjoint

and that

> The leaders may be the same process.

To further muddy the waters, the reason given for "The groups must be disjoint" is based on concerns about the implmentation of MPI_INTERCOMM_CREATE that are not applicable for the case where the leaders are the same process.

The Fix: Delete the text:

> (the two leaders could be the same process)

from the discussion of MPI_INTERCOMM_CREATE.

Replace the text:

> All inter-communicator constructors are blocking and require that the local and remote groups be disjoint in order to avoid deadlock.

with

> All inter-communicator constructors are blocking and require that the local and remote groups be disjoint.
>
> *Advice to users.* The groups must be disjoint for several reasons. Primarily, this is the intent of the intercommunicators – to provide a communicator for communication between disjoint groups. This is reflected in the definition of MPI_INTERCOMM_MERGE, which allows the user to control the ranking of the processes in the created intracommunicator; this ranking makes little sense if the groups are not disjoint. In addition, the natural extension of collective operations to intercommunicators (being considered for MPI-2) makes the most sense when the groups are disjoint. It is the intent of the MPI definition not to preclude future extensions. (*End of advice to users.*)

3.2.4 Clarification of MPI_INTERCOMM_MERGE

Current Status: Passed once.

The error handler on the new intercommuncator in each process is inherited from the communicator that contriubes the local group. Note that this can result in different processes in the same communicator having different error handlers.

3.2.5 Clarification of Binding of MPI_Type_size

Current Status: Passed twice.

This clarification is needed in the MPI-1 description of MPI_Type_size, since the issue repeatedly arises. It is a clarification of the binding.

> *Advice to users.* The binding of MPI_Type_size, specifically, whether the output argument in C is of type `int` as in the current standard or should be changed to something else, has been extensively discussed by the MPI Forum. The final decision is to leave it an `int`. (*End of advice to users.*)

3.2.6 Clarification of MPI_REDUCE

Current Status: Passed twice.

The current text on p. 115 lines 25-28 from MPI-1.1 (June 12, 1995) says:

The datatype argument of MPI_REDUCE must be compatible with op. Predefined operators work only with the MPI types listed in Sec. 4.9.2 and Sec. 4.9.3. User-defined operators may operate on general, derived datatypes.

This text is changed to:

The datatype argument of MPI_REDUCE must be compatible with op. Predefined operators work only with the MPI types listed in Sec. 4.9.2 and Sec. 4.9.3. Furthermore, the datatype and op given for predefined operators must be the same on all processes.

Note that it is possible for users to supply different user-defined operations to MPI_REDUCE in each process. MPI does not define which operations are used in this case on which operands.

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

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

3.2.7 Minor Corrections

Current Status: Passed twice.

The following corrections to MPI-1.1 are (all page and line numbers are for the June 12, 1995 version without changebars):

- Page 11, line 36 reads
  MPI_ADDRESS
  but should read
  MPI_ADDRESS_TYPE

  Discussion:

  This is clearly a mistake since p. 69, line 33 defines a function with the same name. The only question is if everyone agrees this is the correct new name.

- Page 19, lines 1-2 reads
  for (64 bit) C integers declared to be of type longlong int
  but should read
  for (64 bit) C integers declared to be of type long long int

  Discussion:

  Just to make it correct C so it is long long.

- Page 40, line 48 should have the following text added:

  Advice to users. To prevent troubles with the register optimization of the Fortran compilers please pay attention to the hints in section 2.5.2 "A Fortran Problem with register optimization" on page 10. (*End of advice to users.*)

  Discussion:

  This is to point out the new warning given in MPI-1.2. The section and page are in the MPI-2 document.

- Page 41, lines 16-18 reads
  A **empty** status is a status which is set to return tag = MPI_ANY_TAG, source = MPI_ANY_SOURCE, and is also internally configured so that calls to MPI_GET_COUNT and MPI_GET_ELEMENTS return count = 0.
  but should read
  A **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 and MPI_GET_ELEMENTS return count = 0 and MPI_TEST_CANCELLED returns false.

  Discussion:

Filling in the forgotten error fields.

- Page 59, line 3 should have the following text added:

  Advice to users. To prevent troubles with the register optimization of the Fortran compilers please pay attention to the hints in section 2.5.2 "A Fortran Problem with register optimization" on page 10. (*End of advice to users.*)

  Discussion:

  This is to point out the new warning given in MPI-1.2. The section and page are in the MPI-2 document.

- Page 70, line 16 should have the following text added:

  Advice to users. To prevent troubles with the register optimization of the Fortran compilers please pay attention to the hints in section 2.5.2 "A Fortran Problem with register optimization" on page 10. (*End of advice to users.*)

  Discussion:

  This is to point out the new warning given in MPI-1.2. The section and page are in the MPI-2 document.

- Page 71, line 10 reads
  and do not affect the the content of a message
  but should read
  and do not affect the content of a message

  Discussion:

  This is a trivial typo of having the 2 times.

- Page 74, lines 39-45 read
  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.)

  A datatype may specify overlapping entries. If such a datatype is used in a receive operation, that is, if some part of the receive buffer is written more than once by the receive operation, then the call is erroneous.

  The first part was an MPI-1.1 addition. The second part overlaps with it. The old text will be removed so it now reads
  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.)

Discussion:

The intent is to remove duplicate text which seems to have been left. No change in meaning is intended.

- Page 90, line 3 reads
  MPI_Pack_size(count, MPI_CHAR, &k2);
  but should read
  MPI_Pack_size(count, MPI_CHAR, comm, &k2);

Discussion:

This is a minor fix to an example for a missing argument.

- Page 90, line 10 reads
  MPI_Pack(chr, count, MPI_CHAR, &lbuf, k, &position, comm);
  but should read
  MPI_Pack(chr, count, MPI_CHAR, lbuf, k, &position, comm);

Discussion:

This is a minor fix to an example for an argument that is declared to be a pointer and then passed with &.

- Page 109, lines 26/27 and page 110, lines 28/29 reads
  The `j`th block of data sent from each process is received by every process and placed in the `j`th block of the buffer recvbuf.
  but should read
  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.

Discussion:

This is related to a subtle point in usage. The sendbuf is a single item whereas the recvbuf is an "array" of items.

- Page 117, lines 22/23 reads
  MPI provides seven such predefined datatypes.
  but should read
  MPI provides nine such predefined datatypes.

Discussion:

This is an obvious mistake. There are nine listed on line 28-40.

- Page 122, lines 35-36 read
  MPI_OP_FREE(op)

IN	op	operation (handle)

  but should read
  MPI_OP_FREE(op)

INOUT	op	operation (handle)

Discussion:

This appears to have been a mistake. This function frees the function pointer and sets its value to MPI_OP_NULL. Thus, it is both input and output as are the free routines for Groups, Comms, etc.

- Page 125, line 1 reads
  CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, 0, comm, ierr)
  but should read
  CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, comm, ierr)

Discussion:

The extra argument appears to be left over from the previous example with MPI_REDUCE. This is a mistake.

- Page 194, lines 30 and 31 reads
  to the greatest possible, extent,
  but should read
  to the greatest possible extent,

Discussion:

Very minor typo.

- Page 194, line 48 reads
  MPI_ERRHANDLER_CREATE(FUNCTION, HANDLER, IERROR)
  but should read

MPI_ERRHANDLER_CREATE(FUNCTION, ERRHANDLER, IERROR)

Discussion:

This is a small typo to match all the other names for this argument.

- Page 196, lines 1-2 reads
  MPI_ERRHANDLER_FREE(errhandler)

IN	errhandler	MPI error handler (handle)

but should read
MPI_ERRHANDLER_FREE(errhandler)

INOUT	errhandler	MPI error handler (handle)

Discussion:

This appears to have been a mistake. This function frees the function pointer and sets its value to MPI_ERRHANDLER_NULL. Thus, it is both input and output as are the free routines for Groups, Comms, etc.

- Page 202, line 48 reads
  `int MPI_Pcontrol(const int level, ...)`

  but should read
  `int MPI_Pcontrol(int level, ...)`

Discussion:

This is the only function with const and the ramifications are not liked.

- Page 203, line 1 reads
  MPI_PCONTROL(level)
  but should read
  MPI_PCONTROL(LEVEL)

Discussion:

This is a minor typo.

- Page 210, line 44 reads
  MPI_PENDING
  but should read
  MPI_ERR_PENDING

  Discussion:

  This is a typo. See page 197, line 20 for the original definition.

- Page 211, line 44 reads
  MPI_DOUBLE_COMPLEX
  but should be moved to Page 212, line 22 since it is an optional Fortran datatype.

  Discussion:

  On page 19, lines 2-3, it clearly states this is an optional Fortran datatype. It is listed incorrectly in the appendix.

- Page 212, add new lines of text at line 22 and line 25 to read:
  etc.
  Thus, the text will now read:

```
/* optional datatypes (Fortran) */
MPI_INTEGER1
MPI_INTEGER2
MPI_INTEGER4
MPI_REAL2
MPI_REAL4
MPI_REAL8
etc.

/* optional datatypes (C) */
MPI_LONG_LONG_INT
etc.
```

  Discussion:

  On page 19, line 6, concludes the list of optional datatypes with etc. It was not intended to be a complete list of all possible ones. Some have interpreted the appendix to be a complete list since only the ones explictly on page 19 are listed. Adding etc., to the Fortran and C list in the appendix will make it clear these are only some of the possible optional datatypes.

- Page 213, line 28. The following text should be added:
  /* Predefined functions in C and Fortran */ MPI_NULL_COPY_FN MPI_NULL_DELETE_FN MPI_DUP_FN

  Discussion:

These are missing and defined on pages 169, line 17 and p. 170, line 4.

- Page 220, lines 19-20 reads
  int double MPI_Wtime(void)
  int double MPI_Wtick(void)
  but should read
  double MPI_Wtime(void)
  double MPI_Wtick(void)

Discussion:

This was a mistake caused by the macros.

- Page 220, line 20 reads
  `int MPI_Pcontrol(const int level, ...)`

  but should read
  `int MPI_Pcontrol(int level, ...)`

Discussion:

This is the same change as p. 202, line 48

- Page 222, line 34 reads
  INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
  but should read
  INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

Discussion:

An extra argument got added. It needs to be removed.

- Page 222, line 38 reads
  INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
  but should read
  INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

Discussion:

An extra argument got added. It needs to be removed.

- Page 227, lines 19-20 reads
  MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, INTRACOMM, IERROR)
  INTEGER INTERCOMM, INTRACOMM, IERROR
  but should read

MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR)
INTEGER INTERCOMM, NEWINTRACOMM, IERROR

Discussion:

This makes it match the definition in the text on Page 159, lines 34-36. It only changes the name of the argument.

- Page 228, line 46 reads
  MPI_ERRHANDLER_CREATE(FUNCTION, HANDLER, IERROR)
  but should read
  MPI_ERRHANDLER_CREATE(FUNCTION, ERRHANDLER, IERROR)

Discussion:

This change makes the appendix consistant with change on p. 194, line 48.

- Page 229, line 33 reads
  MPI_PCONTROL(level)
  but should read
  MPI_PCONTROL(LEVEL)

Discussion:

This is to make the language binding the same as the change to Page 203, line 1.

Part II

MPI-2

Chapter 4

Process Creation and Management

4.1 Introduction

Current Status: No votes

MPI-1 provides an interface that allows processes in a parallel program to communicate with one another. MPI-1 specifies neither how the processes are created, nor how they establish communication. Moreover, an MPI-1 application is static, that is, no processes can be added to or deleted from an application after it has been started.

MPI users have asked that the MPI-1 model be relaxed to allow process creation and management after an MPI application has been started. A main impetus comes from the PVM [9] research effort, which has provided a wealth of experience with process management and resource control that illustrates their benefits and potential pitfalls.

The MPI Forum decided not to address resource control in MPI-2 because it was not able to design a portable interface that would be appropriate for the wide range of existing and potential resource and process controllers. Resource control can encompass a wide range of abilities, including adding and deleting nodes from a virtual parallel machine, reserving and scheduling resources, managing compute partitions of an MPP, and returning information about available resources.

MPI-2 assumes that resource control is provided exernally – probably by computer vendors, in the case of tightly coupled systems, or by a third party software package when the environment is a cluster of workstations.

The reasons for adding process management to MPI are both technical and practical. Important classes of message passing applications require process control. These include task farms, serial applications with parallel modules, and problems that require a run-time assessment of the number and type of processes that should be started. On the practical side, users of workstation clusters who are migrating from PVM to MPI may be accustomed to using PVM's capabilities for process and resource management. The lack of these features is a practical stumbling block to migration.

While process management is essential, adding it to MPI should not compromise the portability or performance of MPI applications. In particular:

- The MPI-2 process model must apply to the vast majority of current parallel environments. These include everything from tightly integrated MPPs such as the Intel Paragon and the Meiko CS-2 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 continue to guarantee communication determinism, i.e., process management must not introduce unavoidable race conditions.
- MPI must not contain features that compromise performance.
- MPI-1 programs must work under MPI-2, i.e., the MPI-1 static process model must be a special case of the MPI-2 dynamic model.

The MPI process management model addresses these issues in two ways. First, MPI remains primarily a communication library. It does not manage the parallel environment in which a parallel program executes, though it provides a minimal interface between an application and external resource and process managers.

Second, MPI-2 does not change the concept of communicator. Once a communicator is built, it behaves as specified in MPI-1. A communicator is never changed once created, and it is always created using deterministic collective semantics.

4.2 The MPI-2 Process Model

Current Status: No votes.

The MPI-2 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.

4.2.1 Starting Processes

MPI applications may start new processes through an interface to an external process manager, which can range from a parallel operating system (CMOST) to layered software (POE) to an `rsh` command (p4).

MPI_SPAWN starts MPI processes and establishes communication with them, returning an intercommunicator. MPI_SPAWN_MULTIPLE starts several different binaries (or the same binary with different arguments), placing them in the same MPI_COMM_WORLD and returning an intercommunicator.

MPI uses the existing group abstraction to represent processes. A process is identified by a (group, rank) pair.

4.2.2 The Runtime Environment

The MPI_SPAWN and MPI_SPAWN_MULTIPLE routines provide an interface between MPI and the *runtime environment* of an MPI application. The difficulty is that there is an enormous range of runtime environments and application requirements, and MPI must not be tailored to any particular one. Examples of such environments are:

- **MPP managed by a batch queueing system**. Batch queueing systems generally allocate resources before an application begins, enforce limits on resource use (CPU time, memory use, etc), and do not allow a change in resource allocation after a job begins. Moreover, many MPPs have special limitations or extensions, such as a limit 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 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.

Advice to implementors. A virtual machine abstraction is appropriate in most environments and should be part of a "high-quality" implementation. In other cases, providing the virtual machine abstraction may add too much baggage (e.g., daemon processes) or inhibit portability. (*End of advice to implementors.*)

Interaction between MPI and the runtime environment is limited to the following areas:

- A process may start new processes with MPI_SPAWN, MPI_SPAWN_MULTIPLE.
- When a process spawns a child process, it optionally passes extra information to the runtime environment to indicate where or how the process should be started. This extra information may be opaque to MPI.
- An attribute variable MPI_UNIVERSE_SIZE, tells a program how "large" the initial runtime environment is, namely how many processes can usefully be started alltogether. 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.

4.3 Process Manager Interface

Current Status: 2 votes on entire section.

4.3.1 Processes in MPI

A process is represented in MPI by a (group, rank) pair. A process may or may not be an "MPI process" in that it may or may not call MPI_INIT and thus be assigned an MPI_COMM_WORLD. An MPI process cannot communicate directly with another process unless it has a communicator. Note that a (group, rank) identification is not unique because a process may belong to several groups.

4.3.2 Starting Processes and Establishing Communication

The following routine starts a number of MPI processes and establishes communication with them, returning an intercommunicator.

Advice to users. It is possible in MPI to start a static SPMD or MPMD application by starting first one process and having that process start its siblings with MPI_SPAWN. This practice is discouraged primarily for reasons of performance. If possible, it is preferable to start all processes at once, as a single MPI-1 application. (*End of advice to users.*)

MPI_SPAWN(command, argv, maxprocs, info, root, comm, intercomm, array_of_errcodes)

IN	command	name of program to be spawned (string, significant only at root)
IN	argv	arguments to command (array of strings, significant only at root)
IN	maxprocs	maximum number of processes to start (integer, significant only at root)
IN	info	a set of key-value pairs telling the runtime system where and how to start the processes (handle, significant only at root)
IN	root	rank of process in which previous arguments are examined (integer)
IN	comm	communicator containing group of spawning processes (handle)
OUT	intercomm	intercommunicator between original group and the newly spawned group (handle)
OUT	array_of_errcodes	one code per process (array of integer)

```
int MPI_Spawn(char command[], char* argv[], int maxprocs, MPI_Info info,
             int root, MPI_Comm comm, MPI_Comm* intercomm,
             int array_of_errcodes[])
```

```
MPI_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,
             ARRAY_OF_ERRCODES, IERR)
    CHARACTER*(*) COMMAND, ARGV(*)
    INTEGER INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*),
    IERR
```

MPI_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 parent. MPI_SPAWN is collective over comm, and 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_SPAWN. In this sense, MPI_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_SPAWN contains the parent processes in the local group and the child processes in the remote. This intercommunicator is known in the children as MPI_COMM_PARENT.

> *Advice to users.* An implementation may automatically establish communication before MPI_INIT is called. Thus, completion of MPI_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 will be implementation-dependent and will be appropriate for the runtime environment.

Advice to implementors. The implementation should use a "natural" rule for finding executables and determining working directories. For instance, a homogeneous system with a global file system might look first in the working directory of the spawning process, or might search the directories in a PATH environment variable as do Unix shells. An implementation on top of PVM would use PVM's rules for finding executables (usually in ~/pvm3/bin/$PVM_ARCH). An MPI implementation running under POE on an IBM SP would use POE's method of finding executables. A high-quality implementation will give the user some control over how executables are found and working directories are determined. (*End of advice to implementors.*)

If the program named in command does not call MPI_INIT, but instead forks a process that calls MPI_INIT, the results are undefined. Some implementations may allow this case to work.

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. Some implementations may allow you to do this, but may have restrictions, such as requiring that arguments supplied to the shell script must be supplied to the program, or requiring that certain parts of the environment not be changed. (*End of advice to users.*)

The argv argument argv is an array of strings containing arguments that are passed to the program. The first element of argv is the first argument passed to command, not, as is conventional in some contexts, the command itself. The argument list is terminated by NULL in C and an empty string in Fortran. In Fortran, leading and trailing spaces are always stripped, so that a string consisting of all spaces is considered an empty string. The constant MPI_ARGV_NULL may be used in both C and Fortran to indicate an empty argument list. In C, this constant is the same as NULL.

Example 4.1 Examples of argv in C and Fortran
To run the program "ocean" with arguments "-gridfile" and "ocean1.grd" in C:

```
char command[] = "ocean";
char *argv[] = {"-gridfile", "ocean1.grd", NULL};
MPI_Spawn(command, argv, ...);
```

or, if not everything is known at compile time:

```
char *command;
char **argv;
command = "ocean";
argv=(char **)malloc(3 * sizeof(char *));
argv[0] = "-gridfile";
argv[1] = "ocean1.grd";
argv[2] = NULL;
MPI_Spawn(command, argv, ...);
```

In Fortran:

```
CHARACTER*25 command, argv(3)
command = ' ocean '
argv(1) = ' -gridfile '
argv(2) = ' ocean1.grd'
argv(3) = ' '
call MPI_SPAWN(command, argv, ...)
```

Arguments are supplied to the program if this is allowed by the operating system. In C, the MPI_SPAWN argument argv is not exactly the same as the argv argument of main, but is shifted by one element. Specifically, argv[0] of main is provided by by the implementation and conventionally would contain the name of the program (command). argv[1] of main corresponds to argv[0] in MPI_SPAWN, etc. Passing an argv of MPI_ARGV_NULL to MPI_SPAWN results in main receiving argc of 1 and an argv whose element 0 is (conventionally) the name of the program.

If a Fortran implementation supplies routines that allow a program to obtain its arguments, the arguments may be available through that mechanism. In C, if the operating system does not support arguments appearing in argv and if the program is an MPI program, the MPI implementation may add the arguments to the argv that is passed to MPI_INIT. Note that the argv argument to MPI_SPAWN contains only arguments, whereas argv[0] in main conventionally contains the name of the program.

The maxprocs argument MPI tries to spawn maxprocs processes. If it is unable to spawn maxprocs processes, it returns an error of class MPI_ERR_SPAWN and a half-empty intercommunicator. A half-empty intercommunicator is one for which MPI_COMM_REMOTE_GROUP returns MPI_GROUP_EMPTY and MPI_COMM_REMOTE_SIZE returns zero. The reasons why MPI_SPAWN was not able to spawn specific processes are indicated in array_of_errcodes as described below.

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. In principle, the info argument may specify an arbitrary set $\{m_i : 0 \leq m_i \leq \mathsf{maxprocs}\}$ 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, no error is returned, and the number of spawned processes, m, is given by the size of the remote group. In this case, if the number of spawned processes is smaller than maxproc, reasons why the other processes were not spawned are given in array_of_errcodes. If it is not possible to spawn one of the allowed numbers of processes, no processes are spawned and MPI_SPAWN returns an error of class MPI_ERR_SPAWN and a half-empty intercommunicator, indicating specific reasons in array_of_errcodes.

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 4.3.5 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 integer in Fortran. It is a container for a number of user-specified (key,value) pairs. key and value are strings (null-terminated `char*` in C, `character*(*)` in Fortran). Routines to create and manipulate the info argument are described in section 4.5.1.

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 4.3.5). 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_SPAWN could be empty. The ability to do this follows from the fact that MPI does not specify how an executable is found, and the info argument can tell the runtime system where to "find" the executable "" (empty string). Of course a program that does this will not be portable across MPI implementations.

The root argument All arguments before the root argument are examined only on the process whose rank in comm is equal to root. The value of these arguments on other processes is ignored.

The array_of_errcodes argument The array_of_errcodes is an array of length maxprocs in which MPI reports the status of each process that MPI was requested to start. If all maxprocs processes were spawned, array_of_errcodes is filled in with the value MPI_SUCCESS. If only m ($0 \leq m <$ maxprocs) processes are spawned, m of the entries will contain MPI_SUCCESS and the rest will contain an implementation-specific error code indicating the reason MPI could not start the process. MPI does not specify which entries correspond to failed processes. An implementation may, for instance, fill in error codes in one-to-one correspondence with a detailed specification in the info argument. These error codes all belong to the error class MPI_ERR_SPAWN if there was no error in the argument list. An application may pass MPI_ERRCODES_IGNORE in C or Fortran.

MPI_COMM_PARENT MPI provides a predefined communicator MPI_COMM_PARENT. If a process was started with MPI_SPAWN or MPI_SPAWN_MULTIPLE, MPI_COMM_PARENT is the other side of the intercommunicator returned by those calls. Otherwise, MPI_COMM_PARENT is an intercommunicator with a remote group size of 0.

4.3.3 Starting Multiple Executables and Establishing Communication

While MPI_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_SPAWN_MULTIPLE(count, array_of_commands, array_of_argv, array_of_maxprocs, array_of_info, root, comm, intercomm, array_of_errcodes)

IN	count	Number of command lines (integer, size of each of the following arrays)
IN	array_of_commands	programs to be executed (array of strings, significant only at root)
IN	array_of_argv	arguments for commands (array of array of strings, significant only at root)
IN	array_of_maxprocs	maximum number of processes to start for each command line (array of integer, significant only at root)
IN	array_of_info	info objects telling the runtime system where and how to start processes (array of handles, significant only at root
IN	root	rank of process in which previous arguments are examined (integer)
IN	comm	communicator containing group of spawning processes (handle)
OUT	intercomm	intercommunicator between original group and newly spawned group (handle)
OUT	array_of_errcodes	one error code per process (array of integer)

```
int MPI_Spawn_multiple(int count, char* array_of_commands[],
             char** array_of_argv[], int array_of_maxprocs[],
             MPI_Info array_of_info[], int root, MPI_Comm comm,
             MPI_Comm* intercomm, int array_of_errcodes[])
```

```
MPI_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
             ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,
             ARRAY_OF_ERRCODES, IERR)
    INTEGER COUNT, ARRAY_OF_INFO(*), ARRAY_OF_MAXPROCS(*), ROOT, COMM,
    INTERCOMM, ARRAY_OF_ERRCODES(*), IERR
    CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)
```

MPI_SPAWN_MULTIPLE is identical to MPI_SPAWN except that there are multiple executable specifications. The first argument, count, gives the number of specifications. Each of the next four arguments are simply arrays of the corresponding arguments in MPI_SPAWN. For the Fortran version of array_of_argv, the element array_of_argv(i,j) is the jth argument to command number i. This may seem backwards to Fortran programmers who are familiar with Fortran's column-major ordering. However, it is necessary to do it this way to allow MPI_SPAWN to sort out arguments. Note that the leading dimension of array_of_argv *must* be the same as count.

In any language, an application may use the constant MPI_ARGVS_NULL (which is (char ***)0 in C) to specify that no arguments should be passed to any commands. The effect of setting individual elements of array_of_argv to MPI_ARGV_NULL is not defined. To specify arguments for some commands but not others, the commands without arguments should have a corresponding argv whose first elemement is null ((char *)0 in C and empty string in Fortran).

All of the spawned processes have the same MPI_COMM_WORLD. Their ranks in MPI_COMM_WORLD correspond directly to the order in which the command lines are specified in MPI_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 line 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_SPAWN multiple times would create many sets of children with different MPI_COMM_WORLDs whereas MPI_SPAWN_MULTIPLE creates children with a single MPI_COMM_WORLD, so the two methods are not completely equivalent. There are also two performance-related reasons why, if you need to spawn multiple executables, you may want to use MPI_SPAWN_MULTIPLE instead of calling MPI_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 1-dimensional array of size $\sum_{i=1}^{count} n_i$, where n_i is the ith element of array_of_maxprocs. Command line 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_SPAWN.

Example 4.2 Examples of array_of_argv in C and Fortran
To run the program "ocean" with arguments "-gridfile" and "ocean1.grd" and the program "atmos" with argument "atmos.grd" in C:

```
char *array_of_command[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_Spawn(array_of_commands, array_of_argv, ...);
```

Here's how you do it in Fortran:

```
CHARACTER*25 commands(2), array_of_argv(2, 3)
commands(1) = ' ocean '
array_of_argv(1, 1) = ' -gridfile '
array_of_argv(1, 2) = ' ocean1.grd'
array_of_argv(1, 3) = ' '

command(2) = ' atmos '
array_of_argv(2, 1) = ' atmos.grd '
array_of_argv(2, 2) = ' '

call MPI_SPAWN(commands, array_of_argv, ...)
```

4.3.4 Nonblocking requests

Spawning new processes may be an expensive operation. In order to allow processes to do useful work while processes are being spawned, MPI provides non-blocking versions of the each of the routines defined above.

MPI_ISPAWN(command, argv, maxprocs, info, root, comm, intercomm, array_of_errcodes, request)

IN	command	name of program to be spawned (string, significant only at root)
IN	argv	arguments to command (array of strings, significant only at root)
IN	maxprocs	maximum number of processes to start (integer, significant only at root)
IN	info	a set of key-value pairs telling the runtime system where and how to start the processes (handle, significant only at root)
IN	root	rank of process in which previous arguments are examined (integer)
IN	comm	communicator containing group of spawning processes (handle)
OUT	intercomm	intercommunicator between original group and the newly spawned group (handle)
OUT	array_of_errcodes	one code per process (array of integer)
OUT	request	request object (handle)

```
int MPI_Ispawn(char command[], char* argv[], int maxprocs, MPI_Info info,
             int root, MPI_Comm comm, MPI_Comm* intercomm,
             int array_of_errcodes[], MPI_Request* request)
```

```
MPI_ISPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,
             ARRAY_OF_ERRCODES, REQUEST, IERR)
    CHARACTER*(*) COMMAND, ARGV(*)
    INTEGER INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*),
    REQUEST, IERR
```

MPI_ISPAWN_MULTIPLE(count, array_of_commands, array_of_argv, array_of_maxprocs, array_of_info, root, comm, intercomm, array_of_errcodes, request)

IN	count	Number of command lines (integer, size of each of the following arrays)
IN	array_of_commands	programs to be executed (array of strings, significant only at root)
IN	array_of_argv	arguments for commands (array of array of strings, significant only at root)
IN	array_of_maxprocs	maximum number of processes to start for each command line (array of integer, significant only at root)
IN	array_of_info	info objects telling the runtime system where and how to start processes (array of handles, significant only at root
IN	root	rank of process in which previous arguments are examined (integer)
IN	comm	communicator containing group of spawning processes (handle)
OUT	intercomm	intercommunicator between original group and newly spawned group (handle)
OUT	array_of_errcodes	one error code per process (array of integer)
OUT	request	request object (handle)

```
int MPI_Ispawn_multiple(int count, char* array_of_commands[],
             char** array_of_argv[], int array_of_maxprocs[],
             MPI_Info array_of_info[], int root, MPI_Comm comm,
             MPI_Comm* intercomm, int array_of_errcodes[],
             MPI_Request *request)
```

```
MPI_ISPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
             ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,
             ARRAY_OF_ERRCODES, REQUEST, IERR)
    INTEGER COUNT, ARRAY_OF_INFO(*), ARRAY_OF_MAXPROCS(*), ROOT, COMM,
    INTERCOMM, ARRAY_OF_ERRCODES(*), REQUEST, IERR
    CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)
```

For all of the routines in this section, the arguments are identical to those of their blocking counterparts except that there is an additional request argument. The request can be used as input for MPI_WAIT, MPI_TEST, etc. In all cases, the output group or intercomm, and the array_of_errcodes are updated asynchronously and are not valid until the request completes.

The status object returned by MPI_WAIT contains an error code indicating whether or not the spawn was successful. The failure or success of individual processes is indicated in array_of_errcodes. The errorcode can be retrieved as the MPI_ERROR element of the status object returned by MPI_WAIT, MPI_TEST, etc. Success is indicated by a value of

MPI_SUCCESS for the MPI_ERROR field. All other values indicate an error. The values of other fields in the status object are not defined.

MPI_ISPAWN and MPI_ISPAWN_MULTIPLE are collective over processes in comm and the newly created processes. They obey the the following rules:

- A non-blocking collective spawn routine must complete in every process of the associated communicator before MPI_FINALIZE is called.
- The operation is local. A non-blocking collective operation will return even if it is not called immediately on all processes.
- Multiple outstanding non-blocking operations of the same type on the same communicator are allowed, and match up in the order they were called, since there is no tag argument. Operations of different types and/or on different communicators may be interleaved.
- A single collective operation may be blocking on some nodes and nonblocking on others.
- Spawn requests cannot be cancelled. An attempt to cancel a collective request will return an error of error class MPI_ERR_REQUEST.

Advice to implementors. In some cases it may not be possible to fulfill a spawn request immediately, but may be possible to fulfill the request later after waiting for resources to become available. It is left to the implementation to decide how long to wait before giving up. An implementation should implement whatever is appropriate for an environment, possibly providing a mechanism (through the info argument) for an application to modify this behavior. In certain cases it might be reasonable to have different policy defaults for MPI_SPAWN and MPI_ISPAWN. (*End of advice to implementors.*)

4.3.5 Reserved keys

The following key values are reserved for all info arguments in section 4.3. An implementation is not required to interpret these key values, but if it does interpret the key value, it must provide the functionality described.

Advice to implementors. Implementation-specific key names should be chosen so that they are unlikely to conflict with keys chosen by other implementors. A high-quality implementation will ensure this by adding an implementation-specific prefix to key names. (*End of advice to implementors.*)

host Value is a hostname, format of hostname determined by implementation

arch Value is an architecture name. Valid architecture names and what they mean are determined by an implementation

wdir Value is the name of a directory on a machine on which the spawned process(es) execute(s). The format of the directory name is determined by the implementation.

path Value is a directory or set of directories on remote machine where the implementation should look for the executable. The format of path is determined by the implementation.

file Value is the name of a file in which additional information is specified. The format of the filename and internal format of the file are determined by the implementation.

soft Value specifies a set of numbers which are allowed values for the number of processes that MPI_SPAWN (et al.) may create. The format of value is a comma-separated list of Fortran-90 triplets. Negative values and values greater than maxprocs are ignored. MPI will spawn the lagest number of processes it can, consistent with some number in the set. The order in which triplets are given is not significant.

By Fortran-90 triplets, we mean:

1. `a` means a
2. `a:b` means $a, a+1, a+2, \ldots, b$
3. `a:b:c` means $a, a+c, a+2c, \ldots, b$. If $b < a$ then c must be negative.

Examples:

1. `a:b` gives a range between a and b
2. `0:N` gives full "soft" functionality
3. `1,2,4,8,16,32,64,128,256,512,1024,2048,4096` allows power-of-two number of processes.
4. `2:10000:2` allows even number of processes.
5. `2:10:2,7` allows 2, 4, 6, 7, 8, or 10 processes.

4.3.6 Spawn Examples

Manager-worker example, using SPAWN.

```
/* manager */
#include <mpi.h>
main(int, argc, char *arg[])
{
   MPI_Status status;
   int world_size, universe_size, flag;
   MPI_Comm everyone;           /* intercommunicator */


   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &world_size);

   if (world_size != 1)        error("Top heavy with management");

   MPI_Attr_get(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE, &universe_size, &flag);
   if (!flag) {
        printf("This MPI does not support UNIVERSE_SIZE. How many\n"
```

```
                "processes total?")
          scanf("%d", &universe_size);
    }
    if (universe_size == 1) error("No room to start workers");

    MPI_Spawn("worker", MPI_ARGV_NULL, universe_size-1,
              MPI_INFO_NULL, 0, MPI_COMM_SELF, &everyone,
              MPI_ERRCODES_IGNORE);
    /*
     * Parallel code here. The communicator "everyone" can be used
     * to communicate with the spawned processes, which have ranks 0,..
     * MPI_UNIVERSE_SIZE-1 in the remote group of the intercommunicator
     * "everyone".
     */

    MPI_Finalize();
}

/* worker */

#include <mpi.h>
main(int argc, char *argv[])
{
    int size;
    MPI_Init(&argc, &argv);
    MPI_Comm_remote_size(MPI_COMM_PARENT, &size);
    if (size != 1) error("Something's wrong with the parent");

    /*
     * Parallel code here.
     * The manager is represented as the process with rank 0 in (the remote
     * group of) MPI_COMM_PARENT.  If the workers need to communicate among
     * themselves, they can either extract the local group of
     * MPI_COMM_PARENT, or use MPI_COMM_WORLD.  These two
     * should be the same.
     */

    MPI_Finalize();
}
```

SPMD code that starts itself

This is how many PVM programs are typically written and illustrates how such programs could be converted to MPI-2. Note that the first instance must be started in such a way that MPI_COMM_PARENT is half-empty.

```
/* myprog */

#include <mpi.h>
main(int argc, char *argv[])
{
   MPI_Comm everyone;          /* intercommunicator */
   MPI_Comm new_world;         /* corresponding intracommunicator */
   int size, nslots, flag;
   MPI_Init(&argc, &argv);

   MPI_Comm_remote_size(MPI_COMM_PARENT, &size);
   MPI_Attr_get(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE, &nslots, &flag);
   if (size == 0)  /* I'm the parent */
   {
      MPI_Spawn("myproc", MPI_ARGV_NULL, nslots-1,
                MPI_INFO_NULL, 0, MPI_COMM_SELF, &everyone, NULL);
      MPI_Intercomm_merge(everyone, FALSE, &new_world);
   }
   else {
      MPI_Intercomm_merge(MPI_COMM_PARENT, TRUE, &new_world);
   }
    /* SPMD parallel code here, using new_world instead of MPI_COMM_WORLD */

    MPI_Finalize();
}
```

4.4 Establishing Communication

Current Status: Two votes on entire section.

This section provides the functions needed to solve the general case of establishing communication between two sets of MPI processes that do not share an existing communicator. This situation arises in several contexts:

1. Two parts of an application that are started independently need to communicate.
2. A visualization tool wants to attach to a running process.
3. A server may want to accept connections from multiple clients. This situation can be more complex than traditional client-server programs. In particular,
   - Both client and server may be parallel programs.
   - Since this is an application rather than a system service, there may be multiple instances of the server running on the same host.

In each of these situations, neither set of processes spawns the other, but they wish to establish communication. As in the case of MPI_SPAWN, we will use MPI intercommunicators to express this situation. That is, we will define MPI functions that support creation of an intercommunicator in which the local and remote groups are the original sets of processes.

Establishing contact between two groups of processes that do not share an existing communicator is a collective, but asymmetric process. One group of processes indicates its willingness to accept connections from other groups of processes. We will call this group the (parallel) *server*, even if this is not a client/server type of application. The other group connects to the server; we will call it the *client*.

Advice to users. While the names *client* and *server* are used throughout this section, MPI does not provide the traditional robustness of client server systems. The functionality described in this section is intended to allow two cooperating parts of the same application to communicate with one another. For instance, a client that gets a segmentation fault and dies, or one that doesn't participate in a collective operation may cause a server to crash or hang. (*End of advice to users.*)

4.4.1 Names, addresses, ports, and all that

Almost all of the complexity in MPI client/server routines addresses the question "how does the client find out how to contact the server?" The difficulty, of course, is that there is no existing communication channel between them, yet they must somehow agree on a rendezvous point where they will establish communication - Catch 22.

Agreeing on a rendezvous point always involves a third party. The third party may be, for instance, a higher level of management, which tells them both, "thou shalt meet at point X at high noon." The third party might be a go-between, that gets a rendezvous point from the server and passes it to the client. Complicating matters might be the fact that the client doesn't really care what server it contacts, only that it be able to get in touch with one that can handle its request.

Ideally, MPI can accomodate 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_PORT_OPEN routine. It accepts a connection to a given port with MPI_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 a *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_NAME_PUBLISH and the client to retrieve the port name from the service name with MPI_NAME_GET. This allows three levels of portability, with increasing levels of functionality.

1. Applications that do not rely on the ability to publish names are the most portable. Typically the port_name must be transferred "by hand" from server to client.
2. Applications that use the MPI_NAME_PUBLISH mechanism are completely portable among implementations that provide this service. To be portable among all implementations, these applications should have a fall-back mechanism that can be used when names are not published.
3. Applications may ignore MPI's name publishing functionality and use their own mechanism (possibly system-supplied) to publish names. This allows arbitrary flexibility but is not portable.

4.4.2 Server Routines

A server makes itself available with two routines. First it must call MPI_PORT_OPEN to establish a port at which it may be contacted. Secondly it must call MPI_ACCEPT to accept connections from clients.

MPI_PORT_OPEN(info, port_name)

IN	info	implementation-specific information on how to establish an address (string)
OUT	port_name	newly established port (string)

```
int MPI_Port_open(MPI_Info info, char port_name[])
```

```
MPI_PORT_OPEN(INFO, PORT_NAME, IERR)
    CHARACTER*(*) PORT_NAME
    INTEGER INFO, IERR
```

This function establishes a network address, encoded in the port_name string, at which the server will be able to accept connections from clients. port_name is supplied by the system, possibly using information in the info argument.

port_name is a system-supplied string that identifies the given port and can be used by a client to contact the server. The maximum size string that may be supplied by the system is MPI_MAX_NAME_LEN.

port_name is essentially a network address. It is unique within the communication universe to which is belongs (determined by the implementation), and may be used by any client within that communication universe. For instance, if it is an internet (host:port) address, it will be unique on the internet. If it is a low level switch address on an IBM SP it will be unique to that SP.

> *Advice to implementors.* These examples are not meant to constrain implementations. A port_name could, for instance, contain a user name or the name of a batch job, as long as it is unique within some well-defined communication domain. The larger the communication domain, the more useful MPI's client/server functionality will be. (*End of advice to implementors.*)

The precise form of the address is implementation-defined. For instance, an internet address may be a host name or ip address, or anything that the implementation can decode into an ip address. A port name may be reused after it is freed with MPI_PORT_CLOSE 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_PORT_CLOSE(port_name)

IN	port_name	a port (string)

```
int MPI_Port_close(char port_name[])

MPI_PORT_CLOSE(PORT_NAME, IERR)
    CHARACTER*(*) PORT_NAME
    INTEGER IERR
```

This function releases the network address represented by port_name. If there are any pending MPI_IACCEPT operations on the port, the port is marked for deletion but it is not actually released until the operations complete or are cancelled.

MPI_ACCEPT(port_name, info, root, comm, newcomm)

IN	port_name	port name (string, used only on root)
IN	info	implementation-dependent information (handle, used only on root)
IN	root	rank in comm of root node (integer)
IN	comm	communicator over which call is collective (handle)
OUT	newcomm	intercommunicator with client as remote group (handle)

```
int MPI_Accept(char port_name[], MPI_Info info, int root, MPI_Comm comm,
             MPI_Comm* newcomm)

MPI_ACCEPT(PORT_NAME, ROOT, COMM, NEWCOMM, IERR)
    CHARACTER*(*) PORT_NAME
    INTEGER COMM, ROOT, NEWCOMM, IERR
```

MPI_ACCEPT establishes communication with a client. It is collective over the calling communicator. It returns an intercommunicator that allows communication with the client.

The port_name must have been established through a call to MPI_PORT_OPEN.

info is a implementation-defined string that may allow fine control over the ACCEPT call.

4.4.3 Client Routines

There is only one routine on the client side.

MPI_CONNECT(port_name, info, root, comm, newcomm)

IN	port_name	network address (string, used only on root)
IN	info	implementation-dependent information (string, used only on root)
IN	root	rank in comm of root node (integer)
IN	comm	communicator over which call is collective (string)
OUT	newcomm	intercommunicator with server as remote group (handle)

```
int MPI_Connect(char port_name[], MPI_Info::info, int root, MPI_Comm comm,
             MPI_Comm* newcomm)
```

```
MPI_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERR)
    CHARACTER*(*) PORT_NAME, INFO
    INTEGER ROOT, COMM, NEWCOMM, IERR
```

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_ACCEPT.

If the named port does not exist (or has been closed) MPI_CONNECT returns a soft error of class MPI_ERR_PORT and sets newcomm to MPI_COMM_NULL.

If the port exists, but does not have a pending ACCEPT, the connection attempt will eventually time out after an implementation-defined time, or succeed when the server calls ACCEPT. In the case of a time out, MPI_CONNECT returns a soft error of class MPI_ERR_PORT and sets newcomm to MPI_COMM_NULL.

> *Advice to implementors.* The time-out period may be arbitrarily short or long. A high quality implementation will ensure that a CONNECT will usually succeed if the ACCEPT happens at "about the same time." A high quality implementation may also provide a mechanism, through the info arguments to PORT_OPEN, ACCEPT and CONNECT, for the user to specify the behavior. (*End of advice to implementors.*)

port_name is the address of the server. It must be the same as the name returned by MPI_PORT_OPEN 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.

4.4.4 Name publishing

The routines in this section provide a mechanism for publishing names. A (service_name, port_name) pair is published by the server, and may be retrieved by a client using the service_name only. An MPI implementation defines the *scope* of the service_name, that is, the domain over which the service_name can be retrieved. If the domain is the empty set, that is, if no client can retrieve the information, then we say that name publishing is not supported. If name publishing is not supported, these routines return the soft error class MPI_ERR_NOPUBLISHING.

MPI_NAME_PUBLISH(port_name, info, service_name)

IN	port_name	a port name (string)
IN	info	implementation-specific information (handle)
IN	service_name	a service name to associate with the port (string)

```
int MPI_Name_publish(char port_name[], MPI_Info info, char service_name[])
```

```
MPI_NAME_PUBLISH(PORT_NAME, INFO, SERVICE_NAME, IERR)
    INTEGER INFO, IERR
    CHARACTER*(*) PORT_NAME, SERVICE_NAME
```

This routine publishes the pair (port_name, service_name) so that an application may retrieve a system-supplied port_name using a well-known service_name.

The implementation must define the *scope* of a published service name, that is, the domain over which the service name is unique, and conversely, the domain over which the (port name, service name) pair may be retrieved. For instance, a service name may be unique to a job (where job is defined by a distributed operating system or batch scheduler), unique to a machine, or unique to a kerberos realm. The scope may depend on the info argument to MPI_NAME_PUBLISH.

It is allowed to publish more than one service_name for a single port_name. On the other hand, if service_name has already been established within the current scope, the default behavior is that MPI_NAME_PUBLISH fails and returns a soft error of error class MPI_ERR_SERVICE. An MPI implementation may, through an override mechanism in the info argument to MPI_NAME_PUBLISH, provide a way to allow multiple servers with the same service in the same scope. In this case, an implementation-defined policy will determine which of several port names is returned by MPI_NAME_GET.

Note that while service_name has a limited scope, determined by the implementation, port_name always has global scope within the communication universe used by the implementation (i.e., it is globally unique).

If port_name should be the name of a port established by MPI_PORT_OPEN and not yet deleted by MPI_PORT_CLOSE. If it is not, the results are undefined.

Advice to implementors. In some cases, an MPI implementation may use a name service that a user can also access directly. In this case, a name published by MPI could easily conflict with a name published by a user. In order to avoid such conflicts, MPI implementations should mangle service names so that they are unlikely to conflict with user code that makes use of the same service. Such name mangling will of course be completely transparent to the user.

The following situation is problematic but unavoidable, if we want to allow implementations to use nameservers. Suppose there are multiple instances of "ocean" running on a machine. If the scope of a service name is confined to a job, then multiple oceans can coexist. If an implementation provides site-wide scope, however, multiple instances are not possible as all calls to MPI_NAME_PUBLISH after the first will fail. There is no universal solution to this.

To handle these situations, a high quality implementation should make it possible to limit the domain over which names are published. (*End of advice to implementors.*)

MPI_NAME_UNPUBLISH(port_name, service_name)

IN	port_name	a port name (string)
IN	service_name	a service name (string)

```
int MPI_Name_unpublish(char port_name[], char service_name[])
```

```
MPI_NAME_UNPUBLISH(PORT_NAME, SERVICE_NAME, IERR)
    INTEGER IERR
    CHARACTER*(*) PORT_NAME, SERVICE_NAME
```

This routine deletes a service name that has been previously published. Attempting to delete a name that has not been published is erroneous, and is indicated by the error code MPI_ERR_SERVICE.

All published names should be deleted before the corresponding port is closed and before the publishing process exits.

The behavior of MPI_NAME_UNPUBLISH is implementation dependent when a process tries to delete a name that it did not publish.

Advice to implementors. Because the mechanism for name publishing may depend on the info argument to MPI_NAME_PUBLISH, and indirectly on the info argument to MPI_PORT_OPEN, it may be necessary to cache publishing information internally on port_name or service_name so that the name can be properly deleted. (*End of advice to implementors.*)

MPI_NAME_GET(service_name, info, port_name)

IN	service_name	a service name (string)
IN	info	implementation-specific information (handle)
OUT	port_name	a port name (string)

```
int MPI_Name_get(char service_name[], MPI_Info info, char port_name[])
```

```
MPI_NAME_GET(SERVICE_NAME, INFO, PORT_NAME, IERR)
    CHARACTER*(*) SERVICE_NAME
    CHARACTER*(*) PORT_NAME
    INTEGER INFO, IERR
```

This function retrieves a port_name published by MPI_NAME_PUBLISH with service_name. If service_name has not been published, it returns a soft error with error class MPI_ERR_NAME.

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_NAME_PUBLISH to tell the implementation how to publish names, a similar info argument may be required to MPI_NAME_GET to tell the information how to retrieve names.

4.4.5 Non-blocking routines

MPI_IACCEPT(port_name, info, root, comm, newcomm, request)

IN	port_name	port name (string, used only on root)
IN	info	implementation-dependent information (handle, used only on root)
IN	root	rank in comm of root node (integer)
IN	comm	communicator over which call is collective (string)
OUT	newcomm	intercommunicator with client as remote group (handle)
OUT	request	request object (handle)

```
int MPI_Iaccept(char port_name[], MPI_Info info, int root, MPI_Comm comm,
              MPI_Comm* newcomm, MPI_Request* request)
```

```
MPI_IACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, REQUEST, IERR)
    CHARACTER*(*) PORT_NAME
    INTEGER ROOT, INFO, COMM, NEWCOMM, REQUEST, IERR
```

MPI_ICONNECT(port_name, info, root, comm, newcomm, request)

IN	port_name	network address (string, used only on root)
IN	info	implementation-dependent information (handle, used only on root)
IN	root	rank in comm of root node (integer)
IN	comm	communicator over which call is collective (string)
OUT	newcomm	intercommunicator with server as remote group (handle)
OUT	request	request object (handle)

```
int MPI_Iconnect(char port_name[], MPI_Info info, int root, MPI_Comm comm,
              MPI_Comm* newcomm, MPI_Request* request)
```

```
MPI_ICONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, REQUEST, IERR)
    CHARACTER*(*) PORT_NAME,
```

```
INTEGER ROOT, INFO, COMM, NEWCOMM, REQUEST, IERR
```

These functions are identical to their blocking counterparts except that they do not block. An application may test or wait for completion using MPI_WAIT, MPI_TEST, and related routines. It may also call MPI_TEST_CANCELLED on a returned status. These non-blocking collective call obey the same semantics as the MPI_ISPAWN_* calls except that they may be cancelled if there was no mixture of blocking and non-blocking calls. The cancel operation is collective over the original calling communicator. For the status argument returned by MPI_WAIT, etc., only the MPI_ERROR field is defined. This field will be MPI_SUCCESS if the operation was successful and contain an error code otherwise.

4.4.6 Reserved key values

The following key values are reserved for info arguments in section 4.4. An implementation is not required to interpret these key values, but if it does interpret the key value, it must provide the functionality described.

ip_port Value contains IP port number at which to establish a port. (Reserved for MPI_PORT_OPEN only).

ip_address Value contains IP address at which to establish a port. If the address is not a valid IP address of the host on which the MPI_PORT_OPEN call is made, the results are undefined. (Reserved for MPI_PORT_OPEN only).

4.4.7 Client/Server Examples

Discussion: These examples have not been updated

Simplest example - completely portable.

The following example shows the simplest way to use the client/server interface. It does not use service names at all.

On the server side:

```
char myport[MPI_MAX_NAME_LEN];
MPI_Comm intercomm;
/* ... */
MPI_Port_open(NULL, &myport);
printf("port name is: ", %s\n", myport);

MPI_Accept(myport, 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_NAME_LEN];
printf("enter port name: "); fflush(stdout);
gets(name);
MPI_Connect(name, NULL, 0, MPI_COMM_SELF, &intercomm);
```

Ocean/Atmosphere - relies on name publishing

In this example, the "ocean" application is the "server" side of a coupled ocean-atmosphere climate model. It assumes that the MPI implementation publishes names.

```
MPI_Port_open(NULL, &port_name);
MPI_Name_publish(port_name, "ocean");

MPI_Accept(port_name, "ocean", 0, MPI_COMM_SELF, &intercomm);
/* do something with intercomm */
MPI_Name_delete(port_name, "ocean");
```

On the client side:

```
MPI_Name_get("ocean", port_name);
MPI_Connect( port_name, NULL, 0, MPI_COMM_SELF, &intercomm);
```

Ocean/Atmosphere - uses name publishing if possible

This example is similar to the previous one except that is makes a run time decision (compile time would also be possible) based on whether names are published.

```
MPI_Port_open(NULL, port_name);
MPI_Publish_name(port_name,"ocean");

MPI_Accept(port_name, NULL, 0, MPI_COMM_SELF, &intercomm);
if (!MPI_NAMES_ARE_PUBLISHED) {
  if (batch_mode) error(FATAL, "can't transfer port name");
  printf("port name is: ", %s\n", port_name);
}
/* do something with intercomm */
MPI_Name_delete(port_name, "ocean");
```

On the client side:

```
if (!MPI_NAMES_ARE_PUBLISHED) {
  if (batch_mode) error(FATAL, "can't transfer port name");
  printf("low quality MPI implementation encountered. \\
          enter port name: "); fflush(stdout);
  gets(port_name);
```

```
    } else {
      MPI_Name_get("ocean", port_name);
    }
    MPI_Connect(port_name, NULL, 0, MPI_COMM_SELF, &intercomm);
```

Simple client-server example.

This is a simple example; the server accepts only a single connection at a time and serves that connection until the client requests to be disconnected. The server is a single process.

Here is the server. It accepts a single connection and then processes data until it receives a message with tag `1`. A message with tag `0` tells the server to exit.

```
#include "mpi.h"
main( int argc, char **argv )
{
    MPI_Comm client;
    MPI_Status status;
    char port_name[MPI_MAX_NAME_LEN];
    double buf[MAX_DATA];
    int    size, again;

    MPI_Init( &argc, &argv );
    MPI_Comm_size(MPI_COMM_WORLD, size);
    if (size != 0) error(FATAL, "Server too big");
    MPI_Port_open(NULL, port_name);
    printf("server available at %s\n",port_name);
    while (1) {
        MPI_Accept( port_name, NULL, 0, MPI_COMM_WORLD, &client );
        again = 1;
        while (again) {
            MPI_Recv( buf, MAX_DATA, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG,
                      client, &status );
            switch (status.tag) {
                case 0: MPI_Comm_free( &client );
                        MPI_Finalize();
                        return 0;
                case 1: MPI_Comm_disconnect( &client );
                        again = 0;
                        break;
                case 2: /* do something */
                ...
                default:
                        MPI_Abort( MPI_COMM_WORLD, "Unexpected message type" );
                }
            }
        }
}
```

Here is the client.

```
#include "mpi.h"
main( int argc, char **argv )
{
    MPI_Comm server;
    double buf[MAX_DATA];
    char port_name[MAX_NAME_SIZE];

    MPI_Init( &argc, &argv );
    strcpy(port_name, argv[1] );/* assume server's name is cmd-line arg */

    MPI_Connect( port_name, 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();
}
```

If the server needs to manage multiple connections at once, it can use MPI_IACCEPT instead. The client need not be changed.

4.5 Other routines

4.5.1 Info Management Routines

Current Status: Two votes.

Many of the routines in this chapter take an argument info. info is an opaque object with a handle of type MPI_Info in C and integer in Fortran. It consists of (key,value) pairs (both key and value are strings), with a unique value for every defined key. 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.

If a function does not recognize a key, it will ignore it, unless otherwise specified. If an implementation recognizes a key but does not recognize the format of the corresponding value, the result is undefined.

Keys have an implementation-defined maximum length of MPI_INFO_MAX_KEYLEN, which is at least 32 and at most 255. Values have an implementation-defined maximum length of MPI_INFO_MAX_VALUELEN. String lengths for both key and value do not include the final null character in C. C programmers should therefore be careful to allocate one extra character. In Fortran, leading and trailing spaces are stripped from both. Returned values will never be larger than these maximum lengths.

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_INFO_MAX_KEYLEN+1. The limitation on value sizes is so that an implementation is not forced to deal with arbitrarily long strings. MPI_INFO_MAX_VALUELEN might be very large, so it might not be wise to declare a string of that size.

(*End of rationale.*)

When it is an argument to a non-blocking routine, info is parsed before that routine returns, so that it may be modified or freed immediately after return."

MPI_INFO_CREATE(info)

OUT	info	info object created (handle)

```
int MPI_Info_create(MPI_Info *info)
```

```
MPI_INFO_CREATE(INFO, IERR)
    INTEGER INFO, IERR
```

Creates a new info object. The newly created object contains no key/value pairs.

MPI_INFO_SET(info, key, value)

INOUT	info	info object (handle)
IN	key	key (string)
IN	value	value (string)

```
int MPI_Info_set(MPI_Info info, char key[], char value[])
```

```
MPI_INFO_SET(INFO, KEY, VALUE, IERR)
    INTEGER INFO, IERR
    CHARACTER*(*) KEY, VALUE
```

Adds the (key,value) pair to info. Overrides if previously set. key and value are null-terminated strings in C. In Fortran, leading and trailing spaces in key and value are ignored. If either key or value are larger than the allowed maximums, the call is erroneous.

MPI_INFO_DELETE(info, key)

INOUT	info	info object (handle)
IN	key	key (string)

```
int MPI_Info_delete(MPI_Info info, char key[])
```

```
MPI_INFO_DELETE(INFO, KEY, IERR)
    INTEGER INFO, IERR
    CHARACTER*(*) KEY
```

Deletes a (key,value) pair from info. If key is not defined in info, the call returns a continuable error of class MPI_ERR_INFO_NOKEY.

MPI_INFO_GET(info, key, valuelen, value, flag)

IN	info	info object (handle)
IN	key	key (string)
IN	valuelen	length of value arg (integer)
OUT	value	value (string)
OUT	flag	true if key defined, false if not (boolean)

```
int MPI_Info_get(MPI_Info info, char  key[], int valuelen, char value[],
              int* flag)
```

```
MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERR)
    INTEGER INFO, VALUELEN, IERR
    CHARACTER*(*) KEY, VALUE
    LOGICAL FLAG
```

This function retrieves the value associated with key in a previous call to MPI_INFO_SET. If such a key exists, it sets flag to true and returns the value in value, otherwise it sets flag to false and leaves value unchanged. valuelen is the number of characters available in value. If it is less than the actual size of the value, the value is truncated, and MPI_INFO_GET returns the error MPI_ERR_INFO_VALUE.

If key is larger than MPI_INFO_MAX_KEYLEN, the call is erroneous.

MPI_INFO_GET_VALUELEN(info, key, valuelen, flag)

IN	info	info object (handle)
IN	key	key (string)
OUT	valuelen	length of value arg (integer)
OUT	flag	true if key defined, false if not (boolean)

```
int MPI_Info_get_valuelen(MPI_Info info,char key[],int* valuelen,int* flag)
```

```
MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERR)
    INTEGER INFO, VALUELEN, IERR
    LOGICAL FLAG
    CHARACTER*(*) KEY
```

Retrieves the length of the value associated with key. If key is defined, valuelen is set to the length of its associated value and flag is set to true. If key is not defined, valuelen is not touched and flag is set to false. The length returned in C or C++ does not include the end-of-string character.

If key is larger than MPI_INFO_MAX_KEYLEN, the is erroneous.

MPI_INFO_GET_NKEYS(info, nkeys)

IN	info	info object (handle)
OUT	nkeys	number of defined keys (integer)

```
int MPI_Info_get_nkeys(MPI_Info info, int* nkeys)
```

```
MPI_INFO_GET_NKEYS(INFO, NKEYS, IERR)
    INTEGER INFO, NKEYS, IERR
```

Returns the number of currently defined keys in info.

MPI_INFO_GET_NTHKEY(info, n, key)

IN	info	info object (handle)
IN	n	key number (integer)
OUT	key	key (string)

```
int MPI_Info_get_nthkey(MPI_Info info, int n, char key[])
```

```
MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERR)
    INTEGER INFO, N, IERR
    CHARACTER*(*) KEY
```

Returns the nth defined key in info. Keys are numbered $0 \ldots N-1$ where N is the value returned by MPI_INFO_GET_NKEYS. All keys between 0 and $N-1$ are guaranteed to be defined. The number of a given key does not change as long as info is not modified with MPI_INFO_SET or MPI_INFO_DELETE.

MPI_INFO_DUP(info, newinfo)

IN	info	info object (handle)
OUT	newinfo	info object (handle)

```
int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo)
```

```
MPI_INFO_DUP(INFO, NEWINFO, IERR)
    INTEGER INFO, NEWINFO, IERR
```

Duplicates an existing info object. Creates a new object, with same (key,value) pairs, and the same ordering of keys.

MPI_INFO_FREE(info)

INOUT	info	info object (handle)

```
int MPI_Info_free(MPI_Info* info)
```

```
MPI_INFO_FREE(INFO, IERR)
```

```
INTEGER INFO, IERR
```

Frees info and sets it to MPI_INFO_NULL.

For non-blocking operations, the info argument is parsed inside the call, and may be freed or modified as soon as the call returns.

4.5.2 Universe Size

Current Status: 2 votes

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 integer attribute on MPI_COMM_WORLD, MPI_UNIVERSE_SIZE, that allows the application to obtain this information in a portable manner. This attribute indicates the total number of processes that are expected. An application typically subtracts the size of MPI_COMM_WORLD from MPI_UNIVERSE_SIZE to find out how many processes it should spawn. MPI_UNIVERSE_SIZE is initialized in MPI_INIT and is not changed by MPI. If defined, it has the same value on all processes of MPI_COMM_WORLD. MPI_UNIVERSE_SIZE is determined by the application startup mechanism in a way not specified by MPI. (The size of MPI_COMM_WORLD is another example of such a parameter).

Possibilities for how MPI_UNIVERSE_SIZE might be set include

- A `-universe_size` argument to a program that starts MPI processes.
- Automatic interaction with a batch scheduler to figure out how many processors have been allocated to an application.
- An environment variable set by the user.
- Extra information passed to MPI_SPAWN through the info argument.

An implementation must document how MPI_UNIVERSE_SIZE is set. An implementation may not support the ability to set MPI_UNIVERSE_SIZE, in which case the attribute MPI_UNIVERSE_SIZE is not set.

MPI_UNIVERSE_SIZE is a recommendation, not necessarily a hard limit. For instance, some implementations may allow an application to spawn 50 processes per processor, if they are requested. However, it is likely that the user only wants to spawn one process per processor.

MPI_UNIVERSE_SIZE is assumed to have been specified when an application was started, and is in essence a portable mechanism to allow the user to pass to the application (through the MPI process startup mechanism) a piece of critical runtime information. Note that no interaction with the runtime environment is required. If the runtime environment changes size while an application is running MPI_UNIVERSE_SIZE is not updated, and the application must find out through direct communication with the runtime system.

4.5.3 Singleton MPI_INIT

Current Status: 1 vote

A high-quality implementation will allow any process (including those not started with a "parallel application" mechanism) to become an MPI process by calling MPI_INIT. Such a process can then connect to other MPI processes using the MPI_ACCEPT and MPI_CONNECT routines, or spawn other MPI processes. MPI does not mandate this behavior, but strongly encourages it where technically feasible.

> *Advice to implementors.* To start an MPI-1 application with more than one process requires some special coordination. The processes must be started at the "same" time, they must have a mechanism to establish communication, etc. Either the user or the operating system must take special steps beyond simply starting processes.
>
> When an application enters MPI_INIT, clearly it must be able to determine if these special steps were taken. MPI-1 does not say what happens if these special steps were not taken - presumably this is treated as an error in starting the MPI application. MPI-2 recommends the following behavior.
>
> 1. If a process enters MPI_INIT and determines that no special steps were taken (i.e., it has not been given the information to form an MPI_COMM_WORLD with other processes) it succeeds and forms a singleton MPI program, that is, one in which MPI_COMM_WORLD has size 1.
>
> In some implementations, MPI may not be able to function without an "MPI environment." For example, MPI may require that daemons be running or MPI may not be able to work at all on the front-end of an MPP. In this case, an MPI implementation may either
>
> 1. Create the environment (e.g., start a daemon) or
> 2. Return 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 return an error.
>
> (*End of advice to implementors.*)

4.5.4 MPI_APPNUM

Current Status: 1 vote

There is a predefined integer attribute MPI_APPNUM of MPI_COMM_WORLD. If a process was spawned with MPI_SPAWN_MULTIPLE, MPI_APPNUM is the command-line number that generated the current process. Numbering starts from zero. If a process was spawned with MPI_SPAWN it will have MPI_APPNUM equal to zero.

Additionally, if the process was not started by a spawn call, but by an implementation-specific startup mechanism that can handle multiple process specifications, MPI_APPNUM should be set to the number of the corresponding process specification. In particular, if is started with

```
mpiexec spec0 [: spec1 : spec2 : ...]
```

MPI_APPNUM should be set to the number of the corresponding specification.

If an application was not spawned with MPI_SPAWN or MPI_SPAWN_MULTIPLE, and MPI_APPNUM doesn't make sense in the context of the implementation-specific startup mechanism, MPI_APPNUM is not set.

MPI implementations may optionally provide a mechanism to override the value of MPI_APPNUM through the info argument. MPI reserves the following key for all SPAWN calls.

appnum Value contains an integer overrides the default value for MPI_APPNUM in the child.

.

Rationale. When a single application is started, it is able to figure out how many processes there are by looking at the size of MPI_COMM_WORLD. An application consisting of multiple SPMD sub-applications has no way to find out how many sub-applications there are and to which sub-application the process belongs. While there are ways to figure it out in special cases, there is no general mechanism. MPI_APPNUM provides such a general mechanism. (*End of rationale.*)

4.5.5 Releasing Connections

Current Status: 1 vote

Before a client and server connect, they are independent MPI applications. An error in one does not affect the other. After establishing a connection with MPI_CONNECT and MPI_ACCEPT, an error in one may affect the other. It is desirable for a client and server to be able to disconnect, so that an error in one will not affect the other. Similarly, it might be desirable for a parent and child to disconnect, so that errors in the child do not affect the parent, or vice-versa.

- Two processes are **connected** if
    1. They both belong to the same intercommunicator or communicator (including MPI_COMM_WORLD) or
    2. They are are both connected to a third process.
- Two processes are **disconnected** (also **independent**) if they are not connected.
- By the above definitions, connectivity is a transitive property, and divides the universe of MPI processes into disconnected (independent) sets (equivalence classes) of processes.
- Processes which are connected, but don't share the same MPI_COMM_WORLD may become disconnected (independent) if the communication path between them is broken by using MPI_COMM_DISCONNECT.

The following additional rules apply to MPI-1 functions:

- MPI_FINALIZE is collective over a set of connected processes.
- MPI_ABORT does not abort independent processes. As in MPI-1, it may abort all processes in MPI_COMM_WORLD (ignoring its comm argument). Additionally, it may abort connected processes as well, though it makes a "best attempt" to abort only the processes in comm.
- If a process exits without calling MPI_FINALIZE the effect on other connected processes is not defined.

MPI_COMM_DISCONNECT(comm)

INOUT	comm	communicator (handle)

```
int MPI_Comm_disconnect(MPI_Comm* comm)
```

```
MPI_COMM_DISCONNECT(COMM, IERR)
    INTEGER COMM, IERR
```

This function waits for all pending communication on comm to complete and then sets the communicator to MPI_COMM_NULL in the same manner as MPI_COMM_FREE. It is a collective operation. It may not be called with the communicator MPI_COMM_WORLD. It is erroneous to call MPI_COMM_DISCONNECT if not all communication can be completed, for instance, if there has been an send operation without a matching receive, or a collective operation that has not been called on all processes in the communicator. It is not erroneous to have outstanding non-blocking operations, as long as those operations can complete. After MPI_COMM_DISCONNECT returns, all non-blocking operations on that communicator will be finished, and all associated handlers will have been called.

It is the user's responsibility to free all outstanding requests associated with a communicator before calling MPI_COMM_DISCONNECT. A request may be freed by MPI_REQUEST_FREE or some variant of TEST or WAIT.

Note that it may be necessary to disconnect several communicators before two processes are completely independent.

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.*)

Chapter 5

One-Sided Communications

Current Status: Passed twice – semantic section excepted

5.1 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 but where the data distribution is fixed or slowly changing. Each process can compute what data it needs to access or update at other processes. However, processes may not know which data in their own memory need be accessed or updated by remote processes, nor do they know the identity of these processes. Regular send/receive communication requires matching operations by sender and receiver. In our situation, the transfer parameters are all available on one side. In order to issue the matching operations, one needs to distribute this information. This requires all processes to participate in a time consuming global computation, or to periodically poll for potential communication requests to receive and act upon. The use of put or get calls avoids the need for the global computation or the 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. Three communication calls are provided: MPI_PUT (remote write), MPI_GET (remote read) and MPI_ACCUMULATE (remote update). A larger number of synchronization calls are provided, to support different synchronization styles. The design is similar to that of weakly coherent memory systems: correct ordering of memory accesses has to be imposed by the user, using synchronization calls; the implementation can delay communication operations until the synchronization call occurs, for efficiency.

The design of RMA functions allows implementors to take advantage, in many cases, of fast communication mechanisms provided by various platforms, such as coherent or non-coherent shared memory, DMA engines, hardware-supported put/get operations, communication coprocessors, etc. The most frequently used RMA communication mechanisms can be layered on top of message passing. However, support for asynchronous communication agents (handlers, threads, etc.) is needed, for certain RMA functions, in a distributed memory environment.

We shall denote by **origin** the process that performs the call, and by **target** the other 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.

5.2 Initialization

5.2.1 Memory Allocation

In some systems, message-passing and RMA operations will 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 RMA is not mandatory. However, implementations may restrict the use of the MPI_WIN_LOCK and MPI_WIN_UNLOCK functions to windows allocated in such memory.

MPI_MEM_ALLOC(size, comm, base)

IN	size	size of memory segment in bytes (non negative integer)
IN	comm	communicator for group where communication is expected to occur (handle)
OUT	base	initial address of memory allocated (integer)

```
int MPI_Mem_alloc(int size, MPI_Comm comm, void *base)
```

```
MPI_MEM_ALLOC( SIZE, COMM, BASE, IERROR)
    INTEGER SIZE, COMM, IERROR
    INTEGER(KIND=MPI_ADDRESS_SIZE) BASE
```

MPI_MEM_FREE(base)

IN	base	initial address of memory segment allocated by MPI_MEM_MALLOC

```
int MPI_Mem_free(void *base)
```

```
MPI_MEM_FREE( BASE, IERROR)
    INTEGER(KIND=MPI_ADDRESS_SIZE) BASE
```

The Fortran 90 type INTEGER(KIND = MPI_ADDRESS_SIZE) is an address sized integer, as described in Section 9.7.6, on page 193. The functions are useful, in Fortran, only on systems that support "true" (C-like, Cray-like) pointers.

Advice to users. The communicator argument in the call is only a "performance hint": the allocated memory may be used for communication with any communicator, like any other memory.

In the C binding, the base arguments of the calls MPI_Mem_alloc and MPI_Mem_free are not the same variable: a call to MPI_Mem_alloc(&base) should be paired with a call to MPI_Mem_free(base) (one less dereference). Both arguments are declared to be of same type void*, so as to facilitate type casting. (*End of advice to users.*)

Advice to implementors. If MPI_MEM_ALLOC allocates special memory, then a design similar to the design of C malloc and free functions has to be used, in order to find out the size of a memory segment, when the segment is freed. If no special memory is used, MPI_MEM_MALLOC simply invokes malloc, and MPI_MEM_FREE invokes free.

A call to MPI_MEM_ALLOC can be used in shared memory systems to allocate memory in a shared memory segment. (*End of advice to implementors.*)

Example 5.1 Example of use of MPI_MEM_ALLOC, in Fortran with pointer support. We assume 4 byte REALs.

```
REAL A
POINTER (P, A(100,100))   ! no memory is allocated
CALL MPI_MEM_MALLOC(4*100*100, COMM, P, IERR)
                          ! memory is allocated
```

Since standard Fortran does not support (C-like) pointers, this code is not Fortran 77 or Fortran 90 code. Some compilers (in particular g77 and Fortran compilers for Intel) do not support this code.

Missing: Need new error codes.

Discussion: These two functions should move to misc chapter

5.2.2 Window Creation

The initialization operation allows each process to specify, in a collective operation, a "memory window" that is made accessible to accesses by remote processes. The information is attached to an opaque object returned by the call. This object represents the group of processes that own and access the set of windows, and the attributes of each window, as specified by the init call.

MPI_WIN_INIT (base, size, disp_unit, info, comm, win)

IN	base	initial address of window (choice)
IN	size	size of window in bytes (nonnegative integer)
IN	disp_unit	local unit size for displacements in RMA operations (positive integer)
IN	info	info argument
IN	comm	communicator (handle)
OUT	win	window object returned by the call (handle)

```
int MPI_Win_init(void* base, int size, int disp_unit, MPI_Info info,
             MPI_Comm comm, MPI_Win *win)
```

```
MPI_WIN_INIT(BASE, SIZE, DISP_UNIT, INFO, COMM, WIN, IERROR)
    <type> BASE(*)
    INTEGER SIZE, DISP_UNIT, INFO, COMM, WIN, IERROR
```

This is a collective call executed by all processes in the group of comm. It returns a window object that can be used by these processes to perform RMA operations. Each process specifies a window of existing memory that it exposes to RMA accesses by the processes in the group of comm. The window consists of size bytes, starting at address base. A process may elect to expose no memory by specifying size = 0.

The displacement unit argument is provided to facilitate address arithmetics 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.

> *Advice to users.* Common choices for disp_unit are 1 (no scaling), and (in C syntax) `sizeof(type)`, for a window that consists of an array of elements of type `type`. The later choice will allow one to use array indices in RMA calls, and have those scaled correctly to byte displacements, even in a heterogeneous environment. (*End of advice to users.*)

The info argument provides optimization hints to the runtime about the expected usage pattern of the window. The following info key is predefined:

no_locks If set to true, then the implementation may assume that the local window is never locked (by a call to MPI_WIN_LOCK). This implies that this window is not used for 3-party communication, and RMA can be implemented with no (less) asynchronous agent activity at this process.

The various processes in the group of comm may specify completely different target windows, in location, size, displacement units and info arguments. As long as all the gets and puts to a particular process fit its specific target window this should pose no problem.

The same area in memory may appear in multiple windows, each associated with a different communicator. However, concurrent communications to distinct, overlapping windows may lead to erroneous results.

Alternatives: If caching on objects other than communicators is supported, then the following will be added.

The information about the local window associated with win is cached with the newly created window object, and is attached to the following three attributes.

MPI_WINDOW_BASE: Initial address of local address

MPI_WINDOW_SIZE: size of local window, in bytes

MPI_WINDOW_DISP_UNIT: displacement unit used for local window

If a window object win was created by a call to MPI_WIN_INIT(base, size, disp_unit, ...), then MPI_HANDLE_ATTR_GET(win, MPI_WINDOW_BASE, val, flag, MPI_WIN) will return val = base; and similarly, for the other two.

Discussion: Do we want to add query functions to retrieve this information, if caching on windows is not supported?

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_MEM_ALLOC will be better. Such a "good" window for RMA communication is created by first allocating "good" memory with MPI_MEM_ALLOC, next creating an RMA window in the memory allocated with MPI_WIN_INIT.

Also, on some systems, performance is improved when windows boundaries are word or double-word aligned. (*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_INIT 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_MEM_ALLOC, or by other, implementation specific, mechanisms, together with information on the type of memory segment allocated. When a call to MPI_WIN_INIT occurs, then MPI checks which segment contains each window, and decide, accordingly, which mechanism to use for RMA operations.

Vendors may provide additional, implementation-specific mechanisms to allow "good" memory to be used for static variables. (*End of advice to implementors.*)

MPI_WIN_FREE(win)

INOUT	win	window object (handle)

```
int MPI_Win_free(MPI_Win *win)
```

```
MPI_WIN_FREE(WIN, IERROR)
    INTEGER WIN, IERROR
```

Frees the window object win and returns a null handle (equal to MPI_WIN_NULL). The usual object freeing semantics are used: the object deallocation will be delayed if there is an ongoing RMA communication that uses this object; the object will be actually freed some time after the call.

5.3 Communication Calls

MPI supports three RMA communication calls: MPI_PUT transfers data from the caller memory (origin) to the target memory; MPI_GET transfers data from the target memory to the caller memory; and MPI_ACCUMULATE updates locations in the target memory, e.g. by adding to these locations values sent from the caller memory. These operations are *nonblocking*: the call initiates the transfer, but the transfer may continue after the call returns. The transfer is completed, both at the origin and at the target, when a subsequent *synchronization* call is issued by the caller, on the involved window object. These synchronization calls are described in Section 5.4, Page 83. The local communication buffer of an RMA call should not be updated, and the local communication buffer of a get call should not be accessed after the RMA call, until the subsequent synchronization call completes.

Rationale. The rule above is more lenient than for message passing, where we do not allow two concurrent sends, with overlapping send buffers. Here, we allow two concurrent puts with overlapping send buffers. The reasons for this relaxation are

1. Users do not like that restriction, which is not very natural (it prohibits concurrent reads).
2. Weakening the rule does not prevent efficient implementation, as far as we know.
3. Weakening the rule is important for performance of RMA: we want to associate one synchronization call with as many RMA operations is possible. If puts from overlapping buffers cannot be concurrent, than we need to needlessly add synchronization points in the code.

(*End of rationale.*)

The calls use general datatype arguments to specify communication buffers at the origin and at the target. Thus, a transfer operation may also gather data at the source and scatter it at the destination. However, all arguments specifying both communication buffers are provided by the caller.

For all three calls, the target process may be identical with the origin process; i.e., a process may use a put call to move data in its memory.

Rationale. The choice of supported "self-communication" is the same as for message passing. It simplifies some coding, and is very useful with accumulate operations, to allow atomic updates of local variables. (*End of rationale.*)

5.3.1 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 parameters are provided by one call – the call executed by the origin process.

MPI_PUT(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win)

IN	origin_addr	address of origin buffer (choice)
IN	origin_count	number of elements sent (integer)
IN	origin_datatype	datatype of elements sent (handle)
IN	target_rank	rank of target (integer)
IN	target_disp	displacement from start of window to target buffer (nonnegative integer)
IN	target_count	number of elements received (integer)
IN	target_datatype	datatype used at the target (handle)
IN	win	window object used for communication (handle)

```
int MPI_Put( void *origin_addr, int origin_count, MPI_Datatype
             origin_datatype, int target_rank, int target_disp, int
             target_count, MPI_Datatype target_datatype, MPI_Win win)
```

```
MPI_PUT( ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
             TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP,
    TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR
```

Transfers origin_count successive entries of the type specified by the origin_datatype, starting at address origin_addr on the origin node to the target node specified by the comm, 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 that 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, and comm is a communicator for the group of win. The message sent must fit, without truncation, in the target buffer, which must fit in the target window. A high-quality implementation will raise an exception at the origin if this condition is violated.

The target_datatype argument is a handle to a datatype object defined at the origin process. However, this object is interpreted at the target process: the outcome is as if the target datatype object was defined at the target process, by the same sequence of calls used to define it at the origin process. The target datatype should contain only relative displacements, not absolute addresses. The same holds for get and accumulate.

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 heterogenous environment, if only portable datatypes are used. A datatype is *portable*, if it is basic datatype, or it is derived using only the type constructors MPI_TYPE_CONTIGUOUS, MPI_TYPE_VECTOR and MPI_TYPE_INDEXED. Such datatype is portable, because all displacements in such datatype are in terms of extents of basic datatypes. Therefore, if such datatype fits a data layout in one memory, it will fit the corresponding data layout in another memory, if the same declarations were used, even if the two systems have different architectures. On the other hand, if a datatype was constructed using MPI_TYPE_HINDEXED, MPI_TYPE_HVECTOR or MPI_TYPE_STRUCT, then the datatype contains explicit byte displacements. These displacements are unlikely to be chosen correctly if they fit data layout on one memory, but are used for data layouts on another process, running on a processor with a different architecture. In the later case, one should use the mechanisms defined in MPI for transferring datatypes across processes.

The performance of a put transfer can be significantly affected, on some systems, from the choice of window location and the shape and location of the origin and target buffer: transfers to a target window in memory allocated by MPI_MEM_ALLOC will be much faster on systems where some memory is shared; 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 prevent remote accesses to memory outside the window that was exposed by the process. This, both for debugging purposes, and for protection with client-server codes that use RMA. I.e., a high-quality implementation will check window bounds on each RMA call, and raise an MPI exception at the origin call, if an out-of-bound situation occurred. Note that the condition can be checked at the origin. (*End of advice to implementors.*)

5.3.2 Get

MPI_GET(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win)

OUT	origin_addr	address of origin buffer (choice)
IN	origin_count	number of elements received (integer)
IN	origin_datatype	datatype of elements received (handle)
IN	target_rank	rank of target (integer)
IN	target_disp	displacement from window start to target buffer (nonnegative integer)
IN	target_count	number of elements sent (integer)
IN	target_datatype	datatype of elements sent (handle)
IN	win	window object used for communication (handle)

```
int MPI_get( void *origin_addr, int origin_count, MPI_Datatype
             origin_datatype, int target, int target_disp, int
             target_count, MPI_Datatype target_datatype, MPI_Win win)
```

```
MPI_GET( ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET, TARGET_DISP,
             TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET, TARGET_DISP,
    TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR
```

Similar to MPI_PUT, except that the direction of data transfer is reversed. Data are copied from the target memory to the origin. The target buffer should be contained within the target window, and the copied data should fit, without truncation, in the origin buffer. A high quality implementation will raise an MPI exception at the origin, if either condition is violated.

5.3.3 Examples

Example 5.2 We show how to implement the generic indirect assignment `A = B(map)`, where `A`, `B` and `map` have the same distribution, and `map` is a permutation. To simplify, we assume a block distribution with equal size blocks.

```
SUBROUTINE MAP( A, B, map, m, comm, p)
INTEGER m, map(m), comm, p
REAL A(m), B(m)

INTEGER otypes(p), oindex(m),    ! used to construct origin datatypes
        tytpes(p), tindex(m),    ! used to construct target datatypes
        count(p), blen(m), total(p),
        sizeofreal, ierr

! This part does the work that depends on the locations of B.
! Can be reused while this does not change

CALL MPI_EXTENT(MPI_REAL, sizeofreal, ierr)

DO i=1,m
   blen(i) = 1
END DO

DO i=1,p
   count(i) = 1
END DO

CALL MPI_WIN_INIT(B, m*sizeofreal, sizeofreal, info, comm, win, ierr)

! This part does the work that depends on the value of map and
! the locations of the arrays.
! Can be reused while these do not change

! Compute number of elements to be received from each process

DO i=1,m
```

```
   j = map(i)/m+1
   count(j) = count(j)+1
END DO

total(1) = 0
DO i=2,p
   total(i) = total(i-1) + count(i-1)

DO i=1,p
   count(i) = 0
END DO

! compute origin and target indices of elements.
! elelement i moves to location k at process (j-1),
! where map(i) = (j-1)*m + (k-1),
! j = 1..p and k = 1..m


DO i=1,m
   j = map(i)/m+1
   k = MOD(map(i),m)+1
   count(j) = count(j)+1
   oindex(total(j) + count(j)) = i
   tindex(total(j) + count(j)) = k
END DO

! create origin and target datatypes for each get operation
DO i=1,p
   CALL MPI_TYPE_INDEXED(count(i), blen, oindex(total(i)+1),
                             MPI_REAL, otype(i), ierr)
   CALL MPI_TYPE_COMMIT(otype(i), ierr)
   CALL MPI_TYPE_INDEXED(count(i), blen, tindex(total(i)+1),
                             MPI_REAL, ttype(i), ierr)
   CALL MPI_TYPE_COMMIT(ttype(i), ierr)
END DO

! this part does the assignment itself
CALL MPI_WIN_BARRIER(win, ierr)
DO i=1,p
   CALL MPI_GET (A, 1, otype(i), i-1, 0, 1, ttype(i), win, ierr)
END DO
CALL MPI_WIN_BARRIER(win, ierr)

CALL MPI_WIN_FREE(win, ierr)
DO i=1,p
   CALL MPI_TYPE_FREE(otype(i), ierr)
   CALL MPI_TYPE_FREE(ttype(i), ierr)
END DO
```

```
RETURN
```

Example 5.3 A simpler version can be written, that does not require to built a datatype for the target buffer. But, then one needs a separate get call for each element, as illustrated below. This code is much simpler, but much less efficient, for large arrays, unless h/w is fast enough to transfer efficiently single words.

```
SUBROUTINE MAP( A, B, map, m, comm, p)
INTEGER m, map(m), comm, p
REAL A(m), B(m)
INTEGER sizeofreal, ierr

CALL MPI_TYPE_EXTENT(MPI_REAL, sizeofreal, ierr)
CALL MPI_WIN_INIT(B, m*sizeofreal, sizeofreal, info, comm, win, ierr)

CALL MPI_RMA_BARRIER(win, ierr)
DO i=1,m
   j = map(i)/p
   k = MOD(map(i),p)
   CALL MPI_GET(A(i), 1, MPI_REAL, j, k, 1, MPI_REAL, 0,
                                            getcomm, ierr)
END DO
CALL MPI_WIN_BARRIER(win, ierr)
CALL MPI_COMM_FREE(win, ierr)
RETURN
```

5.3.4 Accumulate Functions

It is often useful in a put operation to combine the data moved to the target process with the data that resides at that process, rather then replacing the data there. This will allow, for example, to accumulate a sum, by having all involved processes add their contribution to the sum variable in the memory of one process.

MPI_ACCUMULATE(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, op, win)

IN	origin_addr	initial address of buffer (choice)
IN	origin_count	number of elements in buffer (non negative integer)
IN	origin_datatype	data type of each buffer element (handle)
IN	target_rank	rank of destination process in comm (integer)
IN	target_disp	displacement from start of window to target buffer (nonnegative integer)
IN	target_count	number of elements in target buffer (integer)
IN	target_datatype	datatype used at target (handle)
IN	op	reduce operation (handle)
IN	win	window object (handle)

```
int MPI_Accumulate(void* origin_addr, int origin_count,
            MPI_Datatype origin_datatype, int target_rank, int target_disp,
            int target_count, MPI_Datatype target_datatype, MPI_Op op,
            MPI_Win win)
```

```
MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
            TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP,
    TARGET_DATATYPE, OP, WIN, IERROR
```

Accumulate the contents of the origin buffer (as defined by origin_addr, origin_count and origin_datatype) to the buffer specified by arguments target_count and target_datatype, at offset target_disp, in the target window specified by target_rank and comm, using the operation op. This is like MPI_PUT except that data is combined into the target area instead of overwriting it.

Any of the predefined operations for MPI_REDUCE can be used. User-defined functions cannot be used. For example, if op is MPI_SUM, each element of the origin buffer is added to the corresponding element in the target, replacing the former value in the target.

Both datatype arguments should be basic datatypes, or a derived datatype where all basic components are of the same basic datatype. The operation op applies to elements of that basic type. The target_datatype should not overlap with itself (the same element should not occur twice).

A new predefined operation, MPI_REPLACE, is defined. It corresponds to the associative function $f(a,b) = b$; i.e., the current value in the target memory is replaced by the value supplied by the origin.

Advice to users. MPI_PUT is a special case of MPI_ACCUMULATE, with the operation MPI_REPLACE Note, however, that MPI_PUT and MPI_ACCUMULATE have slightly different atomicity semantics – see Section 5.6. (*End of advice to users.*)

Example 5.4 We want to compute $\mathtt{B(j)} = \sum_{\mathtt{map(i)=j}} \mathtt{A(i)}$. The arrays A, B and map are distributed in the same manner. We write the simple version.

```
SUBROUTINE SUM( A, B, map, m, comm, p)
INTEGER m, map(m), comm, p
REAL A(m), B(m)

CALL MPI_RMA_INIT(B, m, MPI_REAL, info, win, ierr)

CALL MPI_RMA_BARRIER(win, ierr)
DO i=1,m
   j = map(i)/p
   k = MOD(map(i),p)
   CALL MPI_ACCUMULATE(A(i), 1, MPI_REAL, j, k, 1, 0, MPI_SUM,
                                              win, ierr)
END DO
CALL MPI_RMA_BARRIER(win, ierr)

CALL MPI_WIN_FREE(win, ierr)
RETURN
```

This code is identical to the code in Example 5.3, page 81, except that a call to get has been replaced by a call to accumulate. (Note that, if `map` is one-to-one, then the code computes $\texttt{B} = \texttt{A}(\texttt{map}^{-1})$, which the reverse assignment to the one computed in that previous example.) In a similar manner, we can replace in Example 5.2, page 79, the call to get by a call to accumulate, thus performing the computation with only one communication between any two processes.

5.4 Synchronization calls

MPI provides three synchronization mechanisms: The MPI_WIN_BARRIER 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. 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 is more efficient when each process communicates with few (logical) neighbors, and the communication graph is fixed, or changes infrequently. Finally shared and exclusive locks are provided by the two function MPI_WIN_LOCK and MPI_WIN_UNLOCK. Lock synchronization is useful for MPI applications that emulate a shared memory model via MPI calls; e.g., in a "billboard" model, where processes can, at random times, access or update different parts of the billboard.

Synchronization calls accept as a parameter a window object. These calls are used for ordering correctly conflicting accesses to RMA windows, and making sure that they are not concurrent. These synchronization calls may not, in general, be used to synchronize operations other than conflicting accesses to an RMA window.

RMA communications fall in two categories:

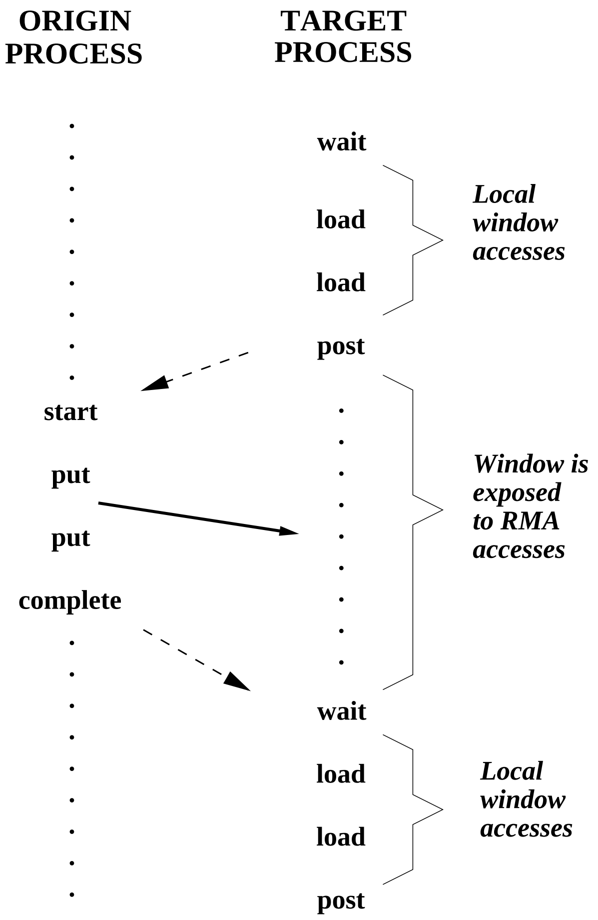

Figure 5.1: 2-party communication. Dashed arrows represent synchronizations and solid arrows represent data transfer.

- 2-party communication, where data is moved from the memory of one process to the memory of another, and both are involved in the communication. Typically, the origin process executes an RMA call, and the target process executes a local, conflicting load or store. The first two synchronization mechanisms fall in this category.
- 3-party communication, where two origin processes 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. The last synchronization mechanism falls in this category.

Figure 5.1 illustrates the general synchronization pattern, for 2-party communication, i.e., for conflicts between RMA accesses and local accesses. The synchronization between `post` and `start` ensures that the put calls of the origin process do not start before the preceding local accesses of the target process to the window have completed; the synchronization between `complete` and `wait` ensures that the put calls of the origin process complete before the following local accesses of the target process to the window start.

Figure 5.1 shows operations occurring in the natural temporal order implied by the

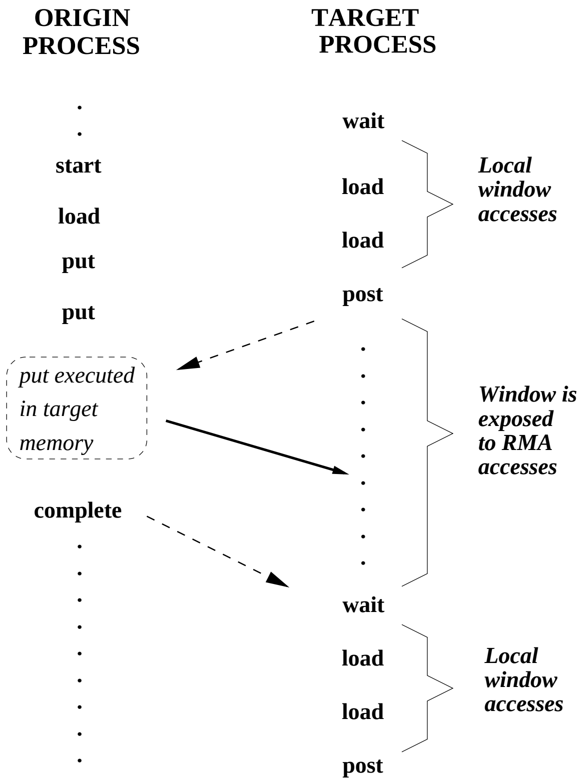

Figure 5.2: 2-party communication, with weak synchronization.

synchronizations: the post call occurs before the matching start call, and the complete call occurs before the matching wait call. However, such **strong** ordering is more than needed for correct ordering of window accesses. MPI also supports **weak** synchronization, as illustrated in Figure 5.2. The access to the target window is delayed until the window is available for RMA access, after the post call; however, the start and put calls, may complete earlier. The synchronization calls order correctly window accesses, but do not necessarily synchronize other operations. This weaker synchronization semantic allows for more efficient implementations.

Since both communicating processes are explicitly involved in the communication, a polling implementation of 2-party communication is possible, to the same extent that it is possible for message-passing.

Figure 5.3 illustrates the general synchronization pattern, for 3-party communication, i.e., for conflicts between RMA accesses. The first origin process communicates data to the second origin process, through the memory of the target process; the target process is not explicitly involved in the communication. The `lock` and `unlock` calls ensure that the two RMA accesses do not occur concurrently (however, they do not ensure that the put by origin 1 will precede the get by origin 2). Although the target process does not participate explicitly in the communication, its implicit participation may be required, especially in a distributed memory environment, in order to access the local memory and grant locks. 3-party communication will require, on distributed memory systems, an asynchronous "RMA agent" (signal handler, thread, daemon, etc.) at the target node that accepts RMA requests and performs the required read or write accesses in the memory of the target process. This agent has to be scheduled for execution on the target node even if the target process does

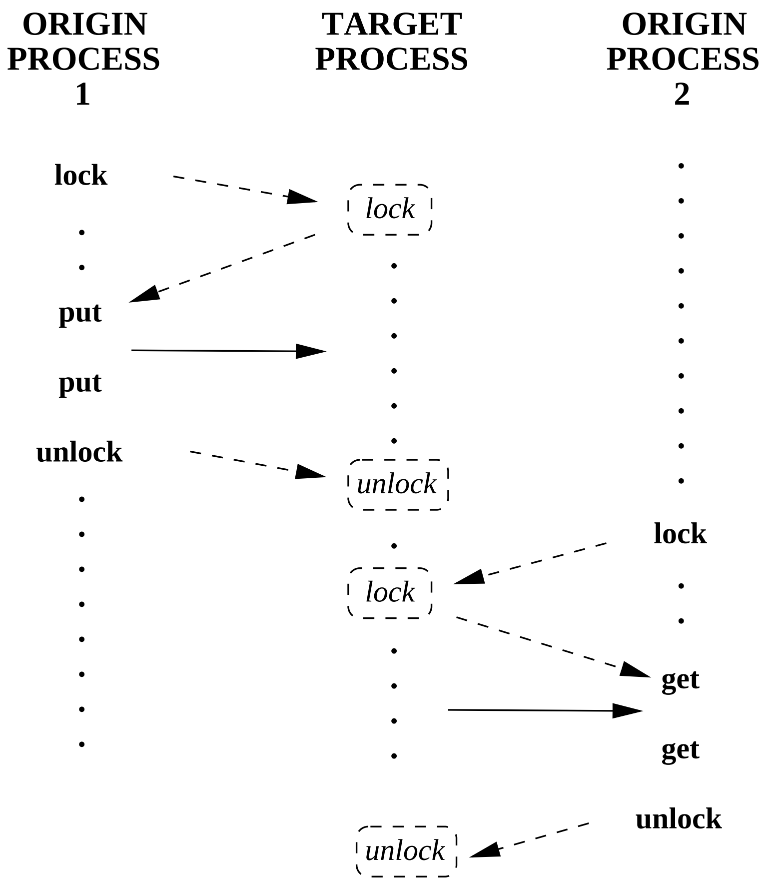

Figure 5.3: 3-party communication. Dashed arrows represent synchronizations and solid arrows represent data transfer.

not execute any MPI call, other than MPI_INIT and MPI_FINALIZE.

5.4.1 General 2-party synchronization

MPI_WIN_START(group, assert, win)

IN	group	group of target processes (handle)
IN	assert	program assertion (integer)
IN	win	window object (handle)

```
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
```

```
MPI_WIN_START(GROUP, ASSERT, WIN, IERROR)
    INTEGER GROUP, ASSERT, WIN, IERROR
```

Starts a sequence of RMA operations targeted to the windows associated with win at the target processes in the group of group. RMA accesses to each of these windows will be delayed, if necessary, until the target processes executed a matching call to MPI_WIN_POST.

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 5.4.4. A value of assert = 0 is always valid.

Discussion: We decided to use groups in start and post, rather than list of ranks, in order to reduce the overhead of argument copying. But we need now to check that the group members are a subset of the group of win. This test may be expensive, too.

MPI_WIN_COMPLETE(win)

IN	win	window object (handle)

```
int MPI_Win_complete(MPI_Win win)
```

```
MPI_WIN_COMPLETE(WIN, IERROR)
    INTEGER WIN, IERROR
```

Complete (at the origin) the sequence of RMA operations on the windows associated with win that were initiated after the last preceding call to MPI_WIN_START(group, assert, win). All these RMA calls should have targets in group.

Consider a sequence of calls in the example below.

Example 5.5

```
MPI_RMA_START(group, flag, win)
...
MPI_PUT(...,win)
...
MPI_RMA_COMPLETE(win)
```

The call to MPI_WIN_COMPLETE blocks until the put call has completed at the origin; and the target window will be accessed by the put operation only if the call to MPI_WIN_START was matched by a call to MPI_WIN_POST by the target process. It is up to the implementation whether the first two calls block. An implementation may have the call to MPI_WIN_START block until the matching call to post occurred; or it may allow the call to MPI_WIN_START to proceed, and block the call to MPI_PUT, until the matching call to post occurred; or, it may allow both calls to proceed and delay the access to the target window until the call to MPI_WIN_COMPLETE occurred and until the matching call to post occurred. Thus, part, or all, of the sequence of instructions start, put, complete may block on the matching post call at the target. However, once this call to post was issued, the sequence will complete, without further dependencies.

MPI_WIN_COMPLETE enforces completion of preceding RMA calls at the origin. A put or accumulate call may not have completed at the target when it has completed at the origin.

Each call to MPI_WIN_START(.., win) has to be matched by a unique following call to MPI_WIN_COMPLETE(win), and vice versa.

MPI_WIN_POST(group, assert, win)

IN	group	group of origin processes (handle)
IN	assert	program assertion (integer)
IN	win	window object (handle)

```
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
```

```
MPI_WIN_POST(GROUP, ASSERT, IERROR)
    INTEGER GROUP, ASSERT, WIN, IERROR
```

Post the local window associated with win. This call will match calls to MPI_WIN_START by the origin processes in group, thus allowing RMA calls by these processes to access the local window.

MPI_WIN_WAIT(win)

IN	win	window object (handle)

```
int MPI_Win_wait(MPI_Win win)
```

```
MPI_WIN_WAIT(WIN, IERROR)
    INTEGER WIN, IERROR
```

Each call to MPI_WIN_POST(group, assert, win) is followed by a call to MPI_WIN_WAIT(win). The former call matches a call to MPI_WIN_START(..., win) issued on each of the origin processes listed in ranks. The later call matches the following call to MPI_WIN_COMPLETE(win) issued by each of these origin processes. The call to MPI_WIN_WAIT will block until all these matching calls to MPI_WIN_COMPLETE 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 have completed at the target

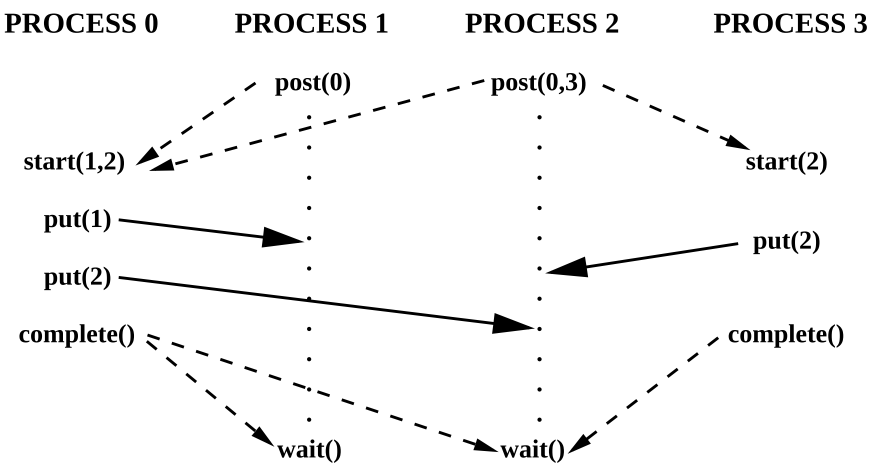

Figure 5.4: 2-party communication. Dashed arrows represent synchronizations and solid arrows represent data transfer.

window.

Each call to MPI_WIN_POST(group, assert, win) has to be matched by a unique following call to MPI_WIN_WAIT(win), and vice versa.

Figure 5.4 illustrates the use of these four functions. Process 0 puts data in the windows of processes 1 and 2 and process 3 puts data in the window of process 2. Each start call lists the ranks of the processes where windows are accessed; each post call lists the ranks of the processes that access the windows. The figure illustrates a possible timing for the events, assuming strong synchronization; in a weak synchronization, the start call may occur ahead of the matching post calls.

MPI_WIN_TEST(win, flag)

IN	win	window object (handle)
OUT	flag	success flag (Boolean)

```
int MPI_Win_test(MPI_Win win, int *flag)
```

```
MPI_WIN_TEST(WIN, FLAG, IERROR)
    INTEGER WIN, IERROR
    LOGICAL FLAG
```

This is the nonblocking version of MPI_WIN_WAIT. It returns flag = true if MPI_WIN_WAIT would return, flag = false, otherwise. The effect of return of MPI_WIN_TEST with flag = true is the same as the effect of a return of MPI_WIN_WAIT. If flag = false is returned, then the call has no visible effect.

MPI_WIN_TEST should be invoked only where MPI_WIN_WAIT can be invoked. Once the call has returned flag = true, it should not be invoked anew, until the window is posted anew.

> *Rationale.* The current design for general 2-party synchronization requires the user to provide complete information on the communication pattern, at each end of a communication link: each origin specifies a list of targets, and each target specifies a list

of origins. This provides maximum flexibility (hence, efficiency) for the implementor: each synchronization can be initiated by either side, since each "knows" the identity of the other. This also provides maximum protection from possible races. On the other hand, the design requires more information than RMA needs, in general: in general, it is sufficient for the origin to know the rank of the target, but not vice versa. Users that want more "anonymous" communication will be required to use the barrier or lock mechanisms. (*End of rationale.*)

Advice to implementors. We outline two possible implementations:

"Push" protocol: the call to MPI_WIN_POST(group, assert, win) causes a message to be sent to each origin process listed in ranks (unless nocheck is specified by assert). This signal allows RMA communication to occur. The origin processes signal back completion when their call to MPI_WIN_COMPLETE occur (the later message may be piggy-backed on data). The call to MPI_WIN_WAIT blocks until all these signals are received.

"Pull" protocol: the call to MPI_WIN_POST(group, assert, win) causes the target window to be marked at the target as posted for the processes in group. The RMA calls from each origin process have to be preceded by a test of the mark. This can imply a full handshake, preceding the RMA data transfer; or, if RMA accesses are executed by an agent at the target, and buffering is acceptable, an eager protocol can be used whereby RMA data is sent and the test occurs at the target (this is similar to the short and long send protocols). The origin processes signal completion when their call to MPI_WIN_COMPLETE occurs. The window is marked as "unposted" for the signalling process. The call to MPI_WIN_WAIT blocks until the window is not posted anymore for any process. (*End of advice to implementors.*)

5.4.2 Barrier

MPI_WIN_BARRIER(assert, win)

IN	assert	program assertion (integer)
IN	win	window object (handle)

```
int MPI_Win_barrier( int assert, MPI_Win win)
```

```
MPI_WIN_BARRIER(ASSERT, WIN, IERROR)
    INTEGER ASSERT, WIN, IERROR
```

The MPI call MPI_WIN_BARRIER(assert, win) synchronizes RMA calls on win. The call is collective on the group of win. All RMA operations on win that were started before the barrier call will be completed at their origin before the barrier call returns at the origin. They will be completed at their target before the barrier call returns at the target. RMA operations on win started by a process after the barrier call returned will access their target window only after MPI_WIN_BARRIER was called by the target process.

5.4.3 Lock

MPI_WIN_LOCK(lock_type, rank, assert, win)

IN	lock_type	either MPI_LOCK_EXCLUSIVE or MPI_LOCK_SHARED
IN	rank	rank of locked window (integer)
IN	assert	program assertion (integer)
IN	win	window object (handle)

```
int MPI_Win_Lock(int lock_type, int rank, int assert, MPI_Win win)
```

```
MPI_WIN_LOCK( LOCK_TYPE, RANK, ASSERT, WIN, IERROR)
    INTEGER RANK, LOCK_TYPE, RANK, ASSERT, WIN, IERROR
```

Starts a sequence of RMA accesses to the window associated with (win, rank). The call also ensures that any RMA access to the window associated with (win, rank) that was protected by a previously released, conflicting lock will have completed at the target.

MPI_WIN_UNLOCK(rank, win)

IN	rank	rank of window (integer)
IN	win	window object (handle)

```
int MPI_Win_unlock(int rank, MPI_Win win)
```

```
MPI_WIN_UNLOCK(RANK, WIN, IERROR)
    INTEGER RANK, WIN, IERROR
```

Completes at the origin a sequence of RMA accesses to the window associated with (win, rank).

RMA accesses that are protected by an exclusive lock will not be concurrent at the target with other RMA accesses to the same window. RMA accesses that are protected by a shared lock will not be concurrent at the target with accesses protected by an exclusive lock to the same window.

Consider the sequence of calls in the example below.

Example 5.6

```
MPI_LOCK(rank, MPI_EXCLUSIVE, assert, win)
MPI_PUT(..., rank, ..., win)
MPI_UNLOCK(rank, win)
```

The call to MPI_WIN_UNLOCK will block until the put transfer completed at the origin. It is up to the implementation whether the lock call is blocking. An implementation may have the call to MPI_WIN_LOCK block until an exclusive lock on the window is acquired. It could postpone lock acquisition until the call to MPI_PUT occurs; or it could postpone lock acquisition and the update to the window until the call to MPI_WIN_UNLOCK occurs.

Locks can also be used to protect local load/store accesses to a window.

Implementors may restrict the use of RMA communication that is synchronized by lock calls to windows in memory allocated by MPI_MEM_ALLOC. Locks can be used portably only in such memory.

Rationale. The implementation of 3-party communication when memory is not shared requires an asynchronous agent. Such agent can be implemented more easily, and can achieve better performance, if restricted to specially allocated memory. It can avoided altogether if shared memory is used. It seems natural to impose restrictions that allows one to use shared memory for 3-rd party communication in shared memory machines.

The downside of this decision is that 3-party communication cannot be used without taking advantage of nonstandard Fortran features: namely, the availability of C-like pointers; these are not supported by some Fortran compilers (g77 and Windows/NT compilers, at the time of writing). Also, 3-party communication cannot be targeted to COMMON blocks, or other statically declared Fortran arrays. (*End of rationale.*)

5.4.4 Assertions

The assert argument in the calls MPI_WIN_POST, MPI_WIN_START, MPI_WIN_BARRIER and MPI_WIN_LOCK is used to provide assertions on the context of the call that may be used to optimize performance. The assert argument does not change program semantics if it provides correct information on the program – it is erroneous to provides incorrect information. Users may always provide assert = 0 to indicate a general case, where no guarantees are made.

Advice to users. Many implementations may not take advantage of the information in assert; some of the information is relevant only for noncoherent, shared memory machines. Users should consult their implementation manual to find which information is useful on each system. On the other hand, applications that provide 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 three integer constants: MPI_NOCHECK, MPI_NOSTORE and MPI_NOPUT. The significance of each constant is described below.

Advice to users. C/C++ Users can use bit vector or (|) to combine these constants; Fortran 90 users can use the bit-vector OR intrinsic. Fortran 77 users can use (non-portably) bit vector OR on systems that support it. Alternatively, Fortran users can portably use integer addition to OR the constants (each constant should appear at most once in the addition!). (*End of advice to users.*)

MPI_WIN_START:

MPI_NOCHECK - The matching calls to MPI_WIN_POST have already completed on all target processes when the call to MPI_WIN_START is made. This is similar to the optimization of "ready-receive" that may save a handshake when the handshake is implicit in the code, anyhow.

MPI_NOSTORE - not significant.

MPI_NOPUT - not significant.

MPI_WIN_POST:

MPI_NOCHECK - The nocheck information provided by MPI_WIN_POST should be the same as the information provided by the matching MPI_WIN_START calls.

MPI_NOSTORE - the local window was not updated by local stores since last synchronization. This may avoid the need for cache synchronization at the post call.

MPI_NOPUT - the local window will not be updated by put or accumulate calls after the post call, until the ensuing (win_wait) synchronization. This may avoid the need for cache synchronization at the wait call.

MPI_WIN_BARRIER:

MPI_NOCHECK - not significant.

MPI_NOSTORE - the local window was not updated by local stores since last synchronization.

MPI_NOPUT - the local window will not be updated by put or accumulate calls after the barrier call, until the ensuing (win_barrier) synchronization.

MPI_WIN_LOCK:

MPI_NOCHECK - No other process holds, or will attempt to acquire a conflicting lock, while the caller holds the window lock. This is useful when mutual exclusion is achieved by other means, but the coherence operations that may be attached to the lock and unlock calls are still required.

MPI_NOSTORE - not significant

MPI_NOPUT - not significant

Advice to users. Note that the nostore flag provides information on what happened *before* the call; the noput flag provides information on what will happen *after* the call. (*End of advice to users.*)

5.5 Examples

Example 5.7 The following example shows a generic loosely synchronous, iterative code, using barrier 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_Barrier(0, win);
   for(i=0; i < toneighbors; i++)
      MPI_Put(frombuf(i), 1, fromtype(i), toneighbor(i),
                         todisp(i), 1, totype(i), win);
   MPI_Barrier(MPI_NOSTORE, win);
   }
```

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 buffers.

Example 5.8 Same generic example, with more computation/communication overlap. We assume that the update phase is broken in two subphases: the first, where the "boundary", which is involved in communication, is updated, and the second, where the "core", which neither use nor provide communicated data, is updated.

```
...
while(!converged(A)){
   update_boundary(A);
   MPI_Win_barrier(MPI_NOPUT, win);
   for(i=0; i < fromneighbors; i++)
      MPI_Get(tobuf(i), 1, totype(i), fromneighbor(i),
                     fromdisp(i), 1, fromtype(i), win);
   update_core(A);
   MPI_Win_barrier(0, win);
   }
```

The get communication can be concurrent with the core update, since they do not access the same locations, and the local update of the origin buffer by the get call can be concurrent with the local update of the core by the `update_core` call. In order to get similar overlap with put communication we would need to use separate windows for the core and for the boundary.

Example 5.9 Same code as in Example 5.7, rewritten using post-start-complete-wait. We assume that each process has a list of processes that access its window.

```
...
while(!converged(A)){
   update(A);
   MPI_Win_post(fromgroup, 0, win);
   MPI_Win_start(togroup, 0, win);
   for(i=0; i < toneighbors; i++)
         MPI_Put(frombuf(i), 1, fromtype(i), toneighbor(i),
                         todisp(i), 1, totype(i), win);
   MPI_Win_complete(win);
   MPI_Win_wait(win);
   }
```

Example 5.10 Same example, with split phases, as in Example 5.8.

```
...
while(!converged(A)){
   update_boundary(A);
   MPI_Win_post(togroup, 0, win);
   MPI_Win_start(fromgroup, 0, win);
   for(i=0; i < fromneighbors; i++)
         MPI_Get(tobuf(i), 1, totype(i), fromneighbor(i),
                     fromdisp(i), 1, fromtype(i), win);
   update_core(A);
   MPI_Win_complete(win);
   MPI_Win_wait(win);
   }
```

Example 5.11 A checkerboard, or double buffer communication pattern, that allows more computation/communication overlap. Array `A0` is updated using values of array `A1`, and vice versa. We assume that communication is symmetric: if process A gets data from process B, then process B gets data from process A. Window `wini` consists of array `Ai`.

```
...
MPI_Win_post(neighbors, MPI_NOCHECK, win0);
MPI_Barrier(comm0);
! the barrier is needed because the start call inside the
! loop uses the nocheck option
while(!converged(A0, A1)){
   /* communication on A0 and computation on A1 */
   update(A1, A0) /* local update of A1 that depends on A0 (and A1) */
   MPI_Win_start(neighbors, MPI_NOCHECK, win0);
   for(i=0; i < neighbors; i++)
          MPI_Get(tobuf0(i), 1, totype0(i), neighbor(i),
                         fromdisp0(i), 1, fromtype0(i), win0);
   update(A1); /* local update of A1 that
                  concurrent with communication that updates A0 */
   MPI_Win_post(neighbors, MPI_NOCHECK, win1);
   MPI_Win_complete(win0);
   MPI_Win_wait(win0);

   /* communication on A1 and computation on A0 */
   update(A0, A1) /* local update of A0 that depends on A1 (and A0)*/
   MPI_Win_start(neighbors, MPI_NOCHECK, win1);
   for(i=0; i < neighbors; i++)
         MPI_Get(tobuf1(i), 1, totype1(i), neighbor(i),
                    fromdisp1(i), 1, fromtype1(i), win1);
   update(A0); /* local update of A0 that depends on A0 only,
                  concurrent with communication that updates A1 */
   MPI_Win_post(neighbors, MPI_NOCHECK, win0);
   MPI_Win_complete(win1);
   MPI_Win_wait(win1);
   }
MPI_Win_complete(win0);
```

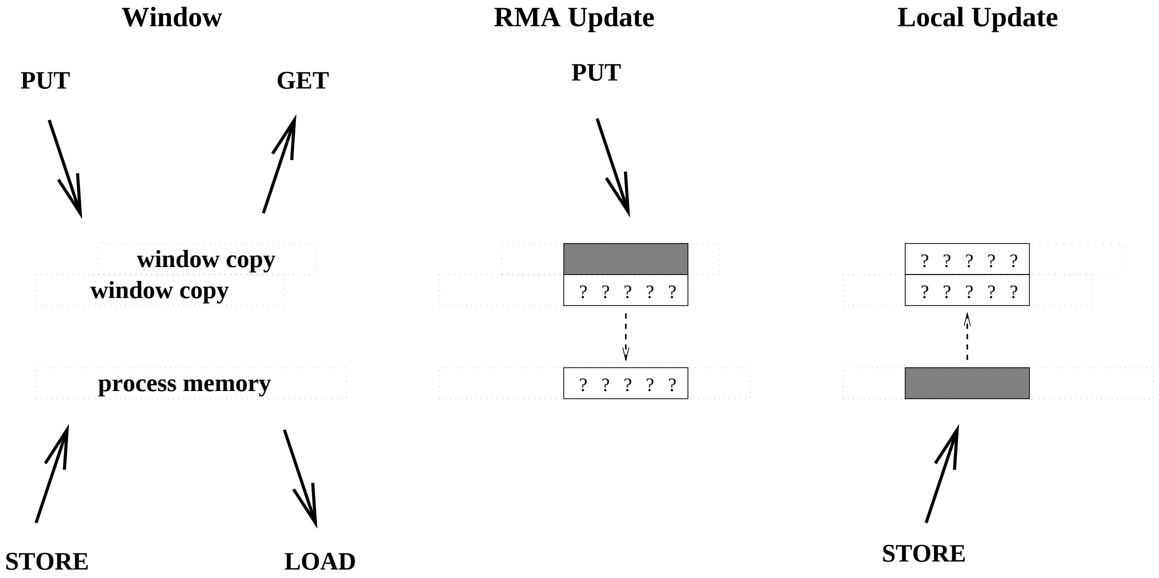

Figure 5.5: Schematic description of window

A process posts the local window associated with `wini` before it completes RMA accesses to the remote windows associated with `win(1-i)`. When the `wait(wini)` call returns, then all neighbors of the calling process have posted the windows associated with `win(1-i)`. Therefore, no check is needed on the following MPI_WIN_START call.

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.

5.6 Semantics and Correctness

Current Status: Passed once

The semantics of RMA operations is best understood by assuming that the system maintains a separate copy of each window. There is only one *private* copy of each variable in process memory, but a distinct *public* copy of the variable for each window that contains it. A load accesses the private copy (this includes MPI sends). A store updates the private copy (this includes MPI receives), but the update may affect the public copy (or copies) of the same locations. A get on a window accesses the public copy associated with that window. A put or accumulate on a window updates the public copy associated with that window, but the update may affect the process memory copy of the same locations, and other window copies. This is illustrated in Figure 5.5.

The following rules specify the latest time at which an operation must complete, and the latest time at which an update of one window copy becomes visible in the other copy.

1. An RMA operation is completed in process memory at the origin by the ensuing call to MPI_WIN_COMPLETE, MPI_WIN_BARRIER or MPI_WIN_UNLOCK that synchronizes this access at the origin.

2. If an RMA operation is completed at the origin by a call to MPI_WIN_BARRIER then the operation is completed in the window copy at the target by the matching call to MPI_WIN_BARRIER 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 in the window copy at the target by the matching call to MPI_WIN_WAIT by the target process.

4. If an operation is completed at the origin by a call to MPI_WIN_UNLOCK then the operation is completed in the window copy at the target by a subsequent call to MPI_WIN_LOCK on the window by any process, or by a call to MPI_WIN_BARRIER on the window by the target process.

5. An update of a location in process memory becomes visible in a window copy of it at latest when an ensuing call to MPI_WIN_POST, MPI_WIN_BARRIER, or MPI_WIN_UNLOCK is executed on that window by the window owner.

6. An update by a put or accumulate call to a window copy location becomes visible in process memory at latest when an ensuing call to MPI_WIN_WAIT, MPI_WIN_BARRIER, or MPI_WIN_LOCK is executed on that window by the window owner.

The MPI_WIN_BARRIER or MPI_WIN_WAIT call that completes the transfer from window copy to process memory (6) is the same call that completes the put or accumulate operation in the window copy (2, 3). However, if a put or accumulate access was synchronized with a lock, then the update of the window copy is complete as soon as *any* process executes a call to MPI_WIN_LOCK on that window (4); on the other hand, the update of the process memory may be delayed until the target process executes a synchronization call on that window (6). Thus, updates to process memory can always be delayed until the process executes a suitable synchronization call. Updates to window copy can also be delayed until the window owner executes a synchronization call, if barriers or post-start-complete-wait synchronization is used. Only when lock synchronization is used it becomes necessary to update the window copy even if the window owner does not execute any related synchronization call.

A correct program must obey the following rules.

1. A location in a window cannot be accessed locally once an update to that location has started, until the update becomes visible in process memory.

2. A location in a window cannot be accessed as a target of an RMA operation once an update to that location has started, until the update becomes visible in the window copy. There is one exception to this rule, in the case where the same variable is updated by two concurrent accumulate operations that use the same operation, with the same basic datatype.

3. A put or accumulate cannot update a target window once a local update or a put or accumulate update that uses another (overlapping) window have started on a location in the target window; this, until the second update becomes visible in the window copy. Conversely, a local update to a location in a window cannot start once a put or accumulate update to that target window has started; this, until the second update becomes visible in process memory. In both cases, the restriction applies to operations even if they access disjoint locations in the window.

A program is erroneous if it violates these rules.

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 where locally updated by stores and for locations that where remotely updated by put or accumulate operations. Without this constraint, the MPI library will have to track precisely which locations in a window where updated by a put or accumulate call. The additional overhead of maintaining such information is considered prohibitive. (*End of rationale.*)

Advice to users. A user can write correct programs by following the following rules:

post-start-complete-wait: A window should not be updated locally while being posted, if it is being updated by put or accumulate calls. Locations updated by put or accumulate calls should not be accessed while the window is posted. Locations accessed by get calls should not be updated while the window is posted.

barrier: During each period between barrier calls, each window is either updated by put or accumulate calls, or updated by local stores, but not both. Locations updated by put or accumulate calls should not be accessed during the same period. Locations accessed by get calls should not be updated during the same period.

lock: Updates to the window are protected by exclusive locks and read-only accesses to the window are protected by shared locks; this, both for local 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 overlaping 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_BARRIER, if RMA accesses to the window are synchronized with barriers; after a local call to MPI_WIN_WAIT, if the accesses are synchronized with post-start-complete-wait; and after a local call to MPI_WIN_UNLOCK or MPI_WIN_BARRIER 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. And two concurrent RMA operations should not access the same location in a target window, unless both are get operations, or both are accumulate operations that use the same window, and update the same location, with the same operation and the same basic datatype. (*End of advice to users.*)

5.6.1 Atomicity

The outcome of concurrent accumulates to the same location, with the same operation and basic datatype, is as if the accumulates where done at that location in some serial order. On the other hand, if two locations are both updated by two accumulate calls, then the updates

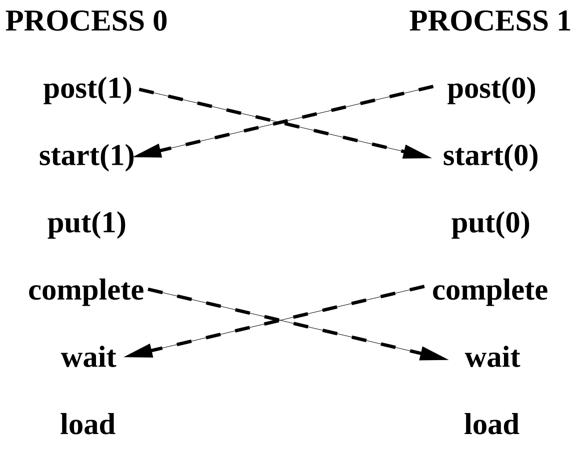

Figure 5.6: Symmetric communication

may occur in reverse order at the two locations. Thus, there is no guarantee that the entire call to MPI_ACCUMULATE is executed atomically. The effect of this lack of atomicity is limited: The previous correctness conditions imply that a location updated by a call to MPI_ACCUMULATE, cannot be accessed by load or an RMA call other than accumulate, until the MPI_ACCUMULATE call has completed (at the target). Different interleavings can lead to different results only to the extent that computer arithmetics are not truly associative or commutative.

5.6.2 Progress

One-sided communication has the same progress requirements as point-to-point communication: once a communication is enabled, then it is guaranteed to complete. RMA calls should not block, except when required for synchronization with other RMA calls.

Consider the code fragment in Example 5.5, on page 87. Some of the calls may block if the target window is not posted. However, if the target window is posted, then the code fragment must complete.

Consider the code fragment in Example 5.6, on page 91. Some of the calls may block if another process holds a conflicting lock. However, if no conflicting lock is held, then the code fragment must complete.

Consider the code fragment in Figure 5.6. Each process updates the window of the other process using a put operation, next accesses its own window. The post calls are nonblocking, and should complete. Once the post calls occurred, RMA access to the windows is enabled, so that each process should complete the sequence of calls start-put-complete. Once these are done, the wait calls should complete at both processes. Thus, this communication should not deadlock, irrespective of the amount of data transfered. Assume, in the last example, that the order of the post and start calls is reversed, at each process. Then, the code may deadlock, as each process may block on the start call, waiting for the matching post to occur.

5.6.3 Registers and Compiler Optimizations

Advice to users. All the material in this section is an advise to users. (*End of advice to users.*)

A coherence problem exists between variables kept in registers and the memory value of these variables. An RMA call may access a variable in memory (or cache), while the up-to-date value of this variable is in register. A get will not return the latest variable value, and a put may be overwritten when the register is stored back in memory.

The problem is illustrated by the following code:

Source of Process 1	**Source of Process 2**	**Executed in Process 2**
bbbb = 777	buff = 999	reg_A:=999
call MPI_PUT(bbbb		stop appl. thread
into buff of process 2)		buff:=777 in PUT handler
		continue appl. thread
call MPI_WIN_BARRIER	call MPI_WIN_BARRIER	
	ccc = buff	ccc:=reg_A

In this example, variable buff is allocated in the register reg_A and therefore ccc will have the old value of buff and not the new value 777.

This problem, which also afflicts in some cases send/receive communication, is discussed more at length in Section 2.5.2.

MPI implementations will avoid this problem for standard conforming C programs. Some Fortran compilers may provide mechanisms to avoid this problem, without disabling compiler optimizations. However, In order to avoid register coherence problems in a portable manner, users should restrict their use of RMA windows to variables stored in COMMON blocks, or to variables that were declared VOLATILE (while VOLATILE is no a standard Fortran declaration, it is supported by many Fortran compilers). Details, and an additional solution are discussed in Section 2.5.2 on page 10.

5.6.4 Implementation model

Advice to users. Users can ignore this section. (*End of advice to users.*)

Advice to implementors. All the material in this section is an advise to implementors (*End of advice to implementors.*)

Distributed Memory

The data transfer effected by a put can be implemented in a distributed memory system by a matching pair of send and receive. The origin process sends the put request using a call to MPI_ISEND, to the target process. The request includes the target displacement, count and datatype arguments. The origin next sends the data using a call to MPI_ISEND, with the origin address, count, and datatype arguments. The nonblocking sends are completed by a wait posted when the RMA completion call MPI_WIN_COMPLETE, MPI_WIN_BARRIER or MPI_WIN_UNLOCK occurs. The target process receives the put request and posts a corresponding receive, using the target displacement, count and datatype arguments. A similar scheme works for the other RMA transfers.

The only nontrivial part of this scheme is to ensure that the target process receives the put request and acts on it, posting a receive for the data, within a bounded amount of time.

A pure polling implementation can be used for 2-party communication, to the same extend that a pure polling implementation can be used for send-receive communication.

Such pure polling implementation requires that the target process polls for pending RMA requests when it executes any MPI call, and that it polls for pending RMA requests while blocked on any MPI call. This scheme guarantees that a put request will be eventually processed, and that deadlocks are not introduced by the implementation.

As for send-receive, such pure polling implementation has the defect that, even if a communication is enabled, the communication may not complete until the target process executes some MPI call. Such a call is bound to occur, unless the target process is caught in an infinite loop, and is likely to occur soon, since the target process is involved in the communication, and cannot access the target window before it executes an MPI synchronization call. However, the delay until this call occurs can be arbitrarily large. This is similar to the case of a process that posts a nonblocking receive, but delays the ensuing wait or test, and any other MPI call, for an arbitrarily long time. The situation here is worse: one will have the counterintuitive effect that the completion of a get can be delayed until the owner of the window that is read reuses this window.

Implementations can ensure that an enabled communication will complete within a time bound that is independent of the state of other processes by replacing or supplementing polling by asynchronous agents, e.g., by using interrupt receives, or by using a timer signal to enforce an upper bound on the waiting time for polling.

A pure polling strategy is clearly unacceptable for 3-party communication, since the target process may never execute MPI calls (MPI_FINALIZE excepted). This case requires an asynchronous agent (signal handler, polling thread, etc.) that processes pending requests within a bounded amount of time. A good quality implementation will guarantee a low bound.

Shared Memory

Put can be implemented in a homogeneous shared memory environment by code executed by the calling process. Suppose one has, in this environment, an implementation of send/receive, using memory-to-memory copy. The put operation can be implemented as a "send" from origin memory to an intermediate buffer, implemented as memory-to-memory copy, followed by a "receive" from the intermediate buffer to the target memory, implemented again using memory-to-memory copy. The intermediate copy can be avoided (or pipelined) for simple (contiguous) buffers or with a more elaborate implementation. On the other hand, additional copying may be required to avoid alignment problems. Similar techniques can be used for other systems that support some form of copying from local to remote memory (node-to-node DMA, reflective memory, etc.). Progress is not an issue in such implementations.

An additional memory-to-memory copy by the target process is required if both processes have access to shared memory, but the origin process cannot access the target window. This copying need be effected in a timely fashion, using similar mechanisms as discussed for distributed memory implementations.

Various weak memory models introduce the following complexities:

Ordering If memory is not serially consistent, explicit calls are needed to ensure that accesses by different processors are ordered correctly, at points that indicate completion of RMA operations.

Coherence In many of the target systems local load/store accesses are not necessarily coherent with remote put/get accesses. It may be the case that put/get access shared memory, and that local caches are not maintained coherent with memory. Or, that a window is in private memory, and a (noncoherent) mirror copy is kept in shared memory, where it can be accessed by other processes.

Atomicity The implementation must guarantee that accumulates are atomic.

Granularity Many target systems do not support atomic stores of arbitrarily sized entries. Load and stores may be restricted to words; and DMA engines may be restricted to cache line or page transfers.

The semantics of RMA operations allow us to associate coherence restoring and memory synchronization operations with RMA synchronization calls. Thus, RMA operations can be implemented efficiently on shared memory systems with weak memory semantics.

We first describe a generic implementation model, and then map this generic implementation onto concrete examples. We assume that the system maintains explicitly a copy of each window, as described in Section 5.6.

Ordering: The implementation must ensure that RMA accesses have completed in the window copy at those points in the execution where RMA operations are said to be complete at the target.

Coherence: When a (local) call to MPI_WIN_WAIT, MPI_WIN_BARRIER or MPI_WIN_LOCK occurs, previous updates to the window copy are made visible in the process memory, by copying the window copy into the process memory. Only updates made to the window copy since the copies were last made coherent need be propagated.

When a (local) call to MPI_WIN_POST, MPI_WIN_BARRIER, or MPI_WIN_UNLOCK occurs, previous updates to the process memory are made visible in the window copy, by copying from process memory to the window copy. Only updates made to the process memory since the copies were last made coherent need be propagated.

There is one difficulty: a call to MPI_WIN_BARRIER by the window owner can require copying in either direction. The correctness rules imply that when the barrier call is reached, either the process memory carries updates that need be propagated to the window copy, or the window copy carries updates that need be propagated to the process memory, but not both. In order to distinguish between the two, one may maintain a *dirty bit* that is set when the window is updated by a put or accumulate call, and reset when the copies are made coherent. A call to MPI_WIN_BARRIER requires copying of the window copy to the process memory if the bit is set, and copying in the reverse direction, otherwise.

The implementation is still correct if some updates propagate from copy to copy earlier than specified here. Thus, the window copies need not be distinct.

Atomicity

The atomicity of accumulate updates can be guaranteed by using an internal window lock. Since puts and gets cannot conflict with accumulates, there is no need to check this lock for regular put and get accesses. Finer locking is possible, with a more complex locking protocol.

Granularity

Suppose that RMA accesses are restricted to aligned words, and updates of smaller entities require a load-modify-store cycle. Then, correct behavior can be achieved by protecting partial word updates with an internal window lock (can be the same as for accumulate). Full word updates need not be lock protected. Finer lock granularity (and higher concurrency) is possible by using different locks for different parts of the window.

Indeed, conflicts may only be caused by concurrent RMA operations that update disjoint parts of the same word, using the same window. This, because RMA updates cannot be concurrent to local updates to the same window, or to updates to the same locations using another window; true conflicts between RMA operations are also disallowed. Thus, two conflicting accesses must be partial word accesses using the same window, and these are protected by the same lock. Partial word accumulates are also protected by that same lock (this is the case anyhow if the same lock is used for all accumulates and for partial word updates.)

An additional problem occurs if the window boundaries are not word aligned. If a word overlaps the window boundary, then a local access to the part of the word that is outside the window can be concurrent with an RMA access to the part of that word that is inside the window. This conflict is avoided if two copies of this word are maintained – using the window copy protocol described above.

A similar problem arises on systems with noncoherent caches, when the window boundary is not cache line aligned. A cache line that partially overlaps the window may become dirty and be written back, overwriting the outcome of a concurrent RMA to the part of the line inside the window. This conflict is avoided if two copies of the line are maintained.

We specialize now this generic algorithm to specific architectures, with noncoherent caches. There are three cases to consider.

1. RMA is implemented using non-coherent shared memory, with write-through noncoherent caches. Both local and remote writes go directly to memory, but both local and remote reads may go to cache.

2. RMA is implemented using non-coherent shared memory, with write-back noncoherent caches. Both local and remote accesses may go to cache, rather than memory.

3. RMA is implemented using a non-coherent DMA engine. Remote accesses go directly to memory, whereas local accesses (both reads and writes) may go to cache.

In all cases, the window copy content is defined by the content of the main memory, whereas the process memory content is defined by the content of the processor cache, or the content of the main memory, for locations that are not cached.

Completing an RMA update is achieved by flushing the cache of the calling processor, so that the update becomes visible in the window copy. Such flush call will be executed as part of calls to MPI_WIN_BARRIER, MPI_WIN_COMPLETE and MPI_WIN_UNLOCK. It is sufficient to flush dirty cache lines that overlap the window.

Copying of the process memory to the window copy is achieved by flushing the cache of the local processor. It is sufficient to flush dirty cache lines that overlap with the window.

Copying of the window copy into the process memory is achieved by invalidating the cache of the local processor. It is sufficient to invalidate the lines that overlap with the window, and are dirty in the public copy.

Window copies may also be used in a shared memory implementation where not all processes can access all memory. E.g., in the case where a window is in process private memory, one may use an window copy that is in shared memory and is accessible by all processes in the communication group. Or, the window copy could be in nonpaged memory, that is accessible to DMA, whereas the process memory is pageable. Then, the copying of one copy to another actually requires memory to memory copy, that can be done by the window owning process. Full window copying can be avoided by maintaining finer dirty bits. (Dirty bits for the window copy are maintained by tracking the locations accessed by put or accumulate calls – this can be part of the code for these calls. Dirty bits for the process memory are harder to maintain. One needs to use techniques similar to those used by software implementations of distributed shared memory: e.g., use page protection to detect local updates at page granularity.)

Chapter 6

Extended Collective Operations

Current Status: No votes on entire chapter

6.1 Introduction

Collective communication capabilities are here for MPI-2, covering these areas:

- Extension of MPI collective operations to intercommunicators,
- Non-blocking collective communication (asynchronous) of a limited form (defined strictly for barrier at present),
- Persistent collective communication (persistent handles) for both intracommunicators and intercommunicators. Blocking forms[1] of persistent operations are defined only.

  Persistent calls must match persistent calls across the group of the communicator, to support optimization.
- New or extended collective operations (*e.g.*, in-place, exclusive scan, etc). Only the so-called "easy cases" are addressed; that is, those where sent data is not overridden by received data on a single call.
- Point-to-point channels (for highest performance messaging) with a simple constructor, is offered.

6.2 Persistent collective operations

For each collective operation offered in MPI-2, a persistent variant is proposed here (early binding version). Implementations will not be significantly complicated by the presence of the early-binding versions, and may in fact choose to implement the standard versions in terms of the persistent forms. This section implies a large number of new functions, but no conceptually new capability for MPI-2. Enumeration of all the language bindings is not made here, but two representatives are offered.

[1]This chapter says nothing at present about offering blocking point-to-point persistent operations, which could be discussed in the Miscellany chapter.

This set of persistent operations applies to all collective operations of MPI, but it may not make sense to use persistent variations in cases where performance is not an issue.

For example:

```
MPI_BARRIER_INIT(comm, request);
...
MPI_DO(request); // blocking operation, defined in Misc. chapter
...
MPI_REQUEST_FREE(request);
```

This operation, and those with actual data motion, should be more optimizable than the straight non-blocking variety, because the number of outstanding non-blocking persistent handles on a communicator is a hint to the implementation about the potential concurrency and amount of safe collective communication space needed (*i.e.*, extra contexts). Hence, for situations where non-blocking collectives are used inside loops, the persistent varieties are preferable proposal's view) to the nonce forms. (Of course, the persistent forms, which have so-called thin interfaces, do not let the user change anything except for contents of buffers, so they are less flexible.

Importantly, the initialization of a persistent collective operation is collective.

The existence of persistent collective operations where data sizes and other arguments are frozen, should be much more optimizable, and amenable to poly-algorithmic selection. Currently, because arguments can change, and in some operations, need not be the same across all processes, it would be exhorbitantly expensive to try to choose the best algorithm on the fly. The persistent approach rewards reuse, and opens the door to implementations that study the collective problem on the MPI_*_INIT call, and then do it more efficiently at each MPI_DO invocation.

MPI_BCAST_INIT(buffer, count, datatype, root, comm, request)

INOUT	buffer	data buffer (choice)
IN	count	count of datatypes to be sent (integer)
IN	datatype	datatype making up buffer (handle)
IN	root	integer rank in group of root
IN	comm	communicator (handle)
OUT	request	request for collective operation (handle)

```
int MPI_Bcast_init(void *buffer, int count, MPI_Datatype datatype, int root,
             MPI_Comm comm, MPI_Request *request)
```

```
MPI_BCAST_INIT(BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERR)
    INTEGER BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERR
```

The semantics of this call are identical to broadcast, except that this call creates a broadcast persistent request without starting it. A subsequent start is needed, which will effect a loosely synchronous broadcast.

MPI_BARRIER_INIT(comm, request)

IN	comm	communicator (handle)
OUT	request	request for persistent operation (handle)

```
int MPI_Barrier_init(MPI_Comm comm, MPI_Request *request)
```

```
MPI_BARRIER_INIT(COMM, REQUEST, IERR)
    INTEGER COMM, REQUEST, IERR
```

The semantics of this call are identical to Barrier, except that a barrier is setup, but not started. A later call to MPI_Start will start a synchronous barrier using the persistent request.

6.3 Limited Non-Blocking Collective Communication

```
Official Vote: Question at October meeting:
  Should we have at least MPI_IBARRIER in chapter 6 (Extended Collective
  Operation)

  Vote: 12 yes / 9 no / 3 abstain
```

Consistent with a limited extension of MPI-1 into MPI-2, just the MPI-1 barrier is extended to provide a non-blocking variant, and this is done by supporting a use of non-blocking start on the persistent operation. In fact, no new calls are needed to support this operation:

```
MPI_BARRIER_INIT(comm, request);
...
MPI_START(request); // start non-blocking barrier
...
MPI_REQUEST_FREE(request);
```

Discussion: Future MPI versions could drop the semantic restriction on non-blocking starts of persisten collectives for other operations. This could also be considered on a case-by-case basis before MPI-2 is finalized.

6.3.1 Obtaining additional Non-Blocking Behavior in MPI-2

In MPI-1, users are meant to achieve other non-blocking collective functionality through one of several mechanisms:

- Generalized Request Technology,
- Threaded programs, working with a thread-compliant MPI implementation, and user-defined policy for caching extra communicators.

6.4 Intercommunicator Constructors

The current MPI interface provides only two intercommunicator construction routines ([19]):

- **MPI_INTERCOMM_CREATE**, creates an intercommunicator from two intracommunicators,
- **MPI_COMM_DUP**, duplicates an existing intercommunicator (or intracommunicator).

The other communicator constructors, MPI_COMM_CREATE and MPI_COMM_SPLIT, currently apply only to intracommunicators. These operations in fact have well-defined semantics for intercommunicators. One other intercommunicator constructor for partitioning intracommunicators into multiple intercommunicators is also proposed.

MPI_COMM_CREATE(comm_in, group, comm_out)

IN	comm_in	original communicator
IN	group	group of process to be in new communicator
OUT	comm_out	new communicator

The language bindings are identical to MPI-1, so are omitted here.
If comm_in is an intercommunicator, then the output communicator is also an intercommunicator where the local side consists only of those processes contained in group (see figure 6.1). The group argument should only contain those processes in the local group of the input intercommunicator that are to be a part of comm_out. If the group does not specify at least one process on the local side of the intercommunicator, or if the calling process is not included in the group, MPI_COMM_NULL is returned.

Example The following example illustrates how the first node in the "left" side of an intercommunicator could be joined with all members on the "right" side of an intercommunicator to form a new intercommunicator.

```
MPI_Comm  inter_comm, new_inter_comm;
MPI_Group local_group, group;
int       rank = 0; /* rank on left side to include in new inter-comm */

/* Construct the original intercommunicator: "inter_comm" */
...

/* Construct the group of processes to be in new intercommunicator */
if (/* I'm on the left side of the intercommunicator */) {
  MPI_Comm_group ( inter_comm, &local_group );
  MPI_Group_incl ( local_group, 1, &rank, &group );
}
else
  MPI_Comm_group ( inter_comm, &group );

MPI_Comm_create ( inter_comm, group, &new_inter_comm );
```

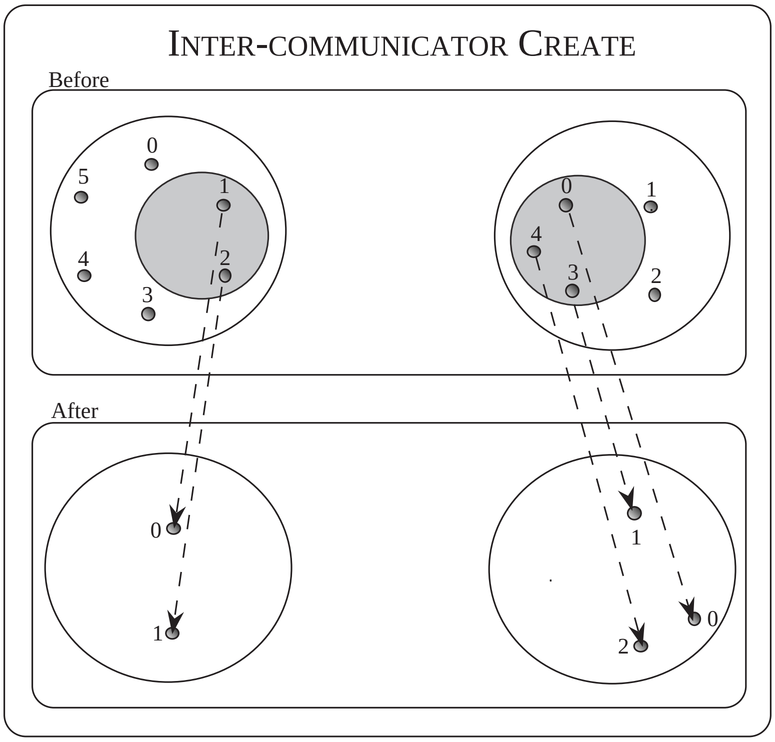

Figure 6.1: Intercommunicator create using MPI_COMM_CREATE() extended to intercommunicators.

MPI_COMM_SPLIT(comm_in, color, key, comm_out)

IN	comm_in	original communicator
IN	color	color of subset assignment
IN	key	color of rank assignment
OUT	comm_out	new communicator

The language bindings are identical to MPI-1, so are suppressed here.
The result of split on an intercommunicator is that those processes on the left with the same color as those processes on the right combine to create a new intercommunicator. The key argument describes the relative rank of processes on each side of the intercommunicator (see figure 6.2). For those colors that are specified only on one side of the intercommunicator but not the other, MPI_COMM_NULL is returned. MPI_COMM_NULL is also returned to those processes that specify MPI_UNDEFINED as the color.

Example (Parallel client-server model) The following client code illustrates how clients on the left side of an intercommunicator could be assigned to a single server from a pool of servers on the right side of an intercommunicator.

```
/* Client code */
MPI_Comm  multiple_server_comm;
MPI_Comm  single_server_comm;
int       color, rank, num_servers;
```

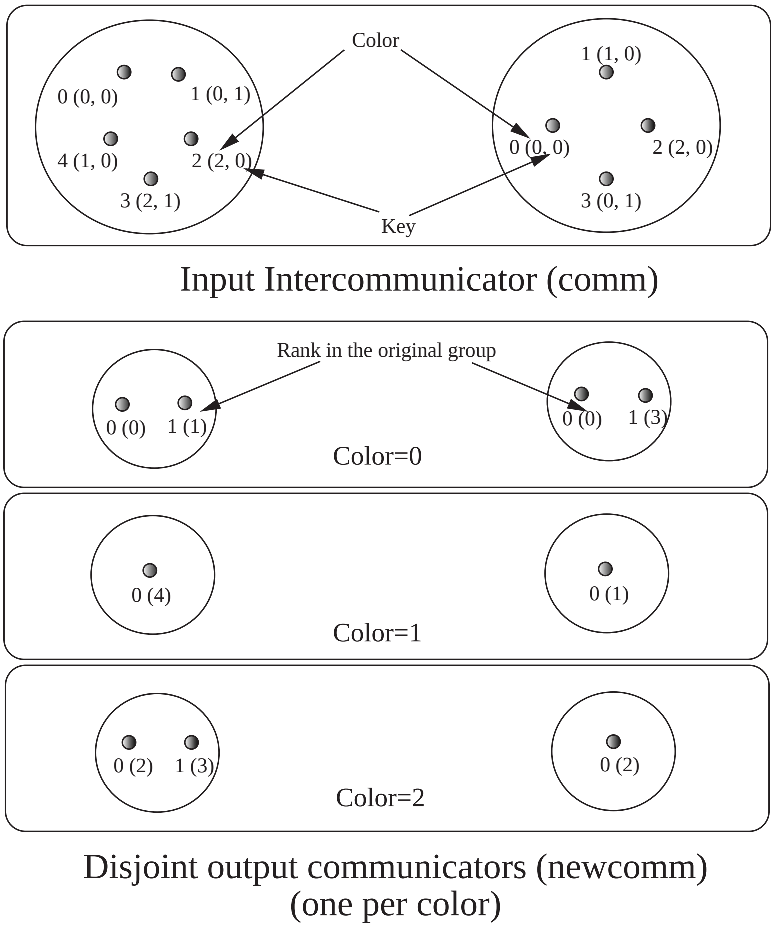

Figure 6.2: Intercommunicator construction achieved by splitting an existing intercommunicator with MPI_COMM_SPLIT() extended to intercommunicators.

```
    /* Create intercommunicator with clients and servers:
       multiple_server_comm */
    ...

    /* Find out the number of servers available */
    MPI_Comm_remote_size ( multiple_server_comm, &num_servers );

    /* Determine my color */
    MPI_Comm_rank ( multiple_server_comm, &rank );
    color = rank % num_servers;

    /* Split the intercommunicator */
    MPI_Comm_split ( multiple_server_comm, color, rank,
                     &single_server_comm );
```

The following is the corresponding server code:

```
    /* Server code */
    MPI_Comm  multiple_client_comm;
    MPI_Comm  single_client_comm;
    int       rank;

    /* Create intercommunicator with clients and servers:
       multiple_client_comm */
    ...

    /* Split the intercommunicator */
    MPI_Comm_rank ( multiple_client_comm, &rank );
    MPI_Comm_split ( multiple_client_comm, rank, 0,
                     &single_client_comm );
```

6.5 Extended Collective Operations

6.5.1 Intercommunicator Collective Operations

In the MPI-1 standard, collective operations only apply to intracommunicators [8, Section 4.2, pg. 90]; however, most MPI collective operations can be generalized to intercommunicators. To understand how MPI can be extended, we can view most MPI intracommunicator collective operations into one of the following categories (see, for instance, [19]):

All-To-All All processes contribute to the result. All processes receive the result.

- MPI_Allgather, MPI_Allgatherv
- MPI_Alltoall, MPI_Alltoallv
- MPI_Allreduce, MPI_Reduce_scatter

All-To-One All processes contribute to the result. One process receives the result.

- MPI_Gather, MPI_Gatherv

- MPI_Reduce

One-To-All One process contributes to the result. All processes receive the result.

- MPI_Bcast
- MPI_Scatter, MPI_Scatterv

Other Collective operations that do not fit into one of the above categories.

- MPI_Scan
- MPI_Barrier

The MPI_Barrier operation does not fit into this classification since no data is being moved. The data movement pattern of MPI_Scan does not fit this taxonomy.

Intercommunicators contain a "left" and a "right" group. Collective operations can be described in terms of these two groups. For example, an all-to-all MPI_Allgather operation can be described as collecting data from all members of one group with the result appearing in all members of the other group (see figure 6.3). 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 For intracommunicators, these two groups are the same. For intercommunicators, these two groups are distinct. For symmetry, each such operation is described in two phases, so that it has a full-duplex behavior.

For MPI-2, the following intracommuniator collective operations also apply to intercommunicators:

- MPI_BCAST,
- MPI_GATHER, MPI_GATHERV,
- MPI_SCATTER, MPI_SCATTERV,
- MPI_ALLGATHER, MPI_ALLGATHERV,
- MPI_ALLTOALL, MPI_ALLTOALLV, (MPI_ALLTOALLW)
- MPI_REDUCE, MPI_ALLREDUCE,
- MPI_REDUCE_SCATTER,
- MPI_BARRIER.

These functions use exactly the same argument list as their MPI-1 counterparts and also work on intracommunicators, as expected. No new language bindings are consequently needed.

6.5.2 Operations that Move Data

Full-duplex (or symmetric semantics) for collective intercommunicator operations have been chosen as the MPI-2 approach for such operations[2]

Two additions are made to all collective communication calls:

[2]Proposal by Marc Snir, IBM.

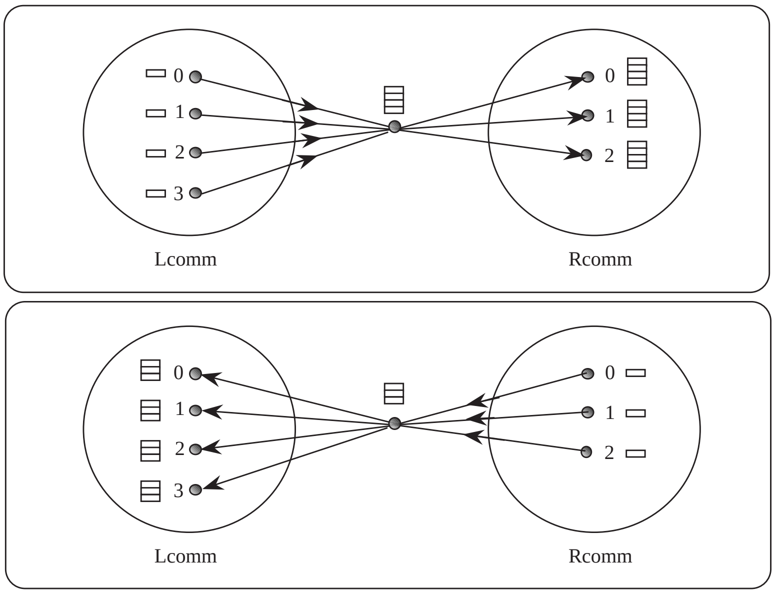

Figure 6.3: Intercommunicator allgather. The focus of data to one process is represented, not mandated by the semantics. The two phases do allgathers in both directions.

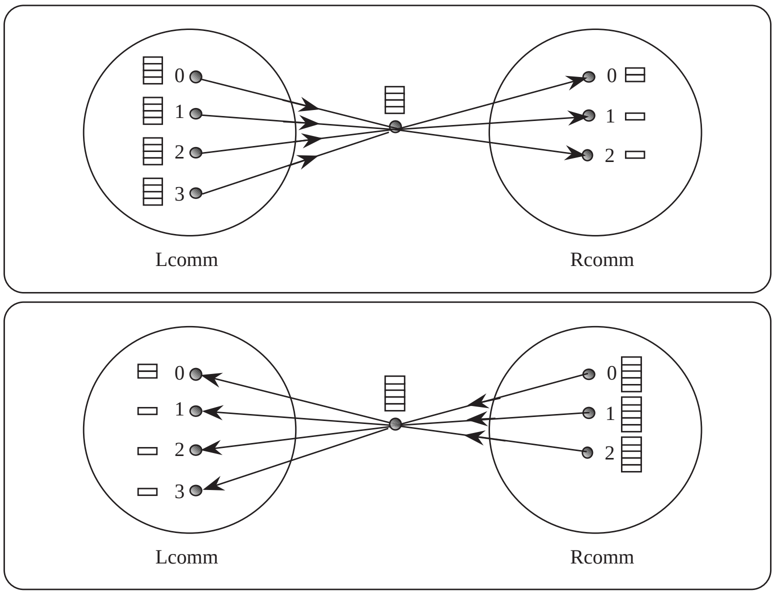

Figure 6.4: Intercommunicator reduce-scatter. The focus of data to one process is logical, not mandated by the semantics.

- Collective communication can occur "in place," with the output buffer being identical to the input buffer. This is specified by providing a special argument value, instead of the send buffer or the receive buffer argument.

- Collective communication applies to intercommunicators. If the operation is rooted (broadcast, gather, scatter), then the transfer is unidirectional. The direction of the transfer is indicated by a special value of the root argument. If the operation is unrooted (alltoall) then the transfer is bidirectional.

  Rationale. Rooted operations are unidirectional by nature, and there is a clear way of specifying direction. Non-rooted operations, such as all-to-all, will often occur as part of an exchange, where it makes sense to communicate in both directions at once. (*End of rationale.*)

Broadcast

MPI_BCAST(buffer, count, datatype, root, comm)

INOUT	buffer	starting address of buffer (choice)
IN	count	number of entries in buffer (integer)
IN	datatype	data type of buffer (handle)
IN	root	rank of broadcast root (integer)
IN	comm	communicator (handle)

The "in place" option not meaningful here. Inasmuch as the language bindings are identical to MPI-1, they are not repeated here.

If comm is an intercommunicator, then the call involves a root process in one group (group A) and all processes in the other group (group B). All processes in 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. Data is broadcast 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.

Gather

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

IN	sendbuf	starting address of send buffer (user's choice)
IN	sendcount	number of elements in send buffer (integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice, significant only at root)
IN	recvcount	number of elements for any single receive (integer, significant only at root)
IN	recvtype	data type of recv buffer elements (significant only at root) (handle)
IN	root	rank of receiving process (integer)
IN	comm	communicator (handle)

"In place" option 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 right place in the receive buffer

If comm is intercommunicator, then the call involves a root process in one group (group A) and all processes in the other group (group B). All processes in 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. Data is gathered from all processes in group B to the root. The send buffer arguments of the processes in group B must be consistent with the receive buffer argument of the root.

MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice, significant only at root)
IN	recvcounts	integer array (of length group size) containing the number of elements that are received from each process (significant only at root)
IN	displs	integer array (of length group size). Entry `i` specifies the displacement relative to recvbuf at which to place the incoming data from process `i` (significant only at root)
IN	recvtype	data type of recv buffer elements (significant only at root) (handle)
IN	root	rank of receiving process (integer)
IN	comm	communicator (handle)

The above argumentation extends to the MPI_GATHERV case.

Scatter

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

IN	sendbuf	address of send buffer (choice, significant only at root)
IN	sendcount	number of elements sent to each process (integer, significant only at root)
IN	sendtype	data type of send buffer elements (significant only at root) (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements in receive buffer (integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	root	rank of sending process (integer)
IN	comm	communicator (handle)

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

If comm is intercommunicator, then the call involves a root process in one group (group A) and all processes in the other group (group B). All processes in 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. Data is gathered from all processes in group B to the root. The send buffer arguments of the processes in group B must be consistent with the receive buffer argument of the root.

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

IN	sendbuf	address of send buffer (choice, significant only at root)
IN	sendcounts	integer array (of length group size) specifying the number of elements to send to each processor
IN	displs	integer array (of length group size). Entry `i` specifies the displacement (relative to `sendbuf` from which to take the outgoing data to process `i`
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements in receive buffer (integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	root	rank of sending process (integer)
IN	comm	communicator (handle)

The above argumentation applies.

"All" Forms and All-to-all

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

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements received from any process (integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)

The "in place" option is specified by passing the value MPI_IN-PLACE to the argument sendbuf at all processes. sendcount and sendtype are ignored. Then the input data of each process is assumed to be in the area where that process would receive its own contribution to the receive buffer.

If comm is an intercommunicator, then each process in group A contributes a data item; these items are concatenated and the result is stored at each process in group B.

Conversely the concatenation of the contributions of the processes in group B is stored at each process in group A. The send buffer arguments in group A must be consistent with the receive buffer arguments in group B, and vice versa.

The in-place option is illegal for intercommunicators.

Advice to users. The communication pattern of allgather executed on an intercommunication domain need not be symmetric. The number of items sent by processes in group A (as specified by the arguments sendcount, sendtype in group A and the arguments recvcount, recvtype in group B), need not equal the number of items sent by processes in group B (as specified by the arguments sendcount, sendtype in group B and the arguments recvcount, recvtype in group A). In particular, one can move data in only one direction, by specifying sendcount $= 0$ for the communication in the reverse direction.

(*End of advice to users.*)

No in-place is supported here.

MPI_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcounts	integer array (of length group size) containing the number of elements that are received from each process
IN	displs	integer array (of length group size). Entry `i` specifies the displacement (relative to recvbuf) at which to place the incoming data from process `i`
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)

Similar logic applies as above, with in-place illegal for intercommunicators.

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

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements sent to each process (integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements received from any process (integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)

No in-place is supported.

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

The in-place option is illegal for intercommunicators.

Advice to users. When all-to-all is executed on an intercommunication domain, then the number of data items sent from processes in group A to processes in group B need not equal the number of items sent in the reverse direction. In particular, one can have unidirectional communication, by specifying sendcount = 0 in the reverse direction.

(*End of advice to users.*)

MPI_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcounts	integer array equal to the group size specifying the number of elements to send to each processor
IN	sdispls	integer array (of length group size). Entry `j` specifies the displacement (relative to `sendbuf` from which to take the outgoing data destined for process `j`
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcounts	integer array equal to the group size specifying the number of elements that can be received from each processor
IN	rdispls	integer array (of length group size). Entry `i` specifies the displacement (relative to recvbuf at which to place the incoming data from process `i`
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)

Similar logic and restrictions apply as for the MPI_ALLTOALL case.

6.5.3 Reductions

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

IN	sendbuf	address of send buffer (choice)
OUT	recvbuf	address of receive buffer (choice, significant only at root)
IN	count	number of elements in send buffer (integer)
IN	datatype	data type of elements of send buffer (handle)
IN	op	reduce operation (handle)
IN	root	rank of root process (integer)
IN	comm	communicator (handle)

No in-place support.

If comm is intercommunicator, then the call involves a root process in one group (group A) and all processes in the other group (group B). All processes in 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. Only send buffer arguments are significant in group B and only receive buffer arguments are significant at the root.

MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	count	number of elements in send buffer (integer)
IN	datatype	data type of elements of send buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)

No in-place support.

If comm is 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 the same count value.

Discussion: We don't have the option of achieving unidirectional communication with intercommunicators, with the current definition. The same holds for reduce-scatter and scan.

MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcounts	integer array specifying the number of elements in result distributed to each process. Array must be identical on all calling processes.
IN	datatype	data type of elements of input buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)

The in place option is specified by passing MPI_IN-PLACE in the sendbuf argument. In this case, the input data is taken from the top of the receive buffer. Note that the area occupied by the input data may be either longer or shorter than the data filled 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 scattered among processes in group B, and vice versa. Within each group, all processes provide the same recvcounts argument, and the sum of the recvcounts entries should be the same for the two groups.

Discussion: The last restriction is needed so that the length of the send buffer be determined by the sum of the local recvcounts entries. Otherwise, a communication is needed to figure out how many elements are reduced. The in-place option can be used with intercommunicators.

Other operations

MPI_BARRIER(comm)

IN	comm	communicator (handle)

`int MPI_Barrier(MPI_Comm comm)`

For MPI-2, comm may be an intercommunicator or an intracommunicator.

MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	count	number of elements in input buffer (integer)
IN	datatype	data type of elements of input buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)

The "in place" option is specified by passing MPI_IN-PLACE in the sendbuf argument. In this case, the input data is taken from the receive buffer, and replaced by the output data.

This operation is illegal for intercommunicators.

6.5.4 Generalized All-to-all Operation

One[3] of the basic data movement operations needed in parallel signal processing is the 2-D matrix transpose. In this section, we show that the existing MPI function that support 2-D matrix transpose is not sufficiently general. To address this deficiency, we propose a new collective function. with "W" appended to its name. We specifically describe the MPI_ALLTOALLW function here, and leave other operations to be proposed for future MPI revisions.

Consider a 2-D matrix, which is initially distributed by striping rows over the available processors. After a transpose operation, it will be distributed by striping columns over the available processors. This basic data movement can be acheived by using the function MPI_ALLTOALL.

However, in addition to moving data between processors, the transpose operation should also change the layout of the matrix from row major order to column major order (*i.e.*, from elements in a row being contiguous in memory, to elements in a column being contiguous in memory). Thus in addition to moving data between processors, MPI_ALLTOALL should also transpose each "sub-cube."

For matrices whose dimensions are an even multiple of the number of processes, this sub-cube tranpose can easily be performed using the appropriate send and receive datatypes in the call to MPI_ALLTOALL. For example, if the source datatype steps throught the

[3] Proposed by Lloyd Lewins, Hughes Aircraft.

subarray elements row first, the destination datatype should step throught the subarray elements column first.

However, for matrices whose dimensions are not an even multiple of the number of processes a problem occurs. In this case, many of the sub-cubes will have the same dimensions, but one or more sub-cubes will have a different dimension. For example, a 18x18 matrix distributed over 4 processors would contain the following sub-cubes:

```
5x5 5x5 4x5 4x5
5x5 5x5 4x5 4x5
5x4 5x4 4x4 4x4
5x4 5x4 4x4 4x4
```

Even using the additional flexibility of MPI_ALLTOALLV it is impossible correctly to perform this transpose operation in one step. Consider the specification of the source datatype for this operation. The first two sub-cubes of the first row require a vector type, containing five blocks each containing five elements (with a stride between blocks of 18). In contrast, the vector type for the second two sub-cubes should only contain four elements. Unfortunately, MPI_ALLTOALLV does not allow an array of datatypes to be specified for the source buffer (only sizes and displacements). Thus MPI_ALLTOALLV is not sufficiently general to handle matrix tranposition.

NOTE: The destination datatype for the above operation can be correctly specified to MPI_ALLTOALLV. The first two sub-cubes of the first column require a vector type containing five blocks each containing one element. If the stride of this datatype is set to one (using MPI_TYPE_STRUCT) the first two sub-cubes can be specified by using a count of five. The second two sub-cubes of the first column can use the same datatype, so long as they specify a count of four.

6.5.5 Generalized All-to-all Function

The following function is the most general form of `All-to-all`. Like MPI_TYPE_STRUCT, the most general type constructor, MPI_ALLTOALLW allows seperate specification of count, displacement and datatype. In addition, to allow maximum flexibility, the displacement of blocks within the send and receive buffers is specified in bytes.

MPI_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm)

IN	sendbuf	starting address of send buffer
IN	sendcounts	integer array equal to the group size specifying the number of elements to send to each processor
IN	sdispls	integer array (of length group size). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for process j
IN	sendtypes	array of datatypes (of length group size). Entry j specifies the type of data to send to process j
OUT	recvbuf	address of receive buffer
IN	recvcounts	integer array equal to the group size specifying the number of elements that can be received from each processor
IN	rdispls	integer array (of length group size). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from process i
IN	recvtypes	array of datatypes (of length group size). Entry i specifies the type of data received from process i
IN	comm	communicator (handle)

```
int MPI_Alltoallw(void* sendbuf, int* sendcounts,
             int* sdispls,MPI_Datatype* sendtypes, void* recvbuf,
             int* recvcounts, int* rdispls,MPI_Datatype* recvtypes,
             MPI_Comm comm)
```

```
MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS,
             RDISPLS, RECVTYPES, COMM)
    INTEGER SENDBUF, SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVBUF,
    RECVBUFS(*), RECVCOUNTS(*), RDISPLS(*), RECVTYPES(*), COMM, IERR
```

The `j`-th block sent from process `i` is received by process `j` and is placed in the `i`-th block of recvbuf. These blocks need not all have the same size.

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

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

$$\mathtt{MPI\_Send}(\mathtt{sendbuf} + \mathtt{sdispls}[\mathtt{i}], \mathtt{sendcounts}[\mathtt{i}], \mathtt{sendtypes}[\mathtt{i}], \mathtt{i}, ...),$$

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

$$\mathtt{MPI\_Recv}(\mathtt{recvbuf} + \mathtt{rdispls}[\mathtt{i}], \mathtt{recvcounts}[\mathtt{i}], \mathtt{recvtypes}[\mathtt{i}], \mathtt{i}, ...).$$

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

A clarification is that no in-place support is offered.

6.6 Exclusive Scan

The following call removes the omission from MPI-1, of an exclusive scan, and can evidently be implemented with acceptable efficiency:

MPI_EXSCAN(sendbuf, recvbuf, count, datatype, op, comm)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	count	number of elements in input buffer (integer)
IN	datatype	data type of elements of input buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)

```
int MPI_Exscan(void *sendbuf, void *recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

```
MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, IERR)
    INTEGER SENDBUF, RECVBUF, COUNT, DATATYPE, OP, IERR
```

The value for process with rank zero is undefined.
This operation applies only to intracommunicators.

6.7 Channels

Channels offer early-binding of send/receive pair information, and provide a non-realtime analog of the channels offered in the realtime chapter in the Journal of Development. These channels offer:

- Early binding of all details of a send/receive pair,
- Negotiation of advance reservations on network or PVCs,
- Addressing any special issues of alignment/misalignment/copying,
- Negotiation for who does any needed data conversion
- Change of protocol as function of length of data or endpoints.
- Collective definition of up to $N(N-1)$ channels with one call.
- None of the overhead of singleton channel initialization, with singleton channel capability as a subset.

The following call performs a blocking point-to-point operation, much like a MPI_SENDRECV, with the endpoint specified by the original constructor.

MPI_BIND(request)

INOUT	request	request object, already constructed (persistent request)

```
int MPI_Bind(MPI_Request *request)
```

```
MPI_BIND(REQUEST, IERR)
    INTEGER REQUEST, IERR
```

Requests created by the channel initialization may be individually freed using MPI_REQUEST_FREE when they are no longer needed. Programmatically, a channel must stop being used before either side is torn down, so a programmer will have to employ explicit barriers or other devices to be sure they are quiescent before being deleted. This is consistent with the concept of delivering higher performance at the cost of involving the user more closely in the details of the program's behavior.

```
% Sending Side (rank 0)
    MPI_Init_send(buf, count, type, 777, 1, comm, &request);
    MPI_Bind(request); /* bi-collective operation to make channel */
    ...
    for(i = 0; i < 10; ++i)
    {
       MPI_Start(request); /* talk to rank 1 */
       ...
       MPI_Wait(&request); /* completion */
    }
    MPI_Request_free(&request); /* free persistent request */

% Receiving Side (rank 1)
    MPI_Init_Recv(buf, count, type, 777, 0 comm, &request);
    MPI_Bind(request); /* bi-collective operation to make channel */
    ...
    for(i = 0; i < 10; ++i)
    {
       MPI_Start(request); /* talk to rank 0 */
       ...
       MPI_Wait(&request); /* completion */
    }
    MPI_Request_free(&request); /* free persistent request */
```

Discussion: Variants with specific buffering support could be offered.

Chapter 7

External Interfaces

7.1 Introduction

Current Status: no votes

This chapter, by its very nature, contains a number of subject areas.

It begins with the functionality to create generalized requests. The objective of this MPI-2 addition is to allow users of MPI to be able to create functionality similar to what is present in MPI. Blocking, nonblocking, and persistent operations can be performed inside of the MPI mechanisms. This functionality could be used to layer the new I/O functionality on top of MPI.

The next section deals with setting the information found in status. This is necessary for both generalized requests and the new types of requests found in MPI-2.

The next section allows users to associate names with communicators. This will allow debuggers and profilers to identify communicators with more useful labels. The following section allows users to add error codes and classes to MPI. With users being able to layer functionality on top of MPI it seems desirable for them to use the same error mechanisms found in MPI.

The next section deals with decoding, transporting, and storing datatypes. 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. In addition, it is desirable to be able to transport a datatype between processes. This will be useful, for example, in one-sided operations and I/O. This capability is also supported. Finally, there are times when users wish to store a datatype where its lifetime is longer than a single MPI job. To allow for reusing this datatype at a later time on different architectures or with a different MPI implementation, it is necessary to define a "canonical" representation of a datatype. This is discussed at the end of this chapter.

The chapter concludes with a discussion of how threads are to be handled in MPI-2. Although thread compliance is not required, the standard specifies how threads are to work if they are provided. This section also includes user handlers which are called when an MPI request (such as a send or receive) finishes.

7.2 Generalized Requests

7.2.1 Introduction

Current Status: one vote

The goal of this MPI-2 extension is to allow users to create requests. These requests could be used to perform additional operations not supported by MPI such as intricate collective operations or I/O functions. It is possible to create blocking, nonblocking, and persistent generalized requests.

The functionality of a generalized request is defined by the callback functions that are associated with it. The callback functions get invoked at specified times to perform the appropriate actions. The callbacks that are defined are for starting, progression/completion detection, completion, and cancellation. A generalized request is created with MPI_META_REQUEST_CREATE and ultimately deallocated with MPI_META_REQUEST_FREE. A persistent version of a generalized request is created with MPI_META_REQUEST_INIT and freed with the current MPI_REQUEST_FREE. A generalized request is started with MPI_DO for blocking and MPI_START{ALL} for nonblocking versions. When a generalized request is started, the start_fn callback function is invoked. Once a generalized request is started, MPI will use the push_fn to cause the generalized request to make progress or detect when it is complete. The handlers described in Section 7.9.4 may also be used for this purpose. MPI is informed that a generalized request is complete by having the push_fn or a handler call MPI_REQUEST_MARK_COMPLETE. The main user code can determine when a generalized request is complete by using MPI_{TEST|WAIT}{ANY|SOME|ALL}. When a test or wait is going to return true, the complete_fn callback is invoked. This can do cleanup and also can fill in the status values appropriately. In the blocking version, where no test or wait is performed, this callback is invoked before the call returns.

A typical sequence of calls for a nonblocking generalized request would be:

```
MPI_META_REQUEST_CREATE
MPI_START: this invokes the start_fn callback
   MPI will repeatedly invoke the push_fn callback until it
    does an MPI_REQUEST_MARK_COMPLETE and returns true.
    This goes on in the ``background''.
MPI_WAIT: wait for generalized request to complete.  Once it has the
 complete_fn is invoked.
MPI_META_REQUEST_FREE
```

A typical sequence of calls for a blocking generalized request would be:

```
MPI_META_REQUEST_CREATE
MPI_DO: this invokes the start_fn callback
   MPI will repeatedly invoke the push_fn callback until it
    does an MPI_REQUEST_MARK_COMPLETE and returns true.
    When the generalized request is done, the  complete_fn is invoked
    before this call returns.
MPI_META_REQUEST_FREE
```

A typical sequence of calls for a nonblocking persistent generalized request would be:

```
MPI_META_REQUEST_CREATE
MPI_REQUEST_INIT
MPI_START: this invokes the start_fn callback
   MPI will repeatedly invoke the push_fn callback until it
    does an MPI_REQUEST_MARK_COMPLETE and returns true.
    This goes on in the ``background''.
MPI_WAIT: wait for generalized request to complete.  Once it has the
 complete_fn is invoked.
MPI_REQUEST_FREE
MPI_META_REQUEST_FREE
```

This section begins with an overview of the callback functions which is followed by the specific calls. Next, an example of creating a nonblocking version of MPI_ALLGATHER is given.

7.2.2 Functionality

Current Status: one vote

The behavior of the request is defined by these callback functions:

start_fn This callback function is invoked when the operation associated with the generalized request is started. This function can be used, for example, to start one or more nonblocking communications that initiate the operation.

push_fn This callback function is invoked by MPI to cause the generalized request to progress or detect completion.

complete_fn This callback function is invoked when the request has completed at a wait or test. It can be used to set return values in extra_state and set status values.

cancel_fn This callback function is invoked whenever there is a failure in callback functions (see section 7.2.3 for a discussion of failure) or whenever MPI_CANCEL is called to cancel the request. This function can be used, for example, to deallocate memory, cancel internal messages, reset state, etc.

A new generalized request is created with

MPI_META_REQUEST_CREATE(start_fn, push_fn, complete_fn, cancel_fn, extra_state, meta_req)

IN	start_fn	Request start callback function for meta_req (function)
IN	push_fn	Request progress or completion marking callback function for meta_req (function)
IN	complete_fn	Request complete callback function for meta_req (function)
IN	cancel_fn	Request cancel/failure callback function for meta_req (function)
IN	extra_state	Extra state to pass to the callback functions (choice)
OUT	meta_req	MPI request (handle)

```
int MPI_Meta_request_create(MPI_Start_function *start_fn, MPI_Push_function
              *push_fn, MPI_Complete_function *complete_fn,
              MPI_Cancel_function *cancel_fn, void *extra_state, MPI_Request
              *meta_req)
```

```
MPI_META_REQUEST_CREATE(START_FN, PUSH_FN, COMPLETE_FN, CANCEL_FN,
              EXTRA_STATE, META_REQ, IERROR)
    EXTERNAL START_FN, PUSH_FN, COMPLETE_FN, CANCEL_FN
    INTEGER EXTRA_STATE, META_REQ, IERROR
```

meta_req is the request handle that will be used to start nonpersistent requests, and the input template to be used for creating persistent requests. This call is local and the meta_req is a local handle that cannot be transfered to another process.

The callback functions are described below with the routines that cause them to be called. If an MPI implementation uses threads, the callbacks can be executed on any thread the system desires. In particular, there is no requirement that a callback functions executes on the thread where it was declared.

Advice to users. The extra_state supplied in this call will be passed to the callback routines. One should take care that any information attached with extra_state be in appropriate memory. The local stack associated with an earlier callback may have disappeared and callbacks can occur on different threads. As a result, the memory used in extra_state should be globally available (heap) for the duration of time that any callback can occur. Users should also keep in mind the potential problem with Fortran registers discussed in Section 2.5.2 on Page 10.

A library may create a generalized request that can be started at a later time by a user. For example, one may set up a generalized request to do an I/O write operation. In this case the communicator or file on which the generalized request acts can change from call to call. The way to pass this information to the generalized request is by associating it with extra_state before the generalized request is started. This can be done by the user or with a routine supplied by the library. Since generalized requests can be nonblocking, care should be taken to make sure that concurrently running versions of the same generalized request keep the extra_state information straight for each instantiation of the generalized request. (*End of advice to users.*)

Rationale. An alternative design is to pass the extra_state information when the generalized request is started. One drawback to this design is that there may be information which is static and required by all instantiations of the generalized request. A second drawback is that the current mechanisms for starting a request do not have such an argument. Doing this would require a new set of functions for starting a request. Overall, the current design was felt to be superior. (*End of rationale.*)

Advice to users. A generalized request may have new information that is needed each time it is started. For example, the amount of data moved for this particular instantiation of the generalized request. When this is the case, the user should add this information to the extra_state "structure" given when the generalized reqeust was created. Users should use extra care if multiple instantiations of a generalized request can be active simultaneously. In this case, the information in extra_state needs to be unique for each active request. (*End of advice to users.*)

A generalized request may be started in one of two ways to get a blocking or nonblocking version. The blocking version is started with the MPI_DO (See Section 7.3) and a nonblocking version is started with MPI_START{ALL}.

Advice to implementors. The functions MPI_START{ALL} are generalized in MPI-2 to accept persistent requests and generalized requests. (*End of advice to implementors.*)

A persistent version of a generalized request is created with a call to MPI_META_REQUEST_INIT:

MPI_META_REQUEST_INIT(meta_req, request)

IN	meta_req	Type of request desired (from MPI_META_REQUEST_CREATE) (handle)
OUT	request	Handle to persistent version of generalized request (handle)

```
int MPI_Meta_request_init(MPI_Request meta_req, MPI_Request *request)
```

```
MPI_META_REQUEST_INIT(META_REQ, REQUEST, IERROR)
    INTEGER META_REQ, REQUEST, IERROR
```

Once a persistent version of a generalized request is created, it can be started as with nonpersistent generalized requests via a call to MPI_START for a nonblocking version and with a call to MPI_DO for a blocking version.

When a generalized request is started by either of the methods above, the start_fn supplied in the MPI_META_REQUEST_CREATE call is invoked. It is defined in C by:

```
typedef int MPI_Start_fn(void *extra_state, MPI_Request request);
```

A Fortran declaration for such a function is as follows:

```
SUBROUTINE START_FN(EXTRA_STATE, REQUEST, IERR)
    INTEGER EXTRA_STATE, REQUEST, IERR
```

The input extra_state is the internal state of the request that was given by the original call to MPI_META_REQUEST_CREATE. start_fn may initiate one or more operations as part of starting the generalized request.

start_fn returns MPI_SUCCESS if it succeeds in C or C++ and the same in IERR in Fortran. The consequence of returning a failure code is discussed later.

The push_fn is called by MPI when it wants to cause the generalized request to make progress. It is defined in C by:

```
typedef int MPI_Push_fn(void *extra_state, MPI_Request request, int *done);
```

A Fortran declaration for such a function is as follows:

```
SUBROUTINE PUSH_FN(EXTRA_STATE, REQUEST, DONE, IERR)
    INTEGER EXTRA_STATE, REQUEST, IERR//LOGICAL DONE
```

The input extra_state is a pointer to the internal state of the request that was given by the original call to MPI_META_REQUEST_CREATE. MPI will continue to recall the push_fn until it returns true in done. push_fn should return false in done if it is not complete. push_fn returns MPI_SUCCESS if it succeeds in C and C++ and the same in IERR in Fortran. The consequence of returning a failure code is discussed later.

Advice to users. Returning true in done tells MPI not to call the push_fn any more. If the push_fn is the sole mechanism for marking a generalized request as complete, then MPI_MARK_REQUEST_COMPLETE must be called in this last invocation of push_fn. Otherwise, MPI will never know the generalized request is complete. (*End of advice to users.*)

A generalized request must make progress in the same sense that an enabled point-to-point request will finish (See Section 3.5 of MPI-1.1 document). Thus, an MPI implementation must call the push_fn inside its progression engine or through a similar mechanism. It is not required that a generalized request make progress at the same "rate" as other MPI requests. An MPI implementation may choose to have generalized requests have lower or higher priority in making progress. However, as stated above, a generalized request must make some progress if it is enabled.

Discussion:
We have to be careful as to exactly how we define progress. In a threaded environment we probably want the same type of definition as the handlers currently have. This seems to be what everyone expects (progress, etc). When there are no threads, then it gets tougher. We may say that no progress will occur until the wait/test is called for a nonblocking request. If the request is freed then this can be a problem. The same type of situation can happen with point-to-point requests so maybe the same progress rule can apply. Presumably with a blocking generalized request the system will make progress as with a blocking point-to-point operation. We need to finalize our thinking here.

A related issue to progress is where the callbacks can be run. We do not have locks in MPI so protecting critical sections becomes a problem. We can say that we do not have portable locks and thus restrict the callbacks that want to be portable. This could restrict someone's ability to create simultaneous nonblocking operations without extra work.

The push_fn is provided because generalized requests may be used to cause non-MPI events to occur. For example, one may wish to cause an I/O write event to occur. In this case, MPI will be unaware of how this event is progressing. This is why the push_fn is necessary. In some systems the I/O write can progress without any outside intervention. For example, the start_fn may turn the request over to an intelligent disk controller. In this case the push_fn callback function might only check with the disk controller to see if the write is complete. In other systems, the I/O write may not finish unless the initiating process takes some active steps during the I/O write. In this case the push_fn can initiate these steps when it is called. Finally, it may not be necessary for a generalized request to have a push_fn. For example, one may be emulating an MPI call and using the handler described in Section 7.9.4 to cause progress and signal completion. For this case, the predefined push_fn called MPI_NULL_PUSH_FN may be used. MPI_NULL_PUSH_FN tells MPI that it does not need to call any push_fn.

Advice to users. Using MPI_NULL_PUSH_FN lets MPI implementations know that it is unnecessary to call any progression callback. This is likely to more efficient than providing an "empty" callback function. (*End of advice to users.*)

Advice to implementors. If MPI_NULL_PUSH_FN is provided, it may be much more efficient to avoid putting the generalized request on a list for progress to occur than to use an "empty" callback for push_fn. (*End of advice to implementors.*)

For nonblocking requests, the push_fn will be called in MPI_{TEST|WAIT}{ANY|SOME|ALL} as appropriate to check for completion of the generalized request. However, MPI makes no more guarantees about fairness in a call to MPI_{TEST|WAIT}{ANY|SOME|ALL} for a generalized request than it did for other requests. Thus, repeatedly calling MPI_WAITANY with a list of requests where another request can be returned complete can lead to a situation where the generalized request is never returned as complete. An MPI implementation, however, is not permitted to pass over a generalized request if it should be checked by the push_fn. For example, an MPI implementation receiving a call to MPI_TEST with a generalized request must call the push_fn. It may not simply return false for completion because the generalized request has not yet been marked complete. As with other cases of progression for generalized requests, an MPI implementation is free to choose the frequency of recalling the push_fn in MPI_WAIT as long as progress ultimately occurs.

An MPI_{TEST|WAIT} will return that a generalized request is complete once it has been marked complete by the user and the complete_fn has completed. The user does this by calling the function:

MPI_REQUEST_MARK_COMPLETE(request)

INOUT	request	Generalized request to be marked as complete (handle)

```
int MPI_Request_mark_complete(MPI_Request *request)
```

```
MPI_REQUEST_MARK_COMPLETE(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
```

This function enables the callback function complete_fn defined as:

```
typedef int MPI_Request_complete_fn(void *extra_state, MPI_Request request,
             MPI_Status *status);
```

A Fortran declaration for such a function is as follows:

```
SUBROUTINE REQUEST_COMPLETE_FN(EXTRA_STATE, REQUEST, STATUS, IERR)
    INTEGER EXTRA_STATE, META_REQ, STATUS(MPI_STATUS_SIZE
```

The status supplied to complete_fn is the status supplied to the MPI_{TEST|WAIT} or MPI_DO call. The complete_fn can fill in any necessary information in the status with the methods described in section 7.4. For nonblocking requests, since the complete_fn is supplied the status from the MPI_{TEST|WAIT} call, it cannot be called until a call to MPI_{TEST|WAIT} is going to return true for completion of this generalized request. The MPI_{TEST|WAIT} will not return true for the generalized request until the complete_fn has completed. For blocking requests, the complete_fn is called right before the call to MPI_DO is about to return. MPI_DO will not return until the complete_fn has completed. This is necessary to allow the callback to fill in the status.

Rationale. Postponing the complete_fn until after the MPI_{TEST|WAIT} simplifies filling in the status. A consequence of this is that an MPI_{TEST|WAIT} involving a generalized request can take longer to complete. This was felt to be a reasonable compromise. (*End of rationale.*)

Advice to users. Users who wish to have actions happen sooner upon completion of a generalized request should execute these operations in the final execution (when MPI_REQUEST_MARK_COMPLETE is called) of the callback push_fn (or its equivalent handler) is called.

Users should take care if MPI_REQUEST_FREE is called on the request. The above semantics mean that the complete_fn will never execute because a wait or test will never occur. (*End of advice to users.*)

In addition to filling in status, the complete_fn can clean up after the generalized request and may put information in extra_state which is of interest to the calling program. complete_fn returns MPI_SUCCESS if it succeeds in C or C++ and the same in IERR in Fortran. The consequence of returning a failure code is discussed later.

Advice to users. If the MPI defined push_fn MPI_NULL_PUSH_FN is used, the user must make sure that the MPI_REQUEST_MARK_COMPLETE is called somewhere. Otherwise, the generalized request will never finish. (*End of advice to users.*)

A persistent version of a generalized request can be restarted with another call to MPI_START{ALL} or MPI_DO once it has completed. Once the user is done with a persistent generalized request, it is freed using the current MPI_REQUEST_FREE.

The cancel_fn is defined in C by:

```
typedef int MPI_Request_cancel_fn(void *extra_state, MPI_Request request);
```

A Fortran declaration for such a function is as follows:

```
SUBROUTINE REQUEST_CANCEL_FN(EXTRA_STATE, REQUEST, IERR)
    INTEGER EXTRA_STATE, REQUEST, IERR
```

This callback function is invoked by MPI whenever the request is to be cancelled, either via a call to MPI_CANCEL or because of a failure in one of the other callback functions (failure is described below in section 7.2.3). This callback function can deallocate the internal data structures and cancel the internal messages of the request (these can be accessible via extra_state). See section 7.2.3 for details. If it was succesful in cancelling the request, cancel_fn returns MPI_SUCCESS in C or C++ and the same in IERR in Fortran. Otherwise, a failure code is returned as described in section 7.2.3.

Finally, since allocation of the meta_req may use some system resources, it can be returned to the system with:

MPI_META_REQUEST_FREE(meta_req)

INOUT	meta_req	Type of request to be freed (handle)

```
int MPI_Meta_request_free(MPI_Request *meta_req)
```

```
MPI_META_REQUEST_FREE(META_REQ, IERROR)
    INTEGER META_REQ, IERROR
```

It is legal to free a meta_req for which an active request still exists. MPI will deallocate the meta_req when it is safe to do so.

7.2.3 Failed User Functions and Cancelling Requests

Current Status: no vote

When any of the start_fn, push_fn, or complete_fn callbacks returns an error code, MPI invokes the cancel_fn callback to cancel the request. MPI also invokes the cancel_fn when the user calls MPI_CANCEL. When cancel_fn returns, MPI marks the request as complete, regardless of whether the request was cancelled or completed normally. If the cancel_fn returns MPI_SUCCESS, it was successful in cancelling the request (i.e., the request had not yet completed), and MPI marks the request's status such that a call to MPI_TEST_CANCELLED for the status will return flag=true. Otherwise, MPI marks the request's status such that MPI_TEST_CANCELLED for the status will return flag=false. Unlike cancelling a point-to-point request, a successful cancel does not imply that all buffers and resources are unchanged. Cleanup is expected of the cancel_fn but not required by MPI. If cancel_fn cannot clean up resources then it can choose to raise an error.

Advice to users.

The extra_state argument can be used to give further information about a request when the cancel_fn does not return MPI_SUCCESS.

(*End of advice to users.*)

MPI provides a default cancel routine called MPI_CANCEL_ERROR. That routine simply raises the MPI error MPI_ERR_GR. Since there is no comm associated with a generalized request, the error raised is on MPI_COMM_WORLD. MPI_CANCEL_ERROR acts as if it returned MPI_SUCCESS. This will mark the generalized request as cancelled.

Rationale. MPI_CANCEL_ERROR is only expected to be used when no error recovery is used in a generalized request. If error recovery is desired then a cancel_fn should be supplied. Thus, MPI_CANCEL_ERROR is likely to be used when cancelling is not supported or the callbacks never fail. In this case, it does not matter what cancel_fn is supplied. The other likely use is when the user is not recovering from an error. By default, errors on MPI_COMM_WORLD are fatal so this error will cause the program to abort and the value returned by MPI_CANCEL_ERROR is not important. If another error handler is being used on MPI_CANCEL_ERROR and it permits the program to continue, it will appear to the user that the generalized request failed. Since error recovery is not the goal of this default error handler, this seemed the logical default. (*End of rationale.*)

7.2.4 Example of MPI_ALLGATHER

Current Status: no vote

In this example the user is trying to perform a nonblocking MPI_ALLGATHER. It will be done with a generalized request by performing $p-1$ wrapped shifts of data ($p = \#$ processes in the communicator).

More specifically, assume each process i has an array of size $a[p]$ and its value is in $a[i]$. The algorithm works by accepting data from process $(i+p-1)\%p$ and sending data to $(i+1)\%p$. At each step, process i forwards the data it received in the last step. Once you shift $p-1$ times, each process has a copy of what was stored on all other processes. Use persistent communication requests to send and receive the data. To accomplish this, have a separate send and recv buffer. Initialize the send buffer on process i to be $a[i]$. After the send and recv have occurred, copy the recv buffer into array a and also into the send buffer and repeat the process until done. (This may not be the normal way to do this but it is done to utilize different features for the sake of discussion. Most people would probably do a sendrecv directly into the correct locations in array a.)

The blocking C code is below. This is to show how it would be done now in MPI without a nonblocking request.

```
MPI_Comm_size(comm, &p);
MPI_Comm_rank(comm, &i);
/* init your location in array */
a[i] = i;
/* set initial send value */
as = a[i]
/* process to recv from and persistent request */
from = (i + p - 1) % p;
MPI_Recv_init(&ar, 1, MPI_DOUBLE, from, 13, comm, &req_recv);
/* process to send to and persistent request */
to = (i + 1) % p;
MPI_Send_init(&as, 1, MPI_DOUBLE, to, 13, comm, &req_send);
/* do send and recv p - 1 times */
for (c1 = 0; c1 < p - 1; c1++) {
   /* begin persistent send and recv */
   MPI_Start(&req_recv);
   MPI_Start(&req_send);
   /* when recv complete copy result in final array a */
   MPI_Wait(&req_recv, &status_recv);
   a[(i - 1 - c1 + p) % p] = ar;  /* add p to make sure mod is positive */
   /* when send completes, copy recv buf into send buf for next round */
   MPI_Wait(&req_send, &status_send);
```

```
    as = ar;
}
/* free up persistent requests */
MPI_Request_free(&req_recv);
MPI_Request_free(&req_send);
```

At the start, $a[i] = i$ and $a[j] = 0$ ($j \neq i$) on process i. At the end, $a[j] = j$ on all processes.

As an example, with $p = 4$, process 1 does:

c1	send	recv
0	$a[1]$	$a[0]$
1	$a[0]$	$a[3]$
2	$a[3]$	$a[2]$

Now to do this with a nonblocking generalized request, create persistent communication requests to send to $(i+1)\%p$ and to request to recv from $(i+p-1)\%p$. The algorithm is demand-driven. Begin by starting the persistent recv and send. Each time one of these finishes, you want MPI to call a handler_fn. When the recv finishes you can copy the data into the correct location in the array a. If the send is done, then you can copy the recv data into the send buffer. Once both are done you can start a new send and recv if you have not finished.

Below is C code.

The main routine which calls the nonblocking operation would do:

```
struct info {
  MPI_Comm *comm;
  int p, i, from;
  MPI_Mutex mutex;
  MPI_Request req_recv, req_send;
  double ar, as, *a;
  int num_completed, num_done;
  MPI_Info handler_attribute;
} extra;

MPI_Request request, meta_req;

/* this supplies the start, push, complete, and cancel functions
   for the request.  Since this uses handlers, there is no push_fn.
   MPI returns in meta_req the new meta request */
MPI_Meta_request_create(start_fn, MPI_NULL_PUSH_FN, complete_fn,
  MPI_CANCEL_ERROR, &extra_state, &meta_req);

/* set up the information needed by the calls.  The communicator to
use in the calls is added to the structure info */
extra.comm = comm;
/* initialize counters and info */
MPI_Comm_size(extra.comm, &extra.p);
```

```
MPI_Comm_rank(extra.comm, &extra.i);
extra.a = (double *)malloc((size_t)sizeof(double) * extra.p)
if (extra.a == NULL) {
  /* error recovery or abort code goes here */
}
/* create a mutex.  This is in the JOD and is NOT part of the official
   MPI standard.  This example needs a lock so either this one or
   another type of lock is needed. */
/* at the current time the mutex attributes have not been specified */
MPI_Mutex_create(MPI_COMM_SELF, MPI_MUTEX_ATTRIBUTE??, &extra.mutex);
/* process to recv from and persistent request */
extra.from = (extra.i + extra.p - 1) % extra.p;
MPI_Recv_init(&extra.ar, 1, MPI_DOUBLE, extra.from, 13, extra.comm,
  &extra.req_recv);
/* process to send to and persistent request */
to = (extra.i + 1) % extra.p;
MPI_Send_init(&extra.as, 1, MPI_DOUBLE, to, 13, extra.comm,
  &extra.req_send);
/* info for handler.  Only need a empty one. */
MPI_Info_create(&extra.handler_attribute);

/* Initialize the request. */
MPI_Meta_request_init(meta_req, &request);

/* start the nonblocking operation.  This calls start_fn. */
MPI_Start(&request);

/* do other stuff that can go on while nonblocking operation is
   going.  This is not shown. */

/* now wait until nonblocking operation is done */
MPI_Wait(&request, &status);

/* you could restart the persistent generalized request here */

/* free up the persistent communications requests since about to free
   this persistent generalized request.  Also done with lock */
MPI_Request_free(&extra.req_recv);
MPI_Request_free(&extra.req_send);
/* This is a JOD function and is NOT standardized */
MPI_Mutex_free(&extra.mutex);

/* done with request, free up. */
MPI_Request_free(&request);

/* done with meta_req, free up */
MPI_Meta_request_free(&meta_req);
```

The functions for the generalized request are:

```
int start_fn(struct info *extra, MPI_Request request)
{
/* init your location in array */
extra->a[extra->i] = extra->i;
/* set initial send value */
extra->as = extra->a[extra->i]
/* haven't done any yet */
extra->num_completed = 0;
extra->num_done = 0;

/* begin persistent recv and send */
MPI_Start(&extra->req_recv);
MPI_Post_Handler(extra->req_recv, extra->handler_attribute, handler_fn,
  extra, MPI_HANDLER_DEFAULT);
MPI_Start(&extra->req_send);
MPI_Post_Handler(extra->req_send, extra->handler_attribute, handler_fn,
  &extra, MPI_HANDLER_DEFAULT);

return MPI_SUCCESS;
}

int complete_fn(struct info *extra, MPI_Request request, MPI_Status *status)
{
/* fill in status */
MPI_Set_elements_status(1, MPI_DOUBLE, status);
status.MPI_SOURCE = extra->from;
status.MPI_TAG = 13;
status.MPI_ERROR = 0;

return MPI_SUCCESS;
}
```

Each time a request (generated inside the generalized request) finishes, the handler_fn gets called:

```
int handler_fn(MPI_Request *request, struct info *extra)
{
/* this function is in JOD and NOT standardized */
MPI_Mutex_lock(extra->mutex);
/* note a send or recv finished */
extra->num_completed++;
/* if num_completed != 2 then both send and recv not done */
if (extra->num_completed != 2) {
    /* this function is in JOD and NOT standardized */
    MPI_Mutex_unlock(extra->mutex);
    return (MPI_SUCCESS);
```

```
}

extra->num_completed = 0; /* reset */
/* this function is in JOD and NOT standardized */
MPI_Mutex_unlock(extra->mutex);

/* We don't get here unless both the send & recv completed */
/* note another round completed */
extra->num_done++;

/* copy received data */
extra->a[(extra->i - 1 - extra->num_done + extra->p) % extra->p]
      = extra->ar;
/* see if completely done */
if (extra->num_done == extra->p - 1) {
    /* mark request done */
    MPI_Request_mark_completed(request);
}
else {
    /* set data to be sent */
    extra->as = extra->ar;

    /* begin new round */
    MPI_Start(&extra->req_recv);
    MPI_Start(&extra->req_send);
    MPI_Post_Handler(&extra->req_recv, MPI_HANDLER_ATTRIBUTE, handler_fn,
      &extra->status_recv, MPI_HANDLER_DEFAULT);
    MPI_Post_Handler(&extra->req_send, MPI_HANDLER_ATTRIBUTE, handler_fn,
      &extra->status_send, MPI_HANDLER_DEFAULT);
}

return (MPI_SUCCESS);
}
```

7.3 Blocking Instantiation of a Request

Current Status: no vote

A persistent collective or generalized request can be started in a blocking fashion with

MPI_DO(request, status)

INOUT	request	request to be started (handle)
OUT	status	status returned for request (handle)

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

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

MPI_DO starts request. When it returns, request has completed and status has the appropriate values for that request. If the request was nonpersistent then it is freed before completion of the function. If the request was persistent then it is inactive when the function returns.

Discussion:
We could also allow one to use this for other requests. For example, starting blocking versions of all the persistent requests.

We could also have MPI_IDO to do the nonblocking equivalent. Right now we use MPI_START for this.

7.4 Associating Information with Status

Current Status: no vote

In MPI-1 requests were associated with point-to-point operations. As a result, the status object defined had fields to return information about point-to-point communications. In MPI-2 there are many different types of requests. These range from new MPI calls for dynamic processes to generalized requests. It is desirable to make all these calls use the same request mechanism. This allows one to wait or test on different types of requests. However, MPI_{TEST|WAIT}{ANY|SOME|ALL} returns a status with information about the request. With the generalization of requests, one needs to define what information will be returned in the status object.

Most of the new calls in MPI-2 only return an error code. Some of the calls in the I/O chapter also return a count in analogy to point-to-point calls in MPI-1. In MPI-2 these calls will fill in the appropriate fields in the status object. Any unused fields will have undefined values. A call to MPI_{TEST|WAIT}{ANY|SOME|ALL} can modify any of the fields in the status object. Specifically, it can modify fields that are undefined. The fields with meaningful value for a given request are defined in the sections with the new request.

Generalized requests raise additional considerations. Here the user provides the functions to deal with the request. Unlike other MPI calls, the user needs to provide the information to be returned in status. The status argument is provided directly to the callback function where the status needs to be set. Users can directly set the values in 3 of the 5 status fields. The count field is opaque. To overcome this, a new call is provided:

MPI_SET_ELEMENTS_STATUS(count, datatype, status)

IN	count	Count to associate with status (integer).
IN	datatype	datatype associated with count (handle).
OUT	status	Status to associate count with (handle)

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

```
MPI_SET_ELEMENTS_STATUS(COUNT, DATATYPE, STATUS, IERROR)
    int COUNT, DATATYPE, STATUS, IERROR
```

This call modifies the opaque part of status so that a call to MPI_GET_ELEMENTS will return count. It is erroneous to use this function on any status returned by a call to an MPI function.

Rationale. MPI_GET_ELEMENTS is used instead of MPI_GET_COUNT because the former can deal with nonintegral numbers of datatypes. (*End of rationale.*)

Advice to users. The status argument is last in this call since it is an OUT parameter. This is in contrast to MPI_GET_ELEMENTS where it is first. (*End of advice to users.*)

The other opaque field in status is the field to indicate if the request had been cancelled. Since generalized requests support cancelling, this will be set automatically by the MPI implementation.

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. (*End of advice to users.*)

Discussion:
I have removed the ability to use set status for functions other than generalized requests. To do this requires an init routine to make sure an other opaque parts of status are set which would have happened at the wait/test.

7.5 Naming Objects

Current Status: one vote

There are many occasions on which it would be useful to allow a user to associate a print name with an MPI communicator, for instance error reporting, debugging, profiling, and so on. This objective can be achieved using the following two functions:

MPI_NAME_PUT (comm, name)

INOUT	comm	Communicator whose name is to be set
IN	name	The character string which is remembered as the name

```
int MPI_Name_put(MPI_Comm comm, char *name)
```

```
MPI_NAME_PUT(COMM, NAME, IERROR)
    INTEGER COMM
    CHARACTER*(*) NAME
    INTEGER IERROR
```

MPI_NAME_GET (comm, name)

IN	comm	Communicator whose name is to be returned
OUT	name	The name previously stored on the communicator, or a null string if no such name exists

```
int MPI_Name_get(MPI_Comm comm, char **name)
```

```
MPI_NAME_GET(COMM, NAME, IERROR)
    INTEGER COMM
    CHARACTER*(*) NAME
    INTEGER IERROR
```

MPI_NAME_PUT allows a user to associate a name string with a communicator. The character string which is passed to MPI_NAME_PUT will be saved inside the MPI library (so can be freed by the caller immediately after the call, or allocated on the stack).

The length of a name which can be stored is limited to the value of MPI_MAX_NAME_STRING. Attempts to put names longer than this will fail, or result in truncation of the name. MPI_MAX_NAME_STRING must have a value of at least 63.

> *Advice to users.* Under circumstances of store exhaustion an attempt to put a name of any length could fail, therefore the value of MPI_MAX_NAME_STRING should be viewed only as a strict upper bound on the name length, not a guarantee that putting names of less than this length will always succeed. (*End of advice to users.*)

> *Advice to implementors.* Implementations which pre-allocate a fixed size space for a name should use the length of that allocation as the value of MPI_MAX_NAME_STRING. Implementations which allocate space for the name from the heap can set MPI_MAX_NAME_STRING to any arbitrarily large value, since MPI_NAME_PUT is always allowed to fail. (*End of advice to implementors.*)

MPI_NAME_GET returns any name which had previously been associated with the given communicator. In C and C++ it returns a pointer to the internal copy of the name, which should not be modified by the user, and which will become invalid should the name be altered (e.g. by a subsequent call to MPI_NAME_PUT, or by the communicator being deleted).

If the user has not associated a name with a communicator, or an error occurs, MPI_NAME_GET will return a null string (all spaces in Fortran, "" in C and C++). The three predefined communicators will have predefined names associated with them. Thus, MPI_COMM_WORLD, MPI_COMM_SELF, and MPI_COMM_PARENT will have the default names of `MPI_COMM_WORLD`, `MPI_COMM_SELF`, and `MPI_COMM_PARENT`. The fact that the system may have chosen to give a default name to a communicator does not prevent the user from setting a name on the same communicator; doing this removes the old name and assigns the new one.

Rationale. We provide separate functions for setting and getting the name of a communicator, rather than simply providing a predefined attribute key for the following reasons:

- It is not, in general, possible to store a string as an attribute from Fortran.
- It is not easy to set up the delete function for a string attribute unless it is known to have been allocated in the heap.
- To make the attribute key useful additional code to call `strdup` is necessary. If this is not standardized then users have to write it. This is extra unneeded work which we can easily eliminate.
- The Fortran binding is not trivial to write (it will depend on details of the Fortran compilation system), and will not be portable. Therefore it should be in the library rather than in user code.

(*End of rationale.*)

Advice to users. The above definition means that it is safe simply to print the string returned by MPI_NAME_GET, 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) marginally increase the store requirement of the program, since the names must be saved. Therefore there is no requirement that names be associated 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.*)

7.6 Adding New Error Classes and Error Codes

Current Status: no votes

Users may want to write a layered library on top of an existing MPI implementation, and this library may have its own set of error codes and classes. An example of such a library is an I/O library based on the I/O chapter in MPI-2. For this purpose, functions are needed to:

1. add a new error class to the ones an MPI implementation already knows.

2. associate error codes with this error class, so that MPI_Error_class works.

3. associate strings with these error codes, so that MPI_Error_string works.

Three new functions are proposed below:

MPI_ADD_ERROR_CLASS(errorclass)

OUT	errorclass	Value for the new error class

```
int MPI_Add_error_class(int *errorclass)
```

```
MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR)
    INTEGER ERRORCLASS, IERROR
```

Creates a new error class and returns the value for it.

Rationale. To avoid conflicts with existing error codes and classes, the value is set by the implementation and not by the user. (*End of rationale.*)

The value of MPI_ERR_LASTCODE is not affected by new user-defined error codes and classes. As in MPI-1, it is a constant value. Instead, a predefined attribute key MPI_LASTUSERCODE is associated with MPI_COMM_WORLD. The attribute value returned by this key is the current maximum error code including the user defined ones. This is a local value and may be different on different processes. It is always the case that MPI_ERR_LASTCODE is less than or equal to the value returned by this key.

Discussion:
Historically we have cached values on MPI_COMM_WORLD. However, it may make more sense to do it on MPI_COMM_SELF since it is local here.

Advice to users. The value returned by the key MPI_LASTUSERCODE will not change unless the user calls a function to explicitly add an error class/code. In a multi-threaded environment, the user must take extra care in assuming this value has not changed. (*End of advice to users.*)

MPI_ADD_ERROR_CODES(errorclass, array_errorcodes count)

IN	errorclass	Error class
OUT	array_errorcodes	Array of new error codes
IN	count	Number of error codes needed

```
int MPI_Add_error_codes(int errorclass, int *array_errorcodes, int count)
```

```
MPI_ADD_ERROR_CODES(ERRORCLASS, ARRAY_ERRORCODES, COUNT, IERROR)
    INTEGER ERRORCLASS, ARRAY_ERRORCODES(*), COUNT, IERROR
```

Creates count number of new error codes, associates them with errorclass, and returns their values in array_errorcodes.

Rationale. To avoid conflicts with existing error codes and classes, the values of the new error codes are set by the implementation and not by the user. (*End of rationale.*)

MPI_ADD_ERROR_STRING(errorcode, string, len)

IN	errorcode	Error code or class
IN	string	Text corresponding to errorcode
IN	len	Length (in printable characters) of the text in string

```
int MPI_Add_error_string(int errorcode, char *string, int len)
```

```
MPI_ADD_ERROR_STRING(ERRORCODE, STRING, LEN, IERROR)
    INTEGER ERRORCODE, LEN, 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.

If MPI_ERROR_STRING is called when no string has been set then MPI_ERR_NOSTRING is generated.

7.7 Accessing MPI Datatypes

Current Status: no vote

MPI-1 provides datatype objects, which allows users to specify an arbitrary layout of data in memory. The layout information, once put in a datatype, could not be decoded from the datatype. There are several cases, however, where accessing the layout information in opaque datatype objects would be useful. This section describes functions for achieving this.

One of the cases where access to internal information of a datatype is useful is when a datatype needs to be passed between processes. MPI_PUT and MPI_GET discussed in Chapter 5 require the origin process to specify a datatype to be used in the target process. A natural way to provide the datatype is to define it in the target and transfer it to the origin before the one-sided call. Datatype passing may also be necessary in client-server type programs. A server may need to access a local buffer in the way only clients can specify.

Another case is when they are used in implementing a portable MPI profiler, tracer, or debugger. Users of such tools may want to identify locations in message buffers actually accessed by MPI. Use of MPI_PACK may give a means for identifying *which* data in a buffer MPI accesses, but not *where*.

A third case is when datatypes are used to represent layouts of data, especially in somewhere other than memory. For example, when performing file I/O using MPI datatypes. Files in a database system or files storing out-of-core data for large-scale scientific computation may want to maintain the same layout of data as used in memory. As an MPI datatype is used to express an arbitrary layout of data in memory, also using it to express a layout of data in a file is natural.

The next section describes a set of datatype accessing functions which constitute a lower-level interface. It could serve the needs of profilers and debuggers as well as yielding a general data layout representation.

The second section describes a way to encode and decode a datatype. The last section describes a canonical representation for datatypes.

7.7.1 Low Level Datatype Accessing

Current Status: two votes

The two functions in this section are used together to decode datatypes into forms which can immediately be fed to datatype constructor functions to define a datatype "equivalent" to the original in terms of their typemaps. Different classes of datatypes are defined as:

fundamental. A fundamental datatype is one of those listed in Section 3.2.2 of MPI-1.1 along with MPI_LB and MPI_UB. The ones listed in Section 3.2.2 of MPI-1.1 are known as basic datatypes. All the fundamental datatypes are predefined in MPI.

intrinsic. An intrinsic datatype is one constructed with a single call to one of the constructor functions in Section 3.12.1 of MPI-1.1.

defined. All other datatypes in MPI are defined datatypes. These include ones that are constructed by other constructor functions (such as in the I/O chapter). These datatypes, by the way MPI constructs datatypes, must be composed of fundamental and intrinsic datatypes.

In MPI, all three of these datatype catagories are known as general datatypes. The results of the functions presented in this section are "equivalent" to the original datatype for fundamental and intrinsic datatypes. For a defined datatype, the result is a definition in terms of the intrinsic and fundamental datatypes that underlie its definition.

Missing:
If new intrinsic datatype constructors are added in MPI-2 then they need to be added to the definition above. Also, if we make the I/O ones intrinsic then that needs to be changed too.

MPI_GET_TYPE_ENVELOPE(datatype, count, combiner)

IN	datatype	datatype to access (handle)
OUT	count	number of blocks (nonnegative integer)
OUT	combiner	combiner (integer)

```
int MPI_Get_type_envelope(MPI_Datatype datatype, int* count, int* combiner)
```

```
MPI_GET_TYPE_ENVELOPE(DATATYPE, COUNT, COMBINER, IERROR)
    INTEGER DATATYPE, COUNT, COMBINER, IERROR
```

Given an MPI datatype datatype, returns information on the format in which MPI_GET_TYPE_CONTENTS described below returns internal information of datatype. The type of the format is returned in combiner as one of the following defined constants:

MPI_FUNDAMENTAL	a fundamental datatype
MPI_CONTIG	MPI_TYPE_CONTIGUOUS
MPI_VECTOR	MPI_TYPE_VECTOR
MPI_HVECTOR	MPI_TYPE_HVECTOR
MPI_INDEXED	MPI_TYPE_INDEXED
MPI_HINDEXED	MPI_TYPE_HINDEXED
MPI_STRUCT	MPI_TYPE_STRUCT

These constants are integers.

If combiner is MPI_FUNDAMENTAL, then the returned count is 1; if it is one of the other constants, then the returned count is the number of blocks which was given, as the count argument, to the datatype constructor function indicated by combiner when datatype was constructed.

The combiner must be the same one used in creating the datatype unless it is a defined datatype. Thus, the information returned by these functions can be used to "recreate" the calling sequence used to create the datatype.

MPI_GET_TYPE_CONTENTS(datatype, start, count, array_blocklens, array_disps, array_hdisps, array_types)

IN	datatype	datatype to access (handle)
IN	start	index of first elements to access (non-negative integer)
IN	count	number of elements to access (non-negative integer)
OUT	array_blocklens	array of blocklengths (array of non-negative integers
OUT	array_disps	array of displacements or stride, expressed in elements (array of non-negative integer)
OUT	array_hdisps	array of displacements or stride, expressed in bytes (array of integer)
OUT	array_types	array of element types (array of handles to datatype objects)

```
int MPI_Get_type_contents(MPI_Datatype datatype, int start, int count,
            int *array_blocklens, int *array_disps, MPI_Aint *array_hdisps,
            MPI_Datatype *array_types)
```

```
MPI_GET_TYPE_CONTENTS(DATATYPE, START, COUNT, ARRAY_BLOCKLENS, ARRAY_DISPS,
            ARRAY_HDISPS, ARRAY_TYPES, IERROR)
    INTEGER DATATYPE, START, COUNT, ARRAY_BLOCKLENS(*), ARRAY_DISPS(*),
    ARRAY_HDISPS(*), ARRAY_TYPES(*), IERROR
```

Returns information about the top-level datatypes that underlie datatype except it skips the first start underlying datatypes and returns information on the next count datatypes.

Advice to users. The datatypes returned in array_types are new handles. The user is responsible for freeing these datatypes with MPI_TYPE_FREE. (*End of advice to users.*)

> *Advice to implementors.* The datatypes returned in array_types must appear to the user as if it is a new copy of the datatype. 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.*)

More specifically, if the combiner returned by MPI_GET_TYPE_ENVELOPE is MPI_FUNDAMENTAL, datatype itself is returned in array_types[0]. start, count, array_blocklens, array_disps, and array_hdisps are not touched.

If the combiner returned by MPI_GET_TYPE_ENVELOPE is MPI_CONTIG, then oldtype that can be passed to MPI_TYPE_CONTIGUOUS to reconstruct an equivalent of datatype is returned in array_types[0]. Array_blocklens, array_disps, and array_hdisps are not touched.

If the combiner returned by MPI_GET_TYPE_ENVELOPE is MPI_VECTOR, then blocklen, stride and oldtype that can be passed to MPI_TYPE_VECTOR to reconstruct an equivalent of datatype are returned in array_blocklens[0], array_disps[0] and array_types[0], respectively. array_hdisps is not touched.

If the combiner returned by MPI_GET_TYPE_ENVELOPE is MPI_HVECTOR, then blocklen, stride and oldtype that can be passed to MPI_TYPE_HVECTOR to construct an equivalent of datatype are returned in array_blocklens[0], array_hdisps[0] and array_types[0], respectively. array_disps is not touched.

If the combiner is MPI_INDEXED, MPI_HINDEXED, or MPI_STRUCT, then define filled:

$$\mathit{filled} = \max(0, \min(\mathit{count}_{\mathrm{con}}, \mathit{count}_{\mathrm{env}} - \mathit{start}))$$

where

$\mathit{count}_{\mathrm{env}} =$ count returned by MPI_GET_TYPE_ENVELOPE

$\mathit{start} =$ start passed in to MPI_GET_TYPE_CONTENTS

$\mathit{count}_{\mathrm{con}} =$ count passed in to MPI_GET_TYPE_CONTENTS

If the combiner returned by MPI_GET_TYPE_ENVELOPE is MPI_INDEXED, then filled elements, starting from start, of array_of_blocklengths and array_of_displacements that can be passed to MPI_TYPE_INDEXED to reconstruct an equivalent of datatype are returned in the first filled positions of array_blocklens and array_disps, respectively, and oldtype that can be passed to the same constructor function is returned in array_types[0]. array_hdisps is not modified.

If the combiner returned by MPI_GET_TYPE_ENVELOPE is MPI_HINDEXED, then filled elements, starting from start, of array_of_blocklengths and array_of_displacements that can be passed to MPI_TYPE_HINDEXED to reconstruct an equivalent of datatype are returned in the first filled positions of array_blocklens and array_hdisps, respectively, and oldtype that can be passed to the same constructor function is returned in array_types[0]. array_disps is not modified.

If the combiner returned by MPI_GET_TYPE_ENVELOPE is MPI_STRUCT, then filled elements, starting from start, of array_of_blocklengths, array_of_displacements, and array_of_types that can be passed to MPI_TYPE_STRUCT to reconstruct an equivalent of datatype are returned in the first filled positions of array_blocklens, array_hdisps, and array_types, respectively. array_disps is not modified.

> *Rationale.* FORTRAN users would be in trouble if MPI_GET_TYPE_CONTENTS did not have start and count and always returned the entire arrays at once. FORTRAN users can't malloc and hence have to pass in arrays to MPI_GET_TYPE_CONTENTS that are a fixed size – a size that might be too small for a particular datatype.
>
> In C and C++, users can access the entire arrays in a single call by providing dynamically allocated arrays whose length is the count returned by MPI_GET_TYPE_ENVELOPE. (*End of rationale.*)

MPI_GET_TYPE_CONTENTS may be used to decode a datatype where the information is used for purposes other than reencoding into a different datatype. For example, a user may wish to convert the datatype into their own representation. For this to be possible, the user must be able to determine what datatypes are returned in array_types. To accomplish this, the following function is provided:

MPI_TYPE_COMPARE(datatype1, datatype2, flag)

IN	datatype1	first datatype (handle)
IN	datatype2	second datatype (handle)
OUT	flag	true if datatypes are the same (logical)

```
int MPI_Type_compare(MPI_Datatype datatype1, MPI_Datatype datatype2, int
              *flag)
```

```
MPI_TYPE_COMPARE(DATATYPE1, DATATYPE2, FLAG)
    integer DATATYPE1, DATATYPE2
    LOGICAL FLAG
```

MPI_TYPE_COMPARE is limited to use with fundamental datatypes. Thus, it should only be used with datatypes which returned MPI_FUNDAMENTAL from MPI_GET_TYPE_CONTENTS. flag is true if both datatypes are the same fundamental one.

> *Rationale.* Limiting the use of MPI_TYPE_COMPARE to fundamental datatypes makes the implementation easier and seems to meet the need above. (*End of rationale.*)

> *Advice to users.* A common use of MPI_TYPE_COMPARE is to make one of the datatypes the one given by MPI_GET_TYPE_CONTENTS and the other be a fundamental datatype defined in mpi.h or mpif.h. By comparing to the possible predefined datatypes, one can determine which one the datatype represents. As with other handles in MPI, using a language comparison operator (== or .eq.) will not return the correct answer. It will compare the handle and not its contents. (*End of advice to users.*)

> *Advice to implementors.* An implementation must return a datatype from MPI_GET_TYPE_CONTENTS such that if MPI_GET_TYPE_ENVELOPE returned MPI_FUNDAMENTAL then using MPI_TYPE_COMPARE to figure out the predefined datatype as described above will work. (*End of advice to implementors.*)

Discussion:
An alternative is to allow any datatype. Do people see a need for this?

7.7.2 Datatype Encoding and Decoding

Current Status: no vote

It is useful, in some cases, for one process to be able to communicate a datatype to another process. Formally, two datatypes are said to be **equivalent** if they were both constructed by the same sequence of datatype constructor calls. Datatype communication should allow one process to create a datatype that is equivalent to a datatype defined on another process. To do so, one needs to encode the datatype opaque object, creating a representation that can be sent to another process and used at that other process to decode the datatype.

The datatype access functions in Section 7.7.1, page 147 can be used to decode a datatype into a user-specified form and type constructor functions can be used to create an equivalent datatype. Here, we encapsulate this functionality into datatype encode and datatype decode function calls. Aside from convenience, this avoids the encode and decode overhead in cases where it is sufficient to transfer an internal representation of the datatype. These functions also provide a mechanism for generating persistent stores of datatype information across program executions. A utility function for computing an upper bound on the size of the buffer required to hold a datatype encoding is also provided.

MPI_TYPE_ENCODE(datatype, buffer, buffsize, len, comm)

IN	datatype	datatype to be encoded (handle)
OUT	buffer	buffer where encoding is to be returned (choice)
IN	buffsize	size of buffer provided for encoding (integer)
OUT	len	size of resulting encoding string (integer)
IN	comm	communicator to be used for transfer of encoding (handle)

```
int MPI_Type_encode(MPI_Datatype datatype, void *buffer, int buffsize, int
             *len, MPI_Comm comm)
```

```
MPI_TYPE_ENCODE(DATATYPE, BUFFER, BUFFSIZE, LEN, COMM, IERROR)
    <type> BUFFER(*)
    INTEGER DATATYPE, BUFFSIZE, LEN, COMM, IERROR
```

Stores in buffer an encoding of datatype, and returns in len the length of the encoding string. This string can be transferred using type MPI_BYTE to any process in comm and decoded there to create an equivalent datatype. Alternatively, comm may be one of the special constant communicators described below. If so, the datatype encoding may be transferred to another process or to persistent storage.

The call fails if the buffer is not large enough to contain the encoding. In this case 0 is returned in len.

MPI_TYPE_ENCODE_SIZE(datatype, maxlen, comm)

IN	datatype	datatype to be encoded (handle)
OUT	maxlen	upper bound of size of resulting encoding string (integer)
IN	comm	communicator to be used for transfer of encoding (handle)

```
int MPI_Type_encode_size(MPI_Datatype datatype, int *maxlen, MPI_Comm comm)
```

```
MPI_TYPE_ENCODE_SIZE(DATATYPE, MAXLEN, COMM, IERROR)
    INTEGER DATATYPE, MAXLEN, COMM, IERROR
```

Stores in maxlen an upper bound of the buffer size required to store an encoding of datatype using communicator comm. comm may be one of the special constant communicators described below for internal or canonical datatype representation.

MPI_TYPE_DECODE(buffer, len, datatype, comm)

IN	buffer	buffer containing encoded datatype (choice)
IN	len	size of encoding string (integer)
OUT	datatype	decoded datatype (handle)
IN	comm	communicator used to transfer the encoding (handle)

```
int MPI_Type_decode(void *buffer, int buffsize, int *len, MPI_Datatype
             *datatype, MPI_Comm comm)
```

```
MPI_TYPE_DECODE(BUFFER, BUFFSIZE, LEN, DATATYPE, COMM, IERROR)
    <type> BUFFER(*)
    INTEGER BUFFSIZE, LEN, DATATYPE, COMM, IERROR
```

Decodes the string of length len in buffer and returns in datatype a handle to a committed datatype *equivalent* to that described by the string. The string should be created by a call to MPI_TYPE_ENCODE with communicator comm. If comm is not one of the two special constant communicators described below, then buffer should also be received by the decoding process with the same communicator used to encode buffer.

> *Rationale.* The argument comm is provided here for the same reasons it is provided in the MPI_PACK and MPI_UNPACK functions: It allows the implementation to select a more efficient encoding according to the target of the communication. In particular, if all processes in the group of comm run in the same environment, then one can merely copy the internal representation of the datatype when encoding or decoding. (*End of rationale.*)

Alternatives: These two functions merely provide a version of MPI_PACK and MPI_UNPACK that is specific to datatypes. It may be simpler just to overload this new functionality on the existing MPI_PACK and MPI_UNPACK functions, by providing a new datatype, for that purpose. I.e., MPI_PACK(datatype, 1, MPI_DATATYPE_HANDLE, outbuf, outsize, position, comm) encodes datatype and MPI_UNPACK(inbuf, insize, position, datatype, 1, MPI_DATATYPE_HANDLE, comm) decodes datatype. The vote at the last meeting was to do it with separate functions.

Universal Datatype Encoding

Current Status: no vote

It is useful, in some cases, to create a datatype encoding that is not specific to a communicator, but may be communicated to any other process. E.g., there may not be a communicator that encompasses all processes. Or, new processes, running on a different architecture, may join the computation later. A new constant communicator handle MPI_COMM_INTERNAL is provided, for that purpose. When a datatype is encoded with comm = MPI_COMM_INTERNAL then an encoding is created that can be decoded on any process in the same implementation. The datatype encoding must be decoded using comm = MPI_COMM_INTERNAL.

Canonical Datatype Encoding

Current Status: no vote

Some libraries can take advantage of datatype encodings so as to provide interoperability across MPI applications at various levels. To do so, canonical datatype encoding and data packing are required with well-defined representations. The former provides the information that can be extracted from a datatype using the datatype access functions in Section 7.7.1, page 147. This encoding is described in Section 7.7.3, page 153. A new constant communicator handle MPI_COMM_CANONICAL is provided for that purpose. When a datatype is encoded with comm = MPI_COMM_CANONICAL then an canonical encoding is created for the datatype. A canonical datatype can be decoded using comm = MPI_COMM_CANONICAL. For a description of canonical data packing, see MPI_DATA_CANONICAL in Section 10.5.

7.7.3 Canonical Datatype Representation

Current Status: no votes

Discussion: Missing types for C++.

As noted in Section 7.7.2, it is useful to create a datatype encoding that is not specific to any communicator but may be communicated to any other process. To do so, a canonical datatype encoding is required with well-defined representations. MPI-2 requires implementations to use the following representation to create MPI_COMM_CANONICAL encodings of datatypes in MPI_TYPE_ENCODE.

Buffer contents. A buffer packed by MPI_TYPE_ENCODE contains an ordered sequence of 8-bit MPI_BYTE values. The sequence is not NULL terminated as the length of the buffer is known from MPI_TYPE_ENCODE. Valid encoding values are given in Table 7.1. In order

#define MPI_Typerep_user	196
#define MPI_Typerep_vector	197
#define MPI_Typerep_hvector	198
#define MPI_Typerep_indexed	199
#define MPI_Typerep_hindexed	200
#define MPI_Typerep_struct	201
#define MPI_Typerep_lowerbound	202
#define MPI_Typerep_upperbound	203
#define MPI_Typerep_0	228
#define MPI_Typerep_1	229
#define MPI_Typerep_2	230
#define MPI_Typerep_3	231
#define MPI_Typerep_4	232
#define MPI_Typerep_5	233
#define MPI_Typerep_6	234
#define MPI_Typerep_7	235
#define MPI_Typerep_8	236
#define MPI_Typerep_9	237
#define MPI_Typerep_openparen	238
#define MPI_Typerep_closeparen	239
#define MPI_Typerep_colon	240
#define MPI_Typerep_slash	241
#define MPI_Typerep_minus	242
#define MPI_Typerep_byte	226
#define MPI_Typerep_packed	225
#define MPI_TYPE_INTEGER	1
#define MPI_TYPE_LOGICAL	2
#define MPI_TYPE_REAL	3
#define MPI_TYPE_DOUBLE_PRECISION	4
#define MPI_TYPE_COMPLEX	5
#define MPI_TYPE_CHARACTER	6
#define MPI_TYPE_CHAR	32
#define MPI_TYPE_UNSIGNED_CHAR	33
#define MPI_TYPE_SHORT	34
#define MPI_TYPE_UNSIGNED_SHORT	35
#define MPI_TYPE_INT	36
#define MPI_TYPE_LONG	38
#define MPI_TYPE_UNSIGNED_LONG	39
#define MPI_TYPE_FLOAT	42
#define MPI_TYPE_DOUBLE	43

Table 7.1: MPI-2 datatype encoding values. The byte code definitions are given for the C preprocessor. In Fortran they would be PARAMETER statements.

<table>
<tr><td>⟨digit⟩</td><td>::=</td><td>MPI_Typerep_0 | MPI_Typerep_1 | MPI_Typerep_2 |
MPI_Typerep_3 | MPI_Typerep_4 | MPI_Typerep_5 |
MPI_Typerep_6 | MPI_Typerep_7 | MPI_Typerep_8 |
MPI_Typerep_9</td></tr>
<tr><td>⟨Int⟩</td><td>::=</td><td>MPI_Typerep_minus ⟨digit⟩ | ⟨digit⟩ | ⟨Int⟩⟨digit⟩</td></tr>
<tr><td>⟨nonNegInt⟩</td><td>::=</td><td>⟨digit⟩ | ⟨nonNegInt⟩ ⟨digit⟩</td></tr>
<tr><td>⟨posDigit⟩</td><td>::=</td><td>MPI_Typerep_1 | MPI_Typerep_2 | MPI_Typerep_3 |
MPI_Typerep_4 | MPI_Typerep_5 | MPI_Typerep_6 |
MPI_Typerep_7 | MPI_Typerep_8 | MPI_Typerep_9</td></tr>
<tr><td>⟨posInt⟩</td><td>::=</td><td>⟨posDigit⟩ | ⟨posInt⟩ ⟨digit⟩</td></tr>
<tr><td>⟨typeDescC⟩</td><td>::=</td><td>MPI_Typerep_byte | MPI_Typerep_packed | MPI_TYPE_CHAR |
MPI_TYPE_UNSIGNED_CHAR | MPI_TYPE_SHORT |
MPI_TYPE_UNSIGNED_SHORT |
MPI_TYPE_INT | MPI_TYPE_LONG |
MPI_TYPE_UNSIGNED_LONG |
MPI_TYPE_FLOAT | MPI_TYPE_DOUBLE</td></tr>
<tr><td>⟨typeDescF77⟩</td><td>::=</td><td>MPI_TYPE_INTEGER | MPI_TYPE_LOGICAL |
MPI_TYPE_REAL | MPI_TYPE_DOUBLE_PRECISION |
MPI_TYPE_COMPLEX | MPI_TYPE_CHARACTER</td></tr>
<tr><td>⟨typeDescUser⟩</td><td>::=</td><td>MPI_Typerep_user⟨nonNegInt⟩ |
MPI_Typerep_user⟨nonNegInt⟩(⟨typeDesc⟩)</td></tr>
<tr><td>⟨typeDescBasic⟩</td><td>::=</td><td>⟨typeDescC⟩ | ⟨typeDescF77⟩ | ⟨typeDescUser⟩</td></tr>
<tr><td>⟨typeDescAdv⟩</td><td>::=</td><td>MPI_Typerep_upperbound | MPI_Typerep_lowerbound |
⟨typeDescBasic⟩</td></tr>
<tr><td>⟨typeDescVec⟩</td><td>::=</td><td>MPI_Typerep_vector:⟨typeDescBasic⟩:⟨posInt⟩:⟨posInt⟩:⟨Int⟩</td></tr>
<tr><td>⟨typeDescHVec⟩</td><td>::=</td><td>MPI_Typerep_hvector:⟨typeDescBasic⟩:⟨posInt⟩:⟨posInt⟩:⟨Int⟩</td></tr>
<tr><td>⟨typeDescIndex⟩</td><td>::=</td><td>MPI_Typerep_indexed:⟨typeDescBasic⟩:⟨posInt⟩:⟨posInt⟩/⟨Int⟩ |
⟨typeDescIndex⟩,⟨posInt⟩/⟨Int⟩</td></tr>
<tr><td>⟨typeDescHIndex⟩</td><td>::=</td><td>MPI_Typerep_hindexed:⟨typeDescBasic⟩:⟨posInt⟩:⟨posInt⟩/⟨Int⟩ |
⟨typeDescHIndex⟩,⟨posInt⟩/⟨Int⟩</td></tr>
<tr><td>⟨typeDescStruct⟩</td><td>::=</td><td>MPI_Typerep_struct:⟨posInt⟩:⟨typeDescAdv⟩/⟨posInt⟩/⟨Int⟩ |
⟨typeDescStruct⟩,⟨typeDescAdv⟩/⟨posInt⟩/⟨Int⟩</td></tr>
<tr><td>⟨typeDescSimple⟩</td><td>::=</td><td>⟨typeDescBasic⟩ | ⟨typeDescVec⟩ |
⟨typeDescHVec⟩ | ⟨typeDescIndex⟩ |
⟨typeDescHIndex⟩ | ⟨typeDescStruct⟩</td></tr>
<tr><td>⟨typeDesc⟩</td><td>::=</td><td>⟨posInt⟩ ⟨typeDescSimple⟩ | ⟨typeDescSimple⟩</td></tr>
</table>

Table 7.2: MPI-2 datatype encoding syntax. In this BNF, colons, parentheses, and slashes (:,(),/) are not the character representations but MPI_Typerep_colon, MPI_Typerep_openparen, MPI_Typerep_closeparen, and MPI_Typerep_slash.

that libraries layered on MPI can develop their own encoded datatype utility functions, users should be able to cast (in C) encoded values to unsigned int (equivalence to CHARACTER in Fortran) for use in logical expressions. For example, a user might wish to check if a certain buffer value is equivalent to a predefined value in Table 7.1.

Syntax. The ordering of values in a buffer packed by MPI_TYPE_ENCODE must follow the syntax conventions defined in Table 7.2 to instantiate a single occurrence of <typeDesc>. For user-defined types, the byte format begins with the value MPI_Typerep_user (196) followed by a sequence number followed by an open parentheses code (238). The top-level description follows ending with a close-parentheses code (239). The top level itself may be defined in terms of lower-level <typeDesc> definitions—thus: a datatype encoding instantiates one <typeDesc> which may in fact be defined by several other <typeDesc> occurrences within the encoded sequence. Any such occurrences are defined upon first use. The basic predefined data types appear in Table 7.2 as <typeDescC> for C and <typeDescF77> for the Fortran 77. See Table 7.1 for the actual values of these constants.

Examples

Missing:
The examples are from a slightly older version of this proposal. They need to be updated.

Type threeslice from example 3.29 on page 75 of MPI-1.1 (assume sizeof real is 4):

```
"U329(MPI_Typerep_hvector:U3291(MPI_Typerep_hvector:
      U3292(MPI_Typerep_vector:MPI_TYPE_REAL:9:1:2):9:1:400):9:1:40000)"
```

Type ltype from example 3.30 on page 77 of MPI-1.1:

```
"U330(MPI_Typerep_indexed:MPI_TYPE_REAL:100:1/99,101/98,202/97,...,10000/0)"
```

Type xpose from example 3.31 on page 77 of MPI-1.1 (assume sizeof real is 4):

```
"U331(MPI_Typerep_hindexed:U3311(MPI_Typerep_vector:MPI_TYPE_REAL:100:1:100)
                          :100:1:4)"
```

Type Particletype from example 3.33 on page 78 of MPI-1.1 (assume double alignment, sizeof int is 4, sizeof double is 8, base = 96):

```
"U333(MPI_Typerep_struct:4:MPI_TYPE_INT/1/0,MPI_TYPE_DOUBLE/6/8,
      MPI_TYPE_CHAR/7/56,MPI_Typerep_upperbound/1/64)"
```

Type Particletype from example 3.34 on page 79 of MPI-1.1 (assume sizeof addr particle is 96, double alignment, sizeof int is 4, sizeof double is 8):

```
"U334(MPI_Typerep_struct:3:MPI_TYPE_INT/1/96,
      MPI_TYPE_DOUBLE/6/104,MPI_TYPE_CHAR/7/152)"
```

Interlanguage Issues: The basic (predefined) data types are returned in the dialect of the creating language.

Advice to users. The following data types are not necessarily portable: hvector, hindexed, struct, and the use of Upper/Lower bound. When moving to systems with different data type sizes or different alignment rules, reproducing a data type from its character representation may not meet the user's expectations. (*End of advice to users.*)

7.8 Identifying Requests

Current Status: two votes

Profilers want to be able to determine what type of request has been passed to wait, test, or start. Furthermore, since certain actions can only be taken on certain types of requests, it is important to be able to determine the type of a request. This can be done with the following function:

MPI_REQUEST_TYPE(request, type_req)

IN	request	request (handle)
OUT	type_req	type of request (integer)

```
int MPI_Request_type(MPI_Request request, int *type_req)
```

```
MPI_REQUEST_TYPE(REQUEST, TYPE_REQ, IERROR)
    INTEGER REQUEST, TYPE_REQ, IERROR
```

This routine returns MPI_REQUEST_PERSISTENT, MPI_REQUEST_GENERALIZED, MPI_REQUEST_PT2PT, or MPI_REQUEST_OTHER if the request is persistent, generalized, point-to-point (send, recv, send/recv), or other form of request, respectively.

Discussion:
Do we want to return all the types of requests?

7.9 MPI and Threads

Current Status: no vote

This section specifies the interaction between MPI calls and preemptive Posix threads. MPI may be implemented in environments where threads are not supported, or perform poorly. Therefore, it is not required that all MPI implementations behave as specified in this section. The requirements here apply to **thread-compliant** implementations.

Discussion: The design is likely to be compatible with non-Posix thread packages as well, but I have not checked NT threads or threads on other systems.

We may want to designate a subset of this section as being mandatory on all MPI systems. E.g., we may require that signal_safe handler be supported.

7.9.1 General

Current Status: no vote

In a thread-compliant implementation, an MPI process is a Posix process, which may be multi-threaded. This process uses Posix pthread calls for spawning and synchronizing threads. Each thread can issue MPI calls; however, threads are not separately addressable: a rank in a send or receive call identifies a process, not a thread. A message sent to a process can be received by any thread in this process.

> *Rationale.* This model corresponds to the Posix model of interprocess communication: the fact that a process is multi-threaded, rather than single-threaded, does not affect the external interface of this process. MPI implementations where MPI processes are Posix threads 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 7.1 Process 0 consists of two threads. The first thread executes a blocking send call MPI_Send(buff1, count, type, 0, 0, comm), whereas the second thread executes a blocking receive call MPI_Recv(buff2, count, type, 0, 0, comm). I.e., the first thread sends a message that is received by the second thread. This communication should always succeed. According to the first requirement, the execution will correspond to some interleaving of the two calls. According to the second requirement, a call can only block the calling thread, and cannot prevent progress of the other thread. If the send call went ahead of the receive call, then the sending thread may block, but this will not prevent the receiving thread from executing. Thus, the receive call will occur. Once both calls occur, the communication is enabled, and both calls will complete. Note that, on the other hand, a single-threaded process that posts a send, followed by a matching receive, may deadlock. The progress requirement for multithreaded implementations is stronger, as a blocked call cannot prevent progress in other threads.

> *Advice to implementors.* MPI calls can be made thread-safe by executing only one at a time, e.g., by protecting MPI code with one process global lock. However, blocked operations cannot hold the lock, as this would prevent progress of other threads in the process. The lock is held only for the duration of an atomic, locally-completing suboperation such as posting a send or completing a send, and is freed in between. High-quality implementations will provide more concurrency in MPI execution, using fine-grained locking. Concurrency can also be achieved by having some of the MPI protocol executed by separate server threads.
>
> In order to support threads, the MPI library has to interact with the thread scheduler (the library has to keep track of threads blocked because of MPI calls and unblock them, as appropriate). (*End of advice to implementors.*)

7.9.2 Clarifications

Current Status: no vote

Multiple threads wait on same request. A program where two threads block, waiting on the same request, is erroneous. Similarly the same request cannot appear in the array of requests of two concurrent wait{any | some | all } calls.

> *Rationale.* This is consistent with the view that a multithreaded execution corresponds to an interleaving of the MPI calls: once a wait was posted on a request, no other wait can be posted on that same request in a single-threaded implementation. This constraint also simplifies implementation, as only one thread will be blocked on any communication event. (*End of rationale.*)

Probe A receive call that uses source and tag values returned by a preceding call to MPI_PROBE or MPI_IPROBE will receive the message matched by the probe call only if there was no other receive after the probe and before that receive. In a multithreaded environment, it is up to the user to enforce this condition using suitable mutual exclusion logic. This can be enforced by making sure that each communicator is used by only one thread on each process.

Collective calls Matching of collective calls on a communicator is done according to the order in which the calls are issued at each processor. If concurrent threads issue such calls on the same communicator, it is up to the user to make sure the calls are correctly ordered, using interthread synchronization.

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 current design allows the exception handler to be executed on the thread where the exception occurred. (*End of rationale.*)

7.9.3 Initialization

Current Status: no vote

A predefined attribute key MPI_THREADS is associated with MPI_COMM_WORLD. The attribute is set if the implementation is thread-compliant.

Discussion: We may use the attribute value to provide additional information on thread support, or use additional attribute keys for that purpose.

MPI_THREAD_INIT()

```
int MPI_Thread_init()
```

```
MPI_THREAD_INIT(IERROR)
    INTEGER IERROR
```

MPI_THREAD_INITIALIZED(flag)

OUT	flag	flag indicating if thread was initialized (Boolean)

```
int MPI_Thread_initialized(int *flag)
```

```
MPI_THREAD_INITIALIZED(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR
```

MPI_THREAD_FINALIZE()

```
int MPI_Thread_finalize()
```

```
MPI_THREAD_FINALIZE(IERROR)
    INTEGER IERROR
```

A thread must register with MPI with the call MPI_THREAD_INIT before it can issue MPI calls. There is one exception: the main thread that returned from the MPI_INIT call need not invoke MPI_THREAD_INIT; the MPI_INIT call initializes this thread for MPI communication.

The function MPI_THREAD_INITIALIZED can be used by a thread to find if MPI_THREAD_INIT has already been invoked by the thread. The call MPI_THREAD_FINALIZE can be used to detach a thread from MPI; a thread cannot make MPI calls after it calls

MPI_THREAD_FINALIZE. These calls are similar to the process calls MPI_INIT, MPI_INITIALIZED and MPI_FINALIZE. However, a thread can call MPI_THREAD_INIT anew after it called MPI_THREAD_FINALIZE, to register again with MPI. However, the same thread should not be reregistered with MPI unless it has already been finalized.

> *Rationale.* Various optimizations are possible when an MPI process is single-threaded, or when it is multithreaded, but only one thread invokes MPI calls. The thread registration functions provide MPI with the information needed to carry these optimizations. (*End of rationale.*)

> *Advice to users.* Like any other MPI call, the three calls listed above can occur only on a process that has invoked MPI_INIT, and before the invocation of MPI_FINALIZE; a call to MPI_THREAD_INIT cannot be substituted for a call to MPI_INIT. (*End of advice to users.*)

7.9.4 User-defined Handlers

Current Status: no vote

In many situations it is desirable to invoke, upon completion of a communication, a handler routine.

The function introduced below can also be used to implement new nonblocking calls, such as nonblocking communication or task spawning. It can also be used inside a generalized request.

MPI_POST_HANDLER(request, info, handler, extra_state, handler_type)

INOUT	request	communication request (handle)
IN	info	info object for handler attributes (handle)
IN	handler	handler (function)
IN	extra_state	extra state for handler (choice)
IN	handler_type	type of handler function (integer)

```
int MPI_Post_handler(MPI_Request request, MPI_Info info,
              MPI_Request_handler handler, void *extra_state, int
              handler_type)
```

```
MPI_POST_HANDLER(REQUEST, INFO, HANDLER, EXTRA_STATE, HANDLER_TYPE, IERROR)
    INTEGER REQUEST, INFO, EXTRA_STATE, HANDLER_TYPE, IERROR
    PROCEDURE HANDLER
```

Discussion:

There was an idea thrown out to have a flag to tell if the posting of the handler worked. Thus, if you were reposting you could find out if the request had already fired the handler so you can do the right thing either way.

Attaches handler to request. This can be a persistent, or nonpersistent request. The call may modify the request object, but does not modify the request handle.

The handler is defined in C by:

```
typedef int MPI_Request_handler(MPI_Request *request, void *extra_state);
```

A Fortran declaration is as follows:

```
SUBROUTINE REQUEST_HANDLER( REQUEST, EXTRA_STATE, IERR)
    INTEGER REQUEST, EXTRA_STATE, IERR
```

The call does not modify the request handle.

The handler is invoked as soon as the request completes, subsequent to its posting. If the request was already complete when the handler was posted, then the handler is invoked as soon as posted. When invoked, the handler is passed in request a handle to the request the handler was posted to, and in extra_state the extra state that was passed when the request was posted. The handler can access the status associated with the request by executing a call to MPI_GET_STATUS. If the request is persistent, then handler is invoked anew each time the request completes.

The invocation of the handler should behave as if a call is made to create a separate thread for handler execution, when the request completes.

The info argument specifies thread attributes for the created thread, such as priority, scheduling algorithm, stack size, etc. The list of supported attribute keys and attribute values, and the default attribute values are implementation-dependent. The following info keys are predefined, in correspondence with Posix thread attributes.

pthread_contentionscope Contention scope for thread. Values are PTHREAD_SCOPE_SYSTEM or PTHREAD_SCOPE_PROCESS.

pthread_inheritsched Specify how schedule attributes are set. One of PTHREAD_INHERIT_SCHED or PTHREAD_EXPLICIT_SCHED.

pthread_schedpolicy Scheduling policy for thread. One of SCHED_FIFO, SCHED_RR, or SCHED_OTHER.

pthread_priority Scheduling priority.

pthread_stacksize Minimum stack size.

pthread_stackaddr Stack address.

pthread_detachstate Determines whether the thread is attached or detached (don't ask me what that means). One of PTHREAD_CREATE_UNDETACHED or PTHREAD_CREATE_DETACHED.

Discussion: We may also want performance hints, such as it is recommended to dedicate a thread to the request (e.g., for persistent handles), or to dynamically allocate the thread when the request completes (e.g., for nonpersistent handles, or infrequent services).

The handler_type argument provides information on restrictions obeyed by the handler code. Possible values are

MPI_HANDLER_DEFAULT: Handler is thread-safe, but obeys no further restrictions.

MPI_HANDLER_SIGNAL_SAFE: Handler is signal-safe.

On some systems, it will be possible to implement more efficiently restricted handlers.

A function is signal-safe if it can be invoked from a signal-catching function. (The Posix terminology is *asynch-signal-safe*). For example, a signal-safe handler can not use mutex calls.

Discussion: The list above is a placeholder: we need to discuss what restrictions make sense for various implementations.

Handler code should be thread-safe. Mutual exclusion with other handler instances, or with computation code is achieved by using Posix thread synchronization calls.

Discussion:
Since we moved MPI thread synchronization calls to JOD how can one get portable thread-safe code. What if we allow handlers when there are no threads? The issue of safety are similar.

An MPI exception occurs if the handler returns with a value other than MPI_SUCCESS.

Discussion: Do we need an inquiry function to find if a handler is attached to a request?

Advice to users. A robust MPI implementation will not consume resources (threads) for posted handlers that have not yet been triggered. However, once a handler starts executing, then a thread is allocated to this execution; the thread is allocated even if the handler code blocks. Users should be careful not to exceed the limit on the number of concurrent threads in their system. (*End of advice to users.*)

Advice to implementors. Since the MPI_POST_HANDLER call does not return a thread id, implementations are not required to spawn a thread when the call occurs; rather than spawning a new thread for each handler instance, one can "recycle" a fixed set of threads, and allocate a thread to a handler only when the request has completed. (*End of advice to implementors.*)

The function MPI_POST_HANDLER can be called anew, with a request that already has an attached handler. The call will attach a new handler to the request. If the previous handler has already started executing, then the new handler will be invoked at the next activation of the request (never, if the request is not persistent). Otherwise, the new handler executes instead of the old handler.

If MPI_POST_HANDLER is called with handler = MPI_HANDLER_NULL then no handler is attached to the request upon return. This allows handlers to be detached.

The user may call MPI_CANCEL(request) on a request with an attached handler, with the usual semantics: either the cancel succeeds and no communication occurs at all, or the communication completes. In either case, the handler is invoked. It can check which was which, by using the call MPI_TEST_CANCELLED.

The user may call MPI_WAIT, MPI_TEST, or any of the derived functions on a request attached to a handler. The wait or test will succeed once the handler has completed execution.

Inside the handler body, the request can be updated by a call to MPI_POST_HANDLER. A new handler is attached that will be invoked at the next completion of the request. The request can also be queried by a call to MPI_GET_STATUS, restarted by a call to MPI_START or MPI_STARTALL, or freed by a call to MPI_REQUEST_FREE. It is erroneous to call MPI_WAIT, MPI_TEST or any of the other derived functions on this request, inside the handler body.

Discussion:
In MPI-1 it says a persistent request is not inactive until you do a wait/test. You cannot post a persistent request until it is inactive. The above semantics seem to make it impossible to repost a persistent request unless we either: 1) change it, or 2) redefine when a persistent request becomes inactive.

The request is not deallocated automatically when the handler completes. The deallocation has to be done by a call to MPI_REQUEST_FREE inside the handler body, or by a call such as MPI_WAIT outside the handler body.

Rationale. Generalized requests and handlers have overlap in their functionality. Both generalized requests and handlers can cause events to trigger when MPI events complete. Generalized requests can also create persistent versions of this capability. Thus, it is natural to ask why MPI-2 has handlers in addition to generalized requests. One reason is that handlers are viewed as simpler to use. Another reason is that an MPI implementation may handle these capabilities in different ways. A generalized request has a user-defined progress function (push_fn). A handler's progress is controlled by MPI. This may lead to situations where some systems can execute handlers on a communication coprocessor but need to execute the push_fn on the main processor. This could make handlers more efficient. On the other hand, it may be cheaper on some systems to treat all callbacks the same way and avoid interrupts. (This needs better wording.) On such a system, handlers may get converted into generalized requests for internal use in MPI. For these reasons, MPI-2 offers both capabilities. (*End of rationale.*)

Advice to implementors. The presence of handlers can cause unexpected complications for an MPI implementation. One potential complication is the fact it is possible to have a second handler become eligible to execute while another handler is still active. If the second handler is allowed to execute, it is possible that the same MPI call can be executed concurrently. One mechanism to overcome this is to make all MPI routines reentrant. A potential negative of this approach is that MPI routines may become slower to achieve this. Thus, the MPI calls would be slower even if no handlers are posted. A second approach is to only allow one handler to execute at a time. This requires that any handler(s) that could execute while another handler is active get queued for later execution. (*End of advice to implementors.*)

Advice to users. Users should remember that an MPI implementation can choose to run only one handler at a time or run multiple handlers at a time. In the former case, it can be dangerous to have handlers that execute for extended periods of time or block. While the handler is executing, no other handler can execute. If the first handler blocks, then the entire process can hang. If the first handler runs for a long period of time then the other handlers need to be queued and their progress will be blocked. This can happen even if the MPI implementation is running in a threaded environment. In the later case, it is possible that two handlers (or even the same handler) can run simultaneously. If there is a potential interaction between handlers, then locks or reentrant routines must be used to avoid potential errors.

In addition, users should keep in mind the potential problem with Fortran registers discussed in Section 2.5.2 on Page 10.

(*End of advice to users.*)

Advice to users. The design supports several styles of handler usage:

Autonomous server: the user does not access the request once the handler is posted. The handler will deallocate the nonpersistent request, when done. For a persistent request, the handler will restart the request. In the latter case, the handler represents a persistent server.

Autonomous, cancellable handler As before, but the request may be cancelled by the user when the service is terminated. The handler code should call MPI_GET_STATUS(request, status), followed by MPI_TEST_CANCELLED(status, flag) before it accesses the request status, or does any further operation on the request.

Communication with side effect: The user code handles the send or receive request the same way requests with no handlers are handled: a wait or test call completes the communication from the user's viewpoint. The handler code is an additional action that is taken, as part of the communication operation, before the user can complete the communication. The handler code can access the request nondestructively by calling MPI_GET_STATUS, but does not update or deallocate the request. This can be used to modify the semantics of send or receive operations.

(*End of advice to users.*)

Discussion:

Several people have expressed that handlers should work in all MPI implementations. Thus, we would need to require them whether or not threads are supported. There is a discussion in generalized requests about progress that generally applies here. Furthermore, if you have handlers without threads then how one locks values to protect critical sections becomes an issue.

Chapter 8

Language Bindings

Discussion:
The text has been changed to reflect the "new" C++ proposal (to include returning non-int values from functions, using exceptions for reporting errors, and expanding the MPI class hierarchy.

8.1 C++ Introduction

Current Status: Two votes

This section presents a complete C++ language interface for MPI. There are some issues specific to C++ that must considered in the design of this interface that go beyond the simple description of language bindings. In particular, in C++, we must be concerned with the design of objects and their interfaces, rather than just the design of a language-specific functional interface to MPI. Fortunately, the original design of MPI was based around the notion of objects, so a natural set of classes is already part of MPI.

Since the original design of MPI-1 did not include a C++ language interface, a complete list of C++ bindings for MPI-1 functions is provided in Appendix C. MPI-2 includes C++ bindings as part of its function specifications.

8.1.1 Design

Current Status: Two votes

The C++ language interface for MPI is designed according to the following criteria:

1. The C++ language interface consists of a small set of classes with a lightweight functional interface to MPI. The classes are based upon the fundamental MPI object types (e.g., communicator, group, etc.).
2. The MPI C++ language bindings provide a semantically correct interface to MPI.
3. There is a one-to-one mapping between MPI functions and their C++ bindings.
4. To the greatest extent possible, the C++ bindings for MPI functions are member functions of MPI classes.

Rationale. Providing a lightweight set of MPI objects that correspond to the basic MPI types is the best fit to MPI's implicit object-based design; methods can be supplied for these objects to realize MPI functionality. The existing C bindings can be used in C++ programs, but much of the expressive power of the C++ language is forfeited. On the other hand, while a comprehensive class library would make user programming more elegant, such a library it is not suitable as a language binding for MPI since a binding must provide a direct and unambiguous mapping to the specified functionality of MPI. (*End of rationale.*)

8.1.2 C++ Classes for MPI

Current Status: Two votes

All MPI classes, constants, and functions are declared within the scope of an `MPI namespace`. Thus, instead of the `MPI_` prefix that is used in C and Fortran, MPI functions essentially have an `MPI::` prefix.

Advice to implementors. Although `namespace` is officially part of the draft ANSI C++ standard, it not yet widely implemented in C++ compilers. Implementations using compilers without `namespace` may obtain the same scoping through the use of a non-instantiable `MPI` class. (To make the `MPI` class non-instantiable, all constructors must be `private`.) (*End of advice to implementors.*)

The members of the `MPI` namespace are those classes corresponding to objects implicitly used by MPI. An abbreviated definition of the `MPI` namespace and its member classes is as follows:

```
namespace MPI {
  class Comm {
    class Intra {
      class Graph;
      class Cart;
    };
    class Inter;
  };
  class Datatype;
  class Errhandler;
  class Group;
  class Op;
  class Request;
  class Status;
  // etc.
}
```

8.1.3 Class Member Functions for MPI

Current Status: Two votes

Besides the member functions which constitute the C++ language bindings for MPI, the C++ language interface has additional functions (as required by the C++ language). In particular, the C++ language interface must provide a constructor and destructor, an assignment operator, and comparison operators.

The bindings in Appendix C use some polymorphism to enforce compile-time type checking (e.g. for different types of `MPI::Request`'s and `MPI::Comm`'s), reference and `const` semantics (important features of the C++ language). Declarations (which apply to all `MPI` member classes) for construction, destruction, copying, assignment, comparison, and mixed-language inter-operability are also provided in Appendix C. The complete set of C++ language bindings for MPI-1 is presented in Appendix C. To maintain consistency with what has gone before, the binding definitions are given in the same order as given for the C bindings in [6].

All member functions (except for constructors and the assignment operator) of `MPI` member classes are virtual functions.

Rationale. Providing virtual member functions is an important part of design for inheritance. Virtual functions can be bound at run-time which allows users of libraries to re-define the behavior of objects already contained in a library. There is a small performance penalty that must be paid (the virtual function must be looked up before it can be called). However, users concerned about this performance penalty can force compile-time function binding. (*End of rationale.*)

Example 8.1 Example showing a derived MPI class.

```
class foo_comm : public MPI::Comm {
public:
  ...
  int Dup(const MPI::Comm& comm) {

    // Class library functionality

    MPI::Comm::Dup(comm);   // Call base MPI function

    // More class library functionality
  }
  ...
};
```

Advice to implementors. Implementors must be careful to avoid side effects from class libraries that use inheritance, especially in layered implementations. For example, if MPI_BCAST is implemented by repeated calls to MPI_SEND or MPI_RECV, the behavior of MPI_BCAST cannot be changed by derived communicator classes that might redefine MPI_SEND or MPI_RECV. The implementation of MPI_BCAST must explicitly use the MPI_SEND (or MPI_RECV) of the base `MPI::Comm` class. (*End of advice to implementors.*)

8.1.4 Semantics

Current Status: Two votes

The semantics of the member functions constituting the C++ language binding for MPI are specified by the MPI function description itself. Here, we specify the semantics for those portions of the C++ language interface that are not part of the language binding. In this subsection, functions are prototyped using the type MPI::⟨CLASS⟩ rather than listing each function for every MPI class; the word ⟨CLASS⟩ can be replaced with any valid MPI class name (e.g., Comm).

Construction / Destruction The default constructor and destructor are prototyped as follows:

```
MPI::<CLASS>(void)

~MPI::<CLASS>(void)
```

In terms of construction and destruction, the MPI user level objects behave like handles. Default constructors for all MPI objects create corresponding MPI::*_NULL handles. That is, when an MPI object is instantiated, it can be compared with its corresponding MPI::*_NULL. For example, `result` in the following code fragment will get the value MPI::IDENT.

```
void foo(void)
{
  MPI::Comm::Intra bar;

  if (bar.Compare(MPI::COMM_NULL) == MPI::IDENT)
    cout << "bar is MPI::COMM_NULL" << endl;
}
```

The destructor for each MPI user level object does *not* invoke the corresponding MPI::-*_free() function (if it exists).

Rationale. Destructors are not automatically invoked for the following reasons:

1. Automatic destruction contradicts the shallow-copy semantics of the MPI classes.
2. The model put forth in MPI makes memory allocation and deallocation the responsibility of the user, not the implementation.
3. Calling MPI::*_free upon destruction could have unintended side effects, including triggering collective operations (this also affects the copy, assignment, and construction semantics). In the following example, we would want neither `foo` nor `bar` to automatically invoke MPI::*_free upon exit from the function.

```
void foo(void)
{
  MPI::Comm::Intra foo(MPI::COMM_WORLD), bar;
  bar = MPI::COMM_WORLD.Dup();
  // rest of function
}
```

(*End of rationale.*)

Copy / Assignment The copy constructor and assignment operator are prototyped as follows:

```
MPI::<CLASS>(const MPI::<CLASS>& data)
```

```
MPI::<CLASS>& MPI::<CLASS>::operator=(const MPI::<CLASS>& data)
```

In terms of copying and assignment, the MPI user level objects behave like handles. Copy constructors perform handle-based (shallow) copies. MPI::Status objects are exceptions to this rule. These objects perform deep copies for assignment and copy construction.

> *Advice to implementors.* Each MPI user level object is likely to contain, by value or by reference, implementation-dependent state information. The assignment and copying of MPI object handles may simply copy this value (or reference). (*End of advice to implementors.*)

Example 8.2 Example using assignment operator. In this example, MPI::Comm::Dup() is *not* called for foo. The object foo is simply an alias for MPI::COMM_WORLD. On the other hand, bar is created with a call to MPI::Comm::Dup() and is therefore a different communicator than foo or MPI::COMM_WORLD.

```
MPI::Comm foo, bar;

foo = MPI::COMM_WORLD;
bar.Dup(MPI::COMM_WORLD);
```

Comparison The comparison operators are prototyped as follows:

```
bool MPI::<CLASS>::operator==(const MPI::<CLASS>& data) const
```

```
bool MPI::<CLASS>::operator!=(const MPI::<CLASS>& data) const
```

The member function operator==() returns true only when the handles reference the same internal MPI object, false otherwise. operator!=() returns the opposite of operator==().

Constants Constants are singleton objects and are declared const.

8.1.5 C++ Datatypes

Current Status: One vote

Table 8.1 lists all of the C++ basic MPI datatypes and their corresponding C and C++ datatypes, Table 8.2 lists all of the Fortran basic MPI datatypes and their corresponding Fortran 77 datatypes. Table 8.3 lists the C++ names for all other MPI datatypes.

MPI::BYTE and MPI::PACKED conform to the same restrictions as MPI_BYTE and MPI_PACKED, listed in sections 3.2.2 and 3.13 of MPI-1, respectively.

The following table defines groups of MPI basic datatypes:

C integer:	MPI::INT, MPI::LONG, MPI::SHORT. MPI::UNSIGNED_SHORT, MPI::UNSIGNED

MPI datatype	C datatype	C++ datatype
MPI::CHAR	signed char	signed char
MPI::SHORT	signed short int	signed short int
MPI::INT	signed int	signed int
MPI::LONG	signed long int	signed long int
MPI::UNSIGNED_CHAR	unsigned char	unsigned char
MPI::UNSIGNED_SHORT	unsigned short int	unsigned short int
MPI::UNSIGNED	unsigned int	unsigned int
MPI::UNSIGNED_LONG	unsigned long int	unsigned long int
MPI::FLOAT	float	float
MPI::DOUBLE	double	double
MPI::LONG_DOUBLE	long double	long double
MPI::BOOL		bool
MPI::COMPLEX		Complex<float>
MPI::DOUBLE_COMPLEX		Complex<double>
MPI::LONG_DOUBLE_COMPLEX		Complex<long double>
MPI::BYTE		
MPI::PACKED		

Table 8.1: C++ names for the MPI C and C++ basic datatypes, and their corresponding C/C++ datatypes.

MPI datatype	Fortran datatype
MPI::CHARACTER	CHARACTER(1)
MPI::INTEGER	INTEGER
MPI::REAL	REAL
MPI::DOUBLE_PRECISION	DOUBLE PRECISION
MPI::LOGICAL	LOGICAL
MPI::F_COMPLEX	COMPLEX
MPI::PACKED	

Table 8.2: C++ names for the MPI Fortran basic datatypes, and their corresponding Fortran 77 datatypes.

MPI datatype	Description
MPI::FLOAT_INT	C/C++ reduction type
MPI::DOUBLE_INT	C/C++ reduction type
MPI::LONG_INT	C/C++ reduction type
MPI::TWOINT	C/C++ reduction type
MPI::SHORT_INT	C/C++ reduction type
MPI::LONG_DOUBLE_INT	C/C++ reduction type
MPI::LONG_LONG_INT	Optional C/C++ type
MPI::TWOREAL	Fortran reduction type
MPI::TWODOUBLE_PRECISION	Fortran reduction type
MPI::TWOINTEGER	Fortran reduction type
MPI::F_DOUBLE_COMPLEX	Optional Fortran type
MPI::INTEGER1	Optional Fortran type
MPI::INTEGER2	Optional Fortran type
MPI::INTEGER4	Optional Fortran type
MPI::REAL2	Optional Fortran type
MPI::REAL4	Optional Fortran type
MPI::REAL8	Optional Fortran type

Table 8.3: C++ names for other MPI datatypes. Implementations may also define other optional types (e.g., `MPI::INTEGER8`).

	MPI::UNSIGNED_LONG
Fortran integer:	MPI::INTEGER
Floating point:	MPI::FLOAT, MPI::DOUBLE, MPI::REAL, MPI::DOUBLE_PRECISION, MPI::LONG_DOUBLE
Logical:	MPI::LOGICAL, MPI::BOOL
Complex:	MPI::F_COMPLEX, MPI::COMPLEX, MPI::F_DOUBLE_COMPLEX, MPI::DOUBLE_COMPLEX, MPI::LONG_DOUBLE_COMPLEX
Byte:	MPI::BYTE

Valid datatypes for each reduction operation is specified below in terms of the groups defined above.

Op	Allowed Types
MPI::MAX, MPI::MIN	C integer, Fortran integer, Floating point
MPI::SUM, MPI::PROD	C integer, Fortran integer, Floating point, Complex
MPI::LAND, MPI::LOR, MPI::LXOR	C integer, Logical
MPI::BAND, MPI::BOR, MPI::BXOR	C integer, Fortran integer, Byte

8.1.6 Mixed-language Inter-operability

Current Status: Two votes

The C++ language interface provides functions listed below for mixed-language interoperability. These functions provide for a seamless transition between C and C++.

```
MPI::<CLASS>& MPI::<CLASS>::operator=(const MPI_<CLASS>& data)

MPI::<CLASS>& MPI::<CLASS>::operator=(const MPI_<CLASS>* data)

MPI::<CLASS>(const MPI_<CLASS>& data)

MPI::<CLASS>(const MPI_<CLASS>* data)

MPI::<CLASS>::operator MPI_<CLASS>() const

MPI::<CLASS>::operator MPI_<CLASS>*()
```

These functions are discussed in Section 9.7.4.

8.1.7 Profiling

Current Status: One vote

This section specifies the requirements of a C++ profiling interface to MPI.

> *Advice to implementors.* Since the main goal of profiling is to intercept function calls from user code, it is the implementor's decision how to layer the underlying implementation to allow function calls to be intercepted and profiled. If an implementation of the MPI C++ bindings is layered on top of MPI bindings in another language (such as C), or if the C++ bindings are layered on top of a profiling interface in another language, no extra profiling interface is necessary because the underlying MPI implementation already meets the MPI profiling interface requirements.
>
> Native C++ MPI implementations that do not have access to other profiling interfaces must implement an interface that meets the requirements outlined in this section.
>
> High quality implementations can implement the interface outlined in this section in order to promote portable C++ profiling libraries. Implementors may wish to provide an option whether to build the C++ profiling interface or not; C++ implementations that are already layered on top of bindings in another language or another profiling interface will have to insert a third layer to implement the C++ profiling interface. (*End of advice to implementors.*)

To meet the requirements of the C++ MPI profiling interface, an implementation of the MPI functions *must*:

1. provide a mechanism through which all of the MPI defined functions may be accessed with a name shift. Thus all of the MPI functions (which normally start with the prefix "`MPI::`") should also be accessible with the prefix "`PMPI::`".

2. ensure that those MPI functions which are not replaced may still be linked into an executable image without causing name clashes.

3. document the implementation of different language bindings of the MPI interface if they are layered on top of each other, so that profiler developer knows whether they

must implement the profile interface for each binding, or can economize by implementing it only for the lowest level routines.

4. where the implementation of different language bindings is is done through a layered approach (e.g. the C++ binding is a set of "wrapper" functions which call the C implementation), ensure that these wrapper functions are separable from the rest of the library.

   This is necessary to allow a separate profiling library to be correctly implemented, since (at least with Unix linker semantics) the profiling library must contain these wrapper functions if it is to perform as expected. This requirement allows the author of the profiling library to extract these functions from the original MPI library and add them into the profiling library without bringing along any other unnecessary code.

5. provide a no-op routine MPI::Pcontrol in the MPI library.

Advice to implementors. Providing two entry points can be done using inheritance. The `PMPI` class contains the "real" entry points to each routine. The non-profiling version of the `MPI` class invokes the corresponding function in the `PMPI` class, while the profiling version of the `MPI` class can gather profiling information before (and/or after) calling the `PMPI` base class; this is very similar to the C profiling scheme.

The key to making the profiling work by simply re-linking programs is by having a header file that *declares* all the MPI functions. The functions must be *defined* elsewhere, and compiled into a library. MPI constants should be declared `static const` in the `MPI` class. For example, the following is an excerpt from a sample `mpi++.h` file:

Example 8.3 *Sample* `mpi++.h` *file.*

```
class PMPI {
public:
  class Comm {
  public:
    int Size(int& size);
  };
  // etc.
};

class MPI : public PMPI {
public:
  class Comm : public PMPI::Comm {
  public:
    int Size(int& size);
  };

  static const MPI::Comm COMM_WORLD;
  // etc.
};
```

The definitions of the functions must be in separate object files; the PMPI member functions and the non-profiling versions of the MPI member functions are compiled into libmpi++.a, while the profiling versions are compiled into libpmpi++.a. Note that the PMPI member functions and the MPI constants must be in different object files than the non-profiling MPI member functions in the libmpi++.a library to prevent multiple definitions of MPI member function names when linking both libmpi++.a and libpmpi++.a. For example:

Example 8.4 pmpi.cc, *to be compiled into* libmpi++.a.

```
int
PMPI::Comm::Size(int &size)
{
  // Implementation of MPI_COMM_SIZE
}
```

Example 8.5 constants.cc, *to be compiled into* libmpi++.a.

```
const MPI::COMM_WORLD;
```

Example 8.6 mpi_no_profile.cc, *to be compiled into* libmpi++.a.

```
int MPI::Comm::Size(int& size)
{
  return PMPI::Comm::Size(size);
}
```

Example 8.7 mpi_profile.cc, *to be compiled into* libpmpi++.a.

```
int MPI::Comm::Size(int& size)
{
  // Do profiling stuff
  int ret = PMPI::Comm::Size(size);
  // More profiling stuff
  return ret;
}
```

(*End of advice to implementors.*)

8.2 Fortran 90 Support

Current Status: No votes.

8.2.1 Overview

Fortran 90 is the current international Fortran standard, containing numerous features designed to make it a more "modern" language than Fortran 77. It seems natural that MPI should be able to take advantage of these new features with a set of bindings tailored to Fortran 90. MPI does not (yet) provide such a set of bindings because of a number of technical difficulties.

MPI defines three levels of Fortran support. The first two are described here, and the last is deferred to future standardization efforts. In the rest of this section, "Fortran" shall refer to Fortran 90 (or its successor) unless qualified.

1. **Basic Fortran Support** An implementation with this level of Fortran support provides the original Fortran (77) bindings specified in MPI-1, with small additional requirements specified in section 8.2.2.
2. **Extended Fortran Support** An implementation with this level of Fortran support provides Basic Fortran Support plus additional features to support Fortran 90 as described in section 8.2.3.
3. **Full Fortran support** This interface is not yet specified. Ideas under consideration by the MPI Forum are described in the MPI Journal of Development (section 16).

Discussion: Should Basic Fortran Support be part of MPI 1.2?

8.2.2 Basic Fortran Support

Because Fortran 90 is (for all practical purposes) a superset of Fortran 77, Fortran 90 (and future) programs can use the original Fortran interface. The following additional requirements are added:

1. Implementations are required to provide the file `mpif.h`, as described in the original MPI-1 specification.
2. `mpif.h` must be valid and equivalent for both fixed- and free- source form.

   Advice to implementors. To make `mpif.h` compatible with both fixed- and free-source forms, to allow automatic inclusion by preprocessors, and to allow extended fixed-form line length, it is recommended to meet requirement 2 by constructing `mpif.h` without any continuation lines. This should be possible because `mpif.h` contains only declarations, and because common block declarations can be split among several lines. To support Fortran 77 as well as Fortran 90, it may be necessary to eliminate all comments from `mpif.h`. (*End of advice to implementors.*)

As noted in the original MPI specification, the interface violates the Fortran standard in several ways. While these caused few problems for Fortran 77 programs, they become much more significant for Fortran 90 programs, so that users must exercise care when using new Fortran 90 features. The violations were originally adopted and have been retained because they are important to the usability of MPI. The rest of this section describes the potential problems in detail. It supercedes and replaces the discussion of Fortran bindings in the original MPI specification (for Fortran 90, not Fortran 77).

The following MPI features are inconsistent with Fortran 90.

1. An MPI subroutine with a choice argument may be called with any argument type.
2. An MPI subroutine with an assumed-size array dummy argument (declared `buf(*)`) may be passed an actual scalar argument.
3. Many MPI routines assume that actual arguments are passed by address.
4. Many MPI routines (non-blocking routines, routines that create a persistent requests, and a few others) assume that the address passed is that of the original data, not that of a temporary copy. Moreover, an MPI implementation may assume that it is safe to read or modify this data after the routine has returned, and outside the flow of control of the program.

Problems due to strong typing

All MPI functions with choice arguments associate actual arguments of different Fortran datatypes with the same dummy argument. This is not allowed by Fortran 77, and in Fortran 90 is technically only allowed if the function is overloaded with a different function for each type. In C, the `void*` formal arguments avoid these problems.

The following code fragment is technically illegal and may generate a compile-time error.

```
integer i
real    x
...
call mpi_send(x, 1, MPI_REAL, ...)
call mpi_send(i, 1, MPI_INTEGER, ...)
```

In practice, it is rare for compilers to do more than issue a warning, though there is concern Fortran 90 compilers are more likely to return errors.

It is also technically illegal in Fortran to pass a scalar actual argument to an array dummy argument. Thus the following code fragment may generate an error since the **buf** argument to MPI_SEND is declared as an assumed-size array `<type> buf(*)`.

```
integer a
call mpi_send(a, 1, MPI_INTEGER, ...)
```

Advice to users. In the unlikely event that you run into one of the problems related to type checking, you may be able to work around it by using a compiler flag, by compiling separately, or by using an MPI implementation with Extended Fortran Support as described in section 8.2.3. An alternative is to use the EQUIVALENCE statement to create another variable with a type accepted by the compiler. (*End of advice to users.*)

Problems due to data copying and sequence association

Implicit in MPI is the idea of a contiguous chunk of memory accessible through a linear address space. MPI copies data to and from this memory. An MPI program specifies the location of data by providing memory addresses and offsets. In the C language, sequence association rules plus pointers provide all the necessary low-level structure.

In Fortran 90, user data is not necessarily stored contiguously (sequence association). Because MPI dummy buffer arguments are assumed-size arrays (e.g., `<type> buf(*)`) Fortran guarantees that data will be contiguous in the subroutine, so that non-contiguous data may be copied to a temporary and copied back out. Fortran (77 and 90) semantics allow compilers to *always* make a temporary copy, but this is rarely if ever done by Fortran 77 compilers because data is already contiguous. In a hypothetical Fortran 90 binding with assumed-shape arrays (e.g. `<type> buf(:)`) the compiler would be less likely (though not forbidden) to make a temporary copy, but the data would not then be contiguous in memory.

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 subsection `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 data to the address of the temporary, which no longer exists.

Fortran 90 derived datatypes

For various technical reasons, it is not allowed to pass a derived datatype to an MPI routine.

Discussion: The sequence attribute may provide a solution to this problem. Unfortunately, explicit interfaces would (?) undo the solution. So to allow derived datatypes to work, we may have to disallow explicit interfaces, or v.v. See discussion on mpi-bind.

Safe Fortran 90/MPI Programming Practice

Advice to users. MPI does not (yet) provide general solutions to the pitfalls described above. It is the application's responsibility to avoid them. The following guidelines will usually ensure that a program can run correctly, but in general it is the application-developer's responsibility to be aware of potential problems and avoid them. (*End of advice to users.*)

Advice to implementors. A high-quality implementation will document the rules a compiler uses for generating temporary copies and explain how this affects safe MPI usage. (*End of advice to implementors.*)

Care should be taken when passing data to the following routines:

1. Nonblocking routines

2. Routines that create persistent request
3. MPI_BUFFER_ATTACH
4. MPI_BUFFER_DETACH
5. MPI_ADDRESS

Users should be aware of the possibility that the compiler may create a temporary copy whenever the data being passed is

1. an array section (created either with subscript triplet notation or vector subscript notation)
2. an assumed-shape array
3. a Fortran 90 pointer
4. the result of an array-valued function (e.g. `matmul(a, b)`)
5. the result of an expression (e.g. `a+b`).

An alternative to using array sections is to create a datatype that describes an array section and pass the array directly.

8.2.3 Extended Fortran Support

Implementations with Extended Fortran support must provide:

1. An `mpi` module
2. A mechanism to prevent fatal type mismatch errors for MPI routines with choice arguments.
3. A new function MPI_TYPE_F90.

Applications may use either the `mpi` module or the `mpif.h` include file. An implementation may require use of the module to prevent type mismatch errors.

> *Advice to users.* It is recommended to use the `mpi` module even if it is not necessary to use it to avoid type mismatch errors on a particular system. Using a module provides several potential advantages over using an include file. (*End of advice to users.*)

The `mpi` module

An MPI implementation must provide a module named `mpi` that can be USEd in a Fortran 90 program. This module must:

- Define all named MPI constants
- Declare MPI functions that return a value.

An MPI implementation may provide in the `mpi` module other features that enhance the usability of MPI while maintaining adherence to the standard. For example, it may:

- Provide interfaces for all or for a subset of MPI routines.
- Provide INTENT information in these interface blocks.

Alternatives: We may want to only require the interface blocks, instead of just allowing them.

Missing: Do we need to specify complete bindings with INTENT?

It must be possible to link together routines that USE `mpi` and INCLUDE `mpif.h`.

No type mismatch problems for subroutines with choice arguments

An MPI implementation is required to provide a mechanism to ensure that MPI choice arguments do not cause compile-time or run-time errors due to type mismatch. An MPI implementation may require applications to use the `mpi` module, or require that it be compiled with a particular compiler flag, in order to avoid type mismatch problems.

Advice to implementors. In the case where the compiler does not generate errors, nothing needs to be done to the existing interface. In the case where the compiler may generate errors, a set of overloaded functions may be used. See the paper of M. Hennecke (add reference). Even if the compiler does not generate errors, explicit interfaces for all routines would be useful for detecting errors in the argument list. Also, explicit interfaces which give INTENT information can reduce the amount of copying for BUF(*) arguments. (*End of advice to implementors.*)

Discussion: There are various restrictions we might want to place on functions with two choice arguments. Do we want to consider these?

Missing: Need to look into potential issues with MPI_OP_CREATE.

Constructor Routine for Basic Fortran 90 Datatypes

The following functions makes all KIND-parameterized variants of the basic Fortran 90 datatypes available to MPI applications.

MPI_TYPE_F90(intrinsictype, kind, newtype)

IN	intrinsictype	an MPI datatype corresponding to a Fortran 90 intrinsic datatype (handle)
IN	kind	a Fortran 90 kind type parameter (integer)
OUT	newtype	the requested MPI datatype (handle)

```
int MPI_Type_f90(MPI_Datatype intrinsictype, integer kind, MPI_Datatype
            newtype)
```

```
MPI_TYPE_F90(INTRINSICTYPE, KIND, NEWTYPE, IERR)
    INTEGER INTRINSICTYPE, KIND, NEWTYPE, IERR
```

This function takes a Fortran 90 intrinsic type of default kind (one of MPI_REAL, MPI_LOGICAL, MPI_CHARACTER, MPI_INTEGER, MPI_COMPLEX), a Fortran 90 KIND parameter, and returns an MPI datatype corresponding to that type and kind.

Example:

```
integer       longtype, quadtype
integer(long) ii
real(selected_real_kind(30)) x
MPI_TYPE_F90(MPI_INTEGER, kind(ii), longtype)
MPI_TYPE_F90(MPI_REAL, kind(x), quadtype)
...

MPI_SEND(ii, 1, longtype, ...)
MPI_SEND(x,  1, quadtype, ...)
```

Chapter 9

Miscellany

There are a number of topics that do not fit conveniently into the other chapters. We collect them here.

9.1 Portable MPI Process Startup

Current Status: Passed twice.

A number of implementations of MPI-1 provide a startup command for MPI programs that is of the form

```
mpirun <mpirun arguments> <program> <program arguments>
```

Separating the command to start the program from the program itself provides flexibility, particularly for network and heterogeneous implementations. For example, the startup script need not run on one of the machines that will be executing the MPI program itself.

Having a standard startup mechanism also extends the portability of Fortran, C, and C++ programs one step further, to the command lines and scripts that manage them. For example, a validation suite script that runs hundreds of programs can be a portable script if it is written using `mpirun`. In order that the "standard" command not be confused with existing practice, which is not standard and not portable among implementations, instead of `mpirun` we specify `mpiexec`.

It is proposed that

```
mpiexec -n <numprocs> <program>
```

be at least one way to start `<program>` with an initial MPI_COMM_WORLD whose group contains `<numprocs>` processes. Other arguments to `mpiexec` may be implementation-dependent.

Currently this is proposed as advice to implementors, rather than as a required part of MPI-2. It is not suggested that this be the only way to start MPI programs.

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_SPAWN (See Section 4.3.5).

Analogous to MPI_SPAWN, we have

```
mpiexec -n    <maxprocs>
       -soft  <        >
       -host  <        >
       -arch  <        >
       -wdir  <        >
       -path  <        >
       -file  <        >
        ...
       <command line>
```

for the case where a single command line for the application program and its arguments will suffice. For the case corresponding to MPI_SPAWN_MULTIPLE there are two possible formats:

Form A:

```
mpiexec { <above arguments> } : { ... } : { ... } : ... : { ... }
```

As with MPI_SPAWN, all the arguments are optional. (Even the `-n x` argument is optional; the default is implementation dependent. It might be `1`, it might be taken from an environment variable, or it might be specified at compile time). The names of the arguments are taken from the keys in the `info` argument to MPI_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.

(*End of advice to implementors.*)

Example 9.1 *Start 16 instances of* `myprog` *on the current or default machine:*

```
mpiexec -n 16 myprog
```

Example 9.2 *Start 10 processes on the machine called* `ferrari`*:*

```
mpiexec -n 10 -host ferrari myprog
```

Example 9.3 *Start the* `ocean` *program on five suns and the* `atmos` *program on 10 RS/6000's:*

```
mpiexec -n 5 -arch sun ocean : -n 10 -arch rs6000 atmos
```

Example 9.4 *Start three copies of the same program with different command-line arguments:*

```
mpiexec myprog infile1 : myprog infile2 : myprog infile3
```

Example 9.5 *Start the* `ocean` *program on five suns and the* `atmos` *program on 10 RS/6000's (Form B):*

```
mpiexec -configfile myfile
```

where `myfile` *contains*

```
-n 5  -arch sun    ocean
-n 10 -arch rs6000 atmos
```

9.2 Convenient Form of MPI_Type_indexed_block

Current Status: Passed twice.

This function allows constant blocksize and arbitrary displacements in the interface to MPI_TYPE_INDEXED. The routine now takes array_of_blocklengths and array_of_displacements as arguments. There are many codes using indirect addressing arising from unstructured grids where the blocksize is always 1 (gather/scatter). We propose this interface, allowing for constant blocksize and arbitrary displacements.

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

IN	count	length of array of displacements
IN	blocklength	size of block
IN	array_of_displacements	array of displacements
IN	oldtype	old datatype
OUT	newtype	new datatype

```
int MPI_Type_Indexed_Block( int count, int blocklength,
             int* array_of_displacements, MPI_Datatype oldtype,
             MPI_Datatype* newtype )
```

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

9.3 Treatment of MPI_Status

The following proposals add to, but do not change, the functionality associated with MPI_STATUS.

9.3.1 Passing MPI_STATUS_IGNORE for MPI_Status

Current Status: Passed twice.

Every call to MPI_RECV includes a status argument, where the system can return details about the message received. There are also a number of other MPI calls, particularly in MPI-2, where status is returned. An object of type MPI_STATUS is not an MPI opaque object; its structure is declared in `mpi.h` and `mpif.h`, and it exists in the users' program. In many cases application programs are constructed so that it is unnecessary for them to examine the status fields. In these cases it is a waste for the user to allocate a status object, and it is particularly wasteful for the MPI implementation to fill in fields in this object.

To cope with this problem, there is a pre-defined constant, MPI_STATUS_IGNORE, which, when passed to a receive or test function, informs the implementation that the status fields are not to be filled in. Note that MPI_STATUS_IGNORE is not a special type of MPI_STATUS object; rather, it is a special value for the argument. That is, in C one would expect it to be NULL, not the address of a special MPI_STATUS.

In general, this optimization can apply to all functions for which MPI_Status or an array of MPI_Status's is an argument. These are all the various forms of MPI_RECV, MPI_TEST, and MPI_WAIT. When an array is passed, as in the _ANY and _ALL functions, a separate constant, MPI_STATUSES_IGNORE, is passed for the array argument.

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 _ANY and _ALL 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.

9.3.2 Non-destructive Test of MPI_Status

Current Status: Passed twice.

This call is useful for accessing the information associated with a request, without deleting the request (in case the user is expected to access it after the handler). It allows one to layer libraries more conveniently, since multiple layers of software may access the same completed request and extract from it the status information. This will also be important for language interoperability if we decide that status objects cannot be transferred across language boundaries, since with this function the request object can be transferred instead.

MPI_GET_STATUS(request, flag, status)

IN	request	an MPI_REQUEST object
OUT	flag	flag, same as from MPI_TEST
OUT	status	MPI_STATUS object if flag is true

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

```
MPI_GET_STATUS( REQUEST, FLAG, STATUS, IERROR)
    INTEGER REQUEST, FLAG, STATUS(MPI_STATUS_SIZE), IERROR
```

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

9.4 Error Class for Invalid Keyval

Current Status: Passed twice.

Key values for attributes are system-allocated, by MPI_KEYVAL_CREATE. Only such values can be passed to the functions that use key values as input arguments. In order to signal that an erroneous key value has been passed to one of these functions, there is a new MPI error class: MPI_ERR_KEYVAL. It can be returned by MPI_Attr_put, MPI_Attr_get, MPI_Attr_delete, MPI_Keyval_free, MPI_Comm_dup, and MPI_Comm_free. The last two are because keyval is an argument to the copy and delete functions for attributes.

9.5 Determining Whether MPI Has Finished

Current Status: Passed once.

One of the goals of MPI was to allow for layered libraries. In order for a library to do this cleanly, it needs to know if MPI is active. In MPI-1 the function MPI_Initialized was provided to tell if MPI had been initialized. The problem arises in knowing if MPI has been finalized. Once MPI has been finalized it is no longer active and cannot be restarted. A library needs to be able to determine this to act accordingly. To achieve this the following function is needed:

MPI_FINALIZED(flag)

OUT	flag	true if MPI was finalized (logical).

```
int MPI_Finalized(int *flag)
```

```
MPI_FINALIZED(flag, IERROR))
    FLAG, IERROR
```

This function returns true if MPI_FINALIZE has completed. It is legal to call MPI_FINALIZED before MPI_INIT and after MPI_FINALIZE.

9.6 Handle Types

The new MPI opaque object MPI_Handle is the union of many handles in MPI. The opaque object MPI_Handle_type in C and HANDLE_KIND in Fortran indicate the handle type in a function call. The allowed values are:

MPI_DATATYPE_HANDLE

MPI_COMM_HANDLE

MPI_GROUP_HANDLE

MPI_REQUEST_HANDLE

MPI_FILE_HANDLE

MPI_WIN_HANDLE

9.7 Language Interoperability

Current Status: Passed twice, unless marked otherwise – C++ functions passed once.

9.7.1 Introduction

It is not uncommon for library developers to use one language to develop an applications library to be used by applications developers using another language. MPI currently supports ANSI C, ANSI C++, and Fortran 77 bindings. It should be possible for applications in any of the supported languages to call MPI-related functions in another language.

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, intercommunications should be possible between applications written in any language.

There are several issues that need to be addressed, to achieve interoperability.

Initialization We need to specify how the MPI environment is initialized for all languages.

Communication of MPI opaque objects We need to specify how are MPI object handles passed between languages. We also need to specify what happens when an MPI object is accessed in one language, to retrieve information set with this object in another language.

Interlanguage communication We need to specify how messages sent in one language can be received in another language.

It is highly desirable that the solution for interlanguage interoperability be extendable to new languages, should MPI bindings be defined for such languages.

9.7.2 Assumptions

We assume that conventions exist for MPI programs to call functions in MPI programs of another language. These conventions specify how to link routines in different langauges into one program, how to call functions in a different language, how to pass arguments between languages, and the correspondence between basic data types of different languages. In general, these conventions will be implementation dependent (however, this is an ongoing effort to standardize a C calling interface in Fortran). Furthernote, not every basic datatype may have a matching type in other languages. For example, C/C++ character strings may not be compatible with Fortran CHARACTER variables. However, we assume that a Fortran INTEGER, as well as a (sequence associated) Fortan array of INTEGERs can be passed to a C or C++ program. We also assume that Fortran, C, and C++ have address sized integers.

9.7.3 Initialization

A call to MPI_INIT(), from any language, initializes MPI for execution in all languages.

> *Advice to users.* Certain implementations use the (inout) argc, argv arguments of the C/C++ version of MPI_INIT in order to propagate the right value for argc, argv to all executing processes. Use of the Fortran version of MPI_INIT to initialize MPI may result in a loss of this ability. (*End of advice to users.*)

The function MPI_INITIALIZED returns the same answer in all languages.

The function MPI_FINALIZE finalizes the MPI environments for all languages.

The function MPI_FINALIZED returns the same answer in all languages.

The function MPI_ABORT kill processes, irrespective of the language used by the caller or by the processes killed.

The MPI environment is initialized in the same manner for all languages by MPI_INIT. I.e., MPI_COMM_WORLD carries the same information regardless of language: same processes, same environmental attributes, same error handlers.

> *Advice to users.* The use of several languages in one MPI program may require the use of special options at compile and/or link time. (*End of advice to users.*)

> *Advice to implementors.* Implementations may selectively link language specific MPI libraries only to codes that need them, so as not to increase the size of binaries for codes that use only one language. The MPI initialization code need perform initialization for a language only if that language library is loaded. (*End of advice to implementors.*)

9.7.4 Transfer of handles

Handles are passed across languages by using an explicit C wrapper to convert handles. The conversion is automatic between C and C++.

The data type MPI_Fint is provided in C/C++ for an integer of the size that matches a Fortran INTEGER; usually, MPI_Fint will be equivalent to int. The datatype MPI_Handle is provided in C/C++ for handles to opaque objects. Any handle type (such as MPI_Comm, MPI_Datatype, etc.). can be converted to this type, and back, without loss

Note that the existence of such univerral handle type is a slight restriction on C/C++ MPI bindings.

> *Advice to implementors.* If C handles are pointers then MPI_Handle can be void*. (*End of advice to implementors.*)

The following two functions are provided in C to convert from a Fortran handle (which is an integer) to a C handle, and vice versa.

```
MPI_Handle MPI_F2C_handle( MPI_Fint f_handle, int handle_kind)

MPI_Fint MPI_C2F_handle( MPI_Handle c_handle, int handle_kind)
```

If f_handle is a valid Fortran handle to an opaque object, and handle_kind specifies the type of object, then MPI_F2C_handle returns a valid C handle to that same object; if f_handle is a null Fortran handle, then MPI_F2C_handle returns a null C handle; if f_handle is an invalid Fortran handle, then MPI_F2C_handle returns an invalid C handle; and similarly, for MPI_C2F_handle.

The argument handle_kind specifies the type of handle encoded or decoded. It can have one of the following values: MPI_COMM_HANDLE, MPI_DATATYPE_HANDLE, MPI_FILE_HANDLE, MPI_GROUP_HANDLE, MPI_REQUEST_HANDLE, OR MPI_WIN_HANDLE.

Rationale. The handle_kind argument allows for implementations that use different handle allocation mechansims in different languages, for different types of objects. E.g., an implementation where handles are table indices, and where a different table is kept for each language and each type of object. (*End of rationale.*)

Discussion: Is this rationale strong enough?

Example 9.6 The example below illustrates how the Fortran MPI function MPI_TYPE_COMMIT can be implemented by wrapping the C MPI function MPI_Type_commit with a C wrapper to do handle conversions.

```
! FORTRAN PROCEDURE
SUBROUTINE MPI_TYPE_COMMIT( DATATYPE, IERR)
INTEGER DATATYPE, IERR
CALL MPI_X_TYPE_COMMIT(DATATYPE, IERR)
RETURN
END

! C wrapper

void MPI_X_TYPE_COMMIT( MPI_Fint *f_handle, *ierr)
{
MPI_Datatype datatype;

datatype = (MPI_Datatype)MPI_F2C_handle( *f_handle, MPI_DATATYPE_HANDLE);
*ierr = (MPI_Fint)MPI_Type_commit( &datatype);
*f_handle = MPI_C2F_handle( (MPI_Handle)datatype,
MPI_DATATYPE_HANDLE);
return();
}
```

The same approach can be used for all other MPI functions. The call to MPI_INT2HANDLE (resp. MPI_HANDLE2INT) can be omitted when the handle is an OUT (resp. IN) argument, rather than INOUT.

Rationale. The design here provides a convenient solution for the prevalent case, where a C wrapper is used to allow Fortran code to call a C library, or C code to call a Fortran library. The use of C wrappers is much more likely than the use of

Fortran wrappers, because it is much more likely that a variable of type INTEGER can be passed to C, than a C handle can be passed to Fortran.

The use of conversion functions, rather than procedures, allow us to use inlining, especially in the case where these functions are 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.*)

Advice to users. The user needs to coerce the result returned by MPI_int2handle from MPI_Handle to a specific handle type. (*End of advice to users.*)

Current Status: Two votes

C and C++ The C++ language interface provides the functions listed below for mixed-language inter-operability. The token <CLASS> is used below to indicate any valid MPI opaque handle name (e.g. Comm).

The following two operator=() functions allow assignment from both a C MPI handle and a pointer to a C MPI handle, respectively.

```
MPI::<CLASS>& MPI::<CLASS>::operator=(const MPI_<CLASS>& data)
```

The constructor below creates a C++ MPI object from a C MPI handle. This allows the automatic promotion of a C MPI handle to a C++ MPI handle.

```
MPI::<CLASS>(const MPI_<CLASS>& data)
```

Example 9.7 In order for a C program to use a C++ library, the C++ library must export a C interface that provides appropriate conversions before invoking the underlying C++ library call. This example shows a C interface function that invokes a C++ library call with a C communicator; the communicator is automatically promoted to a C++ handle when the underlying C++ function is invoked.

```
void cpp_lib_call(MPI::Comm\& cpp_comm);

extern "C" {
  void c_interface(MPI_Comm c_comm);
}

void c_interface(MPI_Comm c_comm)
{
  cpp_lib_call(c_comm);
}
```

The following function allows conversion from C++ objects to C MPI handles. In this case, the casting operator is overloaded to provide the functionality.

```
MPI::<CLASS>::operator MPI_<CLASS>() const
```

Example 9.8 A C library routine is called from a C++ program. The C library routine is prototyped to take an MPI_Comm as an argument.

```
extern "C" {
  void c_lib_call(MPI_Comm c_comm);
}

void cpp_function()
{
  MPI::Comm::Intra cpp_comm(MPI::COMM_WORLD.Dup());
  c_lib_call(cpp_comm);
}
```

Rationale. Providing conversion from C to C++ via constructors and from C++ to C via casting allows the compiler to make automatic conversions. Calling C from C++ becomes trivial, as does the provision of a C or Fortran interface to a C++ library. (*End of rationale.*)

Finally, the following two functions promote a *pointer* to a C handle to a C++ class, and cast a C++ class to a *pointer* to a C handle.

```
MPI::<CLASS>(const MPI_<CLASS>* data)

MPI::<CLASS>::operator MPI_<CLASS>*()
```

Example 9.9 With these functions, a C++ program can call MPI_Cancel() as follows:

```
  MPI::Request cpp_request;
  ...
  MPI_Cancel(cpp_request);
```

Rationale. While the provided casting and promotion functions allow the seamless conversions between C and C++ MPI handles, there are several situations in the MPI C bindings (and third party C library calls) where a *pointer* to an MPI C handle is required (e.g., MPI_CANCEL). Currently, a C++ user would have to explicitly cast the C++ handle to C variable, and then use the & operator to get its address. For example, the C++ syntax to call the C MPI_Cancel() function is as follows:

```
MPI::Request cpp_request;
MPI_Request c_request;
...
c_request = cpp_request;
MPI_Cancel(&c_request);
```

(*End of rationale.*)

It is important to note that all C++ objects and their corresponding C handles can be used interchangeably by an application. For example, an application can cache an attribute on MPI_COMM_WORLD and later retrieve it from MPI::COMM_WORLD.

9.7.5 MPI Opaque objects

In general, 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_EXTENT will return the same information in all three 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 basic datatypes can be used in datatype constructors in any language. If a datatype is committed, it can be used for communication in any languages.

The function MPI_ADDRESS returns the same value in all languages. On the other hand, we do not require that the constant MPI_BOTTOM has the same value in all languages.

Example 9.10

```
! FORTRAN CODE
REAL R(5)
INTEGER TYPE, ADDR, IERR

! create an absolute datatype for array R
CALL MPI_ADDRESS( R, ADDR, IERR)
CALL MPI_TYPE_STRUCT(1, 5, ADDR, MPI_REAL, TYPE, IERR)
CALL C_ROUTINE(TYPE)

/* C code */

void C_ROUTINE(MPI_Fint *ftype)
{
int count = 5;
void *handle;
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_Addr(&count, displs[0]);
displs[1] = 0;
types[0] = MPI_INT;
types[1] = (MPI_Datatype)MPI_Int2handle(ftype);
MPI_Type_struct(2, lens, displs, types, &newtype);
```

```
MPI_Type_commit(&newtype);

MPI_Send(MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
/* the message sent contains an int count of 5, followed  */
/* by the 5 REAL entries of the Fortran array R.          */
}
```

Rationale. The current MPI standard specifies that MPI_ADDRESS can be used in initialization expressions in C, but not in Fortran. Since Fortran does not support normally call by value, then MPI_ADDRESS must be in Fortran the name of a predefined static variable, e.g., a variable in an MPI declared COMMON block. On the other hand, in C, it is natural to take MPI_BOTTOM = 0 (Caveat: Defining MPI_BOTTOM = 0 implies that NULL pointer cannot be distinguished from MPI_BOTTOM; may be MPI_BOTTOM = 1 is better ...) Requiring that the Fortran and C values be the same will complicate the initialization process. (*End of rationale.*)

Advice to implementors. The following implementation can be used: MPI addresses, as returned by MPI_ADDRESS, will have the same value in all languages. One obvious choice is that MPI addresses be identical to regular addresses. The address is stored in the datatype, when datatypes with absolute addresses are constructed. When a send or receive operation is performed, then addresses stored in a datatype are interpreted as displacements that are all augmented by a base address. This base address is (the address of) buf, or zero, if buf = MPI_BOTTOM. Thus, if MPI_BOTTOM is zero then a send or receive call with buf = MPI_BOTTOM is implemented exactly as a call with a regular buffer argument: in both cases the base address is buf. On the other hand, if MPI_BOTTOM is not zero, then the implementation has to be slightly different. A test is performed to check whether buf = MPI_BOTTOM. If true, then the base address is zero, otherwise it is buf. In particular, if MPI_BOTTOM does not have the same value in Fortran and C/C++, then an additional test for buff = MPI_BOTTOM is needed in at least one of the languages.

It may be desirable to use a value other than zero for MPI_BOTTOM even in C/C++, so as to distinguish it from a NULL pointer. If MPI_BOTTOM = c then one can still avoid the test buff = MPI_BOTTOM, by using the displacement from MPI_BOTTOM, i.e., the regular address - c, as the MPI address returned by MPI_ADDRESS and stored in absolute datatypes. (*End of advice to implementors.*)

9.7.6 Addresses

Current Status: Passed once.

Some of the datatype accessors and constructors have arguments of type MPI_Aint (in C) or MPI::Aint in C++, to hold addresses. The corresponding arguments, in Fortran, have type INTEGER. This causes Fortran and C/C++ to be incompatible, in an environment where addresses have 64 bits, but Fortran INTEGERs have 32 bits.

This is a problem, irrespective of interlanguage issues. Suppose that a Fortran process has an address space of ≥ 4 GB. What should be the value returned in Fortran by MPI_ADDRESS, for a variable with an address above 2^{32}? The proposed design aims to solve this issue, while maintaining compatibility with current Fortran codes.

The constant MPI_ADDRESS_KIND is defined so that, in Fortran 90, INTEGER(KIND = MPI_ADDRESS_KIND) is an address sized integer type (typically, but not necessarily, MPI_ADDRESS_KIND is 4 on 32 bit address machines and 8 on 64 bit address machines). Similarly, the constant MPI_INTEGER_KIND is defined so that INTEGER(KIND = MPI_INTEGER_KIND) is a default size INTEGER.

There are seven functions that have address arguments: MPI_TYPE_HVECTOR, MPI_TYPE_HINDEXED, MPI_TYPE_STRUCT, MPI_ADDRESS, MPI_TYPE_EXTENT MPI_TYPE_LB and MPI_TYPE_UB.

Four new functions are provided to supplement the first four functions in this list. These functions are described in Section 9.8, page 197 . The remaining three functions are supplemented by the new function MPI_GET_EXTENT_X, described in that same section. The new functions are synonimous with the old functions in C/C++, or on Fortran systems where default INTEGERs are address sized. In Fortran, they accept arguments of type INTEGER(KIND = MPI_ADDRESS_KIND), wherever arguments of type MPI_Aint are used in C. On Fortran 77 systems that do not support the Fortran 90 KIND notation, and where addresses are 64 bits whereas default INTEGERs are 32 bits, these arguments will be of type INTEGER*8. The old functions will continue to be provided, for portability. However, users are encouraged to switch to the new functions, in Fortran, so as to avoid problems on systems with an address range $> 2^{32}$, and to provide compatibility across languages.

Groups

A group object represent the same group in all languages: the same processes, with the same ranks. Group accessors such as MPI_GROUP_SIZE return the same value.

Communicators

A communicator represents the same group and same communication domain in all languages. In particular, a message sent from one language using a communicator can be received from another language, using the matching communicator on another process; collective communication can involve processes executing code written in different languages, using matching communicators; accessors such as MPI_COMM_SIZE return the same value; etc.

Topology information associated to a communicator can be accessed in any of the three languages.

Windows

Current Status: No vote

A window object represents the same set of windows in all languages. In particular an RMA operation issued in one language can use a window defined in another language; a synchronization call issued in one language can match a synchronization call issued by another process in another language; etc.

Attributes

Current Status: Passed once.

Attribute keys can be allocated in one language and freed in another. Similarly, attributes values can be set in one langage 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_IO, etc.)

Attribute keys declared in one language are associated with copy and delete functions in that language (the functions provided by the MPI_Keyval_create call). When a communicator is duplicated, for each attribute, the corresponding copy function is called, using the right calling convention for the language of that function; and similarly, for the delete callback function.

> *Advice to implementors.* This requires that attributes be tagged either as "C", "C++" or "Fortran", and that the language tag be checked in order to use the right calling convention for the callback function. (*End of advice to implementors.*)

The attribute manipulation functions described in section 5.7 of the MPI1 standard define 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/C++ pointers will have 64 bits. This is a problem if communicator attributes are used to move information from a Fortran caller to a C/C++ callee, or vice-versa.

MPI will store, internally, address sized attributes. If Fortran INTEGERs are smaller, then the Fortran function MPI_ATTR_GET will return the least significant part of the attribute word; the Fortran function MPI_ATTR_PUT will set the least significant part of the attribute word, which will be sign extended to the entire word. (These two functions may be invoked explicitly by user code, or implicitly, by attribute copying callback functions.)

As for addresses, new functions are provided that manipulate in Fortran address sized attributes, and are identical to the old functions in C/C++. These functions are described in Section 9.9, page 202 . Users are encouraged to use these new functions.

Requests

A valid request created in one language can be accessed, updated, freed, or canceled in another language.

Error handlers

Error handlers can be allocated and deallocated in any combination of languages. When an MPI exception occurs, then the last error handler associated with the relevant communicator is invoked, irrespective of the language environment where the error occurred and the environment where the handler was associated with the communicator. The handler will be invoked with the right argument passing convention.

> *Advice to implementors.* Error handler objects need to have a language tag, like attribute keys, so as to use the right calling convention. These handlers have, in C and C++, a "`stdargs`" argument list. It might be useful to provide to the handler information on the language environment where the error occurred. (*End of advice to implementors.*)

9.7.7 Reduce operations

Reduce operations defined in one language (by a call to MPI_OP_CREATE) can be used or deallocated in any other language.

> *Advice to users.* Reduce operations receive, as one of their argument, the datatype of the operands. Thus, one can define "polymorphic" reduce operations that work for C, C++, and Fortran datatypes. (*End of advice to users.*)

> *Advice to implementors.* Reduce operation objects need to carry a language tag, like attribute keys, so as to use the right calling convention. (*End of advice to implementors.*)

9.7.8 Status

Current Status: Discussed

The following two functions are provided in C to convert from a Fortran status (which is an array of integers) to a C handle (which is a structure), and vice versa.

```
MPI_F2C_status( MPI_Fint *f_status, MPI_Status *cstatus)

MPI_C2F_Status( MPI_Status *cstatus, MPI_Fint *f_status)
```

If f_status is a valid Fortran status, then MPI_F2C_status returns in c_status a valid C status with the same content.

If c_status returns in f_status a valid Fortran status.

Discussion: Do we want the functions to recognize and pass MPI_STATUS_IGNORE? Do we want these two functions to manage array of statuses? Do we want an error code to be returned? It is not clear that any error checking can be done.

9.7.9 Constants

MPI constants have the same value in all languages. This includes, error codes, constants such as MPI_PROC_NULL or MPI_ANY_TAG, etc. 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 9.7.4. Also constant "addresses", i.e., special values for reference arguments that are not handles, such as MPI_BOTTOM or MPI_STATUS_IGNORED may have different values in Fortran, C, and C++.

9.7.10 Interlanguage communication

The type matching rules for communications in MPI are not changed: The datatype specification for each item sent should match, textually, the datatype specification used to receive this item (unless one of the types is MPI_PACKED). And 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 9.11 In the example below, a Fortran array is sent from Fortran and received in C.

```
! FORTRAN CODE
REAL R(5)
INTEGER TYPE, ADDR, IERR, MYRANK

! create an absolute datatype for array R
CALL MPI_ADDRESS( R, ADDR)
CALL MPI_TYPE_STRUCT(1, 5, ADDR, MPI_REAL, TYPE, IERR)
CALL MPI_TYPE_COMMIT( TYPE)

CALL MPI_COMM_RANK( MPI_COMM_WORLD, MYRANK, IERR)
IF (MYRANK.EQ.0) THEN
CALL MPI_SEND( MPI_BOTTOM, 1, TYPE, 1, 0, MPI_COMM_WORLD, IERR)
ELSE
CALL MPI_HANDLE2INT(TYPE, CODE)
CALL C_ROUTINE(CODE)
END IF


/* C code */

void C_ROUTINE(MPI_Fint *fhandle)
{
void *handle;
MPI_Datatype type;
MPI_Status status;

type = (MPI_Datatype)MPI_Int2handle(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.

9.8 New datatype manipulation functions

Current Status: Passed once.

New functions are provided to supplement the type manipulation functions that have address sized integer arguments. The new functions will use, in their Fortran binding address sized INTEGERs, thus solving problems currently encountered when the application address range is $> 2^{32}$. Also, a new, more convenient type constructor is provided to modify the lower bound and extent of a datatype.

9.8.1 Type constructors with explicit addresses

The four functions below supplement the four corresponding type constructor functions from MPI-1. The new functions are synonymous with the old functions in C/C++, or on Fortran systems where default INTEGERs are address sized. In Fortran, they accept arguments of type INTEGER(KIND = MPI_ADDRESS_SIZE), wherever arguments of type MPI_Aint are used in C. On Fortran 77 systems that do not support the Fortran 90 KIND notation, and where addresses are 64 bits whereas default INTEGERs are 32 bits, these arguments will be of type INTEGER*8. The old functions will continue to be provided, for portability. However, users are encouraged to switch to the new functions, in Fortran.

The new functions are listed below.

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

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

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

```
MPI_TYPE_HVECTOR_X(COUNT, BLOCKLENGTH, STIDE, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_SIZE) STRIDE
```

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

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

```
int MPI_Type_hindexed_x(int count, int *array_of_blocklengths,
             MPI_Aint *array_of_displacements, MPI_Datatype oldtype,
```

```
              MPI_Datatype *newtype)
```

```
MPI_TYPE_HINDEXED_X(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
              OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), OLDTYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_SIZE) ARRAY_OF_DISPLACEMENTS(*)
```

MPI_TYPE_STRUCT_X(count, array_of_blocklengths, array_of_displacements, array_of_types, newtype)

IN	count	number of blocks (integer) – also number of entries in arrays array_of_types, array_of_displacements and array_of_blocklengths
IN	array_of_blocklength	number of elements in each block (array of integer)
IN	array_of_displacements	byte displacement of each block (array of integer)
IN	array_of_types	type of elements in each block (array of handles to datatype objects)
OUT	newtype	new datatype (handle)

```
int MPI_Type_struct_x(int count, int *array_of_blocklengths,
              MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types,
              MPI_Datatype *newtype)
```

```
MPI_TYPE_STRUCT_X(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
              ARRAY_OF_TYPES, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_TYPES(*), NEWTYPE,
    IERROR
    INTEGER(KIND=MPI_ADDRESS_SIZE) ARRAY_OF_DISPLACEMENTS(*),
```

MPI_ADDRESS_X(location, address)

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

```
int MPI_Address_x(void* location, MPI_Aint *address)
```

```
MPI_ADDRESS_X(LOCATION, ADDRESS, IERROR)
    <type> LOCATION(*)
    INTEGER IERROR
    INTEGER(KIND=MPI_ADDRESS_SIZE) ADDRESS
```

Advice to users. Current Fortran MPI codes will run unmodified, and will port to any system. However, they may fail if addresses larger than $2^{32} - 1$ are used in the program. New codes should be written so that they use the new functions. This provides compatibility with C/C++ and avoids errors on 64 bit architectures. However, such newly written codes may need to be (slightly) rewritten to port to old

Fortran 77 environments that do not support KIND declarations. (*End of advice to users.*)

9.8.2 Extent and Bounds of Datatypes

The following function supplements the three functions MPI_TYPE_UB, MPI_TYPE_LB and MPI_TYPE_EXTENT. It also returns address sized integers, in the Fortran binding.

MPI_GET_EXTENT_X(datatype, lb, extent)

IN	datatype	datatype to get information on
OUT	lb	lower bound of datatype
OUT	extent	extent of datatype

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

```
MPI_GET_EXTENT_X(DATATYPE, LB, EXTENT,IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND = MPI_ADDRESS_SIZE) LB, EXTENT
```

Returns the lower bound and the extent of datatype (as defined by the MPI standard, Section 3.12.2).

MPI allows one to change the extent of a datatype, using lower bound and upper bound markers (MPI_LB and MPI_UB). This is useful, as it allows to control the stride of successive datatypes that are replicated by datatype constructors, or are replicated by the count argument in a send or recieve call. However, the current mechanism for achieving it is painful; also it is restrictive, as MPI_LB and MPI_UB are "sticky": once present in a datatype, they cannot be overiden (e.g., the upper bound can be moved up, by adding a new MPI_UB marker, but cannot be moved down below an existing MPI_UB marker). A new type constructor is provided to facilitate these changes.

MPI_TYPE_RESIZED_X(oldtype, lb, extent, newtype)

IN	oldtype	input datatype (handle)
IN	lb	new lower bound of datatype (integer)
IN	extent	new extent of datatype (integer)
OUT	newtype	output datatype (handle)

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

```
MPI_TYPE_RESIZED_X(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)
    INTEGER DATATYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_SIZE) 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. This is as if any previous **lb** and **ub** markers were erased, and a new pair of lower bound and upper bound markers were 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.

> *Advice to users.* It is strongly recommended that users use these two new functions, rather than the old MPI functions to set and access lower bound, upper bound and extent of datatypes. (*End of advice to users.*)

9.8.3 True Extent of Datatypes

Suppose we implement gather as a spanning tree implemented on top of point-to-point routines. Since the recv buffer is only valid on the root process, one will need to allocate some temporary space for receiving data on intermediate nodes. The difficultly is in determining the size one needs to allocate. This occurs since the user can modify the extent using the MPI_UB and MPI_LB values. The writer of the gather routine could determine this information by decoding the datatype. However, this is more work and more painful than desired. Thus, a new function is provided which returns the true extent of the datatype.

MPI_TRUE_EXTENT_X(datatype, true_lb, true_extent)

IN	datatype	datatype to get information on
OUT	true_lb	true lower bound of datatype
OUT	true_extent	true size of datatype

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

```
MPI_TRUE_EXTENT_X(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
    INTEGER DATATYPE, IERROR
    INTGER(KIND = MPI_ADDRESS_SIZE) 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 MPI_LB markers. true_extent returns the true size of the datatype, i.e., the extent of the corresponding typemap, ignoring MPI_LB and MPI_UB markers, and performing no rounding for alignment. If the typemap associated with datatype is

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

Then

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

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

and

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

(Compares this with the definitions in Section 3.12.3 of the MPI standard.)

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

9.9 New attribute caching functions

Current Status: Discussed

New functions are provided to supplement the attribute caching functions. The new functions will use, in their Fortran binding address sized attributes, thus solving C to Fortran compatibility problems currently encountered when C pointers and Fortran integers do not have the same size.

MPI_KEYVAL_CREATE_X(copy_fn, delete_fn, keyval, extra_state)

IN	copy_fn	Copy callback function for keyval
IN	delete_fn	Delete callback function for keyval
OUT	keyval	Key value for future access
IN	extra_state	Extra state for callback functions

```
int MPI_Keyval_create_x(MPI_Copy_function *copy_fn, MPI_delete_function
             *delete_fn, int *keyval, void *extra_state
```

```
MPI_KEYVAL_CREATE_X(COPY_FN, DELETE_FN, KEYVAL, EXTRA STATE, IERROR)
    EXTERNAL COPY_FN, DELETE_FN
    INTEGER KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

This function replaces MPI_KEYVAL_CREATE. The C binding is identical. The Fortran binding differs in that extra_state is an address-sized integer. Also, the copy and delete callback functions have Fortran bindings that are consistent with address-sized attributes. The Fortran delcarations for the functions are

```
SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE,
        ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR)
INTEGET OLDCOMM, KEYVAL, IERR
INTEGER(KING=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
        ATTRIBUTE_VAL_OUT
LOGICAL FLAG
```

and

```
SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE,
        IERR)
INTEGER COMM, KEYVAL, IERR
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

MPI_ATTR_PUT_X(comm, keyval, attribute_val)

INOUT	comm	communicator to which attribure will be attached
IN	keyval	key value
IN	attribute_val	attribute value

```
int MPI_Attr_put_x(MPI_Comm comm, int keyval, void* attribute_val)
```

```
MPI_ATTR_PUT_X(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER COMM, KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
```

This function is replaces MPI_KEYVAL_PUT. The C binding is identical. The Fortran binding differs in that attribute_val is an address-sized integer.

MPI_ATTR_GET_X(comm, keyval, attribute_val, flag)

INOUT	comm	communicator to which the attribute is attached
IN	keyval	key value
OUT	attribute_val	attribute value, unless flag = false
OUT	flag	false if no attribute is associated with the key

```
int MPI_Attr_get_x(MPI_Comm comm, int keyval, void *attribute_val)
```

```
MPI_ATTR_GET_X(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER COMM, KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
    LOGICAL FLAG
```

This function replaces MPI_ATTR_GET. the C binding is identical. The Fortran binding differs in that attribute_val is an address-sized integer.

9.10 Continuable Errors

Current Status: No votes.

Typically, MPI says nothing about the internal state of itself after an error handler has been invoked. There are a number of error classes within MPI that should have no negative impact on the internal state of MPI. That is to say, if such a "continuable" errors is detected and no other external events cause degradation, it is safe to continue using MPI within an application. Since whether an error is continuable or not may be dependent on the context, instead of a list of continuable errors there is a function to determine whether an error is (at the moment) continuable.

We provide the function to indicate whether MPI believes an error is "continuable" or not. The answer may depend on run-time context.

MPI_ERR_IS_CONTINUABLE (error, flag)

IN	Error	MPI Error Status (actual status or a class)
OUT	Flag	Boolean indication of whether the implementation allows safe continued operation in the event of Error being encountered

```
int MPI_Err_is_continuable( int error, int *flag )
```

```
MPI_ERR_IS_CONTINUABLE( ERROR, FLAG, IERROR )
    INTEGER ERROR, IERROR
    LOGICAL FLAG
```

There is a new, continuable error class, MPI_ERR_WOULDBLOCK. This error class may be used by a non-blocking function to inform the caller that the operation cannot proceed without blocking. When this error is returned, the state of the application is equivalent to the one prior to the function call.

An implementation is not required to use this error class, but if it does, it must provide the functionality described.

Discussion: What does it mean when a continuable error occurs in a collective operation?

There is a predefined error handler MPI_CONT_ERRORS_RETURN, which allows continuable errors to return error codes and non-continuable errors to abort. The default error handler is MPI_ERRORS_ARE_FATAL.

Discussion: Perhaps an implementation should provide an error handler that aborts for non-continuable errors and calls a user-defined error handler for continualbe errors.

Example 9.12 *This example shows the use of continuable errors.*

```
MPI_Errhandler_set(MPI_COMM_WORLD, MPI_CONT_ERRORS_RETURN);
err = MPI_Isend( ... , &req);
if ( err != MPI_SUCCESS) {
    MPI_Error_class(err,&class);
    if (class == MPI_ERR_WOULDBLOCK) {
        MPI_Send( ... );
    else
        ...
    }
}
```

9.11 New Elementary Datatypes

9.11.1 Wide Characters

A new datatype, MPI_WCHAR, is added, for the purpose of dealing with international character sets such as Unicode.

MPI_WCHAR is a 'C' type, which corresponds to the type `wchar_t` defined in `<stddef.h>`. There are no predefined reduction operations for MPI_WCHAR.

> *Rationale.* The fact that MPI_CHAR is associated with the C datatype `char`, which in turn is often used as a substitute for the "missing" `byte` datatype in C makes it most natural to define this as a new datatype specifically for multi-byte characters. (*End of rationale.*)

9.11.2 Signed Characters and Reductions

MPI-1 doesn't allow reductions on signed or unsigned `char`s. Since this (formally) prevents a C programmer from performing reduction operations on such types (which could be useful, particularly in an image processing application where pixel values are often represented as "unsigned char"), it seems that we should now fix this omission.

MPI-1.2 already has the C types MPI_CHAR and MPI_UNSIGNED_CHAR. However there is a problem here in that MPI_CHAR is intended to represent a character, not a small integer, and therefore will be translated between machines with different character representations.

To overcome this, a new MPI elementary datatype MPI_SIGNED_CHAR is addeded to the list of elementary datatypes in Appendix A of MPI-2, which corresponds to the ANSI C datatype `signed char`.

> *Advice to users.*
>
> The types MPI_CHAR and MPI_CHARACTER are intended for characters, and so will be translated to preserve the printable representation, rather than the bit 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.*)

The types MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR can be used in reduction operations. MPI_CHAR (which represents printable characters) cannot be used in reduction operations. This is an extension to MPI-1.2, since MPI-1.2 does not allow the use of MPI_UNSIGNED_CHAR in reduction operations (and does not have the MPI_SIGNED_CHAR type).

In a heterogeneous environment, MPI_CHAR and MPI_WCHAR will be translated so as to preserve the printable charater, whereas MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR will be translated so as to preserve the integer value.

Chapter 10

I/O

Current Status: No votes on entire chapter

10.1 Introduction

Different parallel file systems support different interfaces and the lack of a standard portable interface makes it impossible to develop a truly portable parallel program. It is not feasible to develop large scientific applications from scratch for each generation of parallel machine. In the scientific world, portability and performance are inseparable—a program is not considered truly portable unless it not only compiles, but also runs efficiently. UNIX provides a model of a widely portable file system, but the portability and optimization needed for parallel I/O cannot be achieved with traditional UNIX.

The significant optimizations required for efficiency (e.g., grouping [14], collective buffering [1, 2, 15, 18, 20], and disk-directed I/O [12]) can only be implemented as part of a parallel I/O environment if it supports a high-level interface to describe the partitioning of file data among processes and a collective interface describing 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 requests, and control over physical file layout on storage devices (disks). The I/O environment described in this section provides these facilities.

Instead of defining file access modes to express the common patterns for accessing a shared file (broadcast, reduction, scatter, gather), we chose another approach which consists of expressing the data partitioning via derived datatypes. Compared to a limited set of pre-defined access patterns, this approach has the advantage of added flexibility and expressiveness.

10.1.1 Definitions

file An MPI file is a persistent, ordered collection of data bytes supporting random access to any integral set of bytes. 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 *displacement* is an absolute byte position relative to the beginning of a file. A displacement is an offset relative to the default view (see below).

etype An *etype* (*elementary* datatype) is the unit of data access and positioning, and can be any MPI basic or derived datatype. Data access is performed in etype units, reading or writing whole datums 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 type, a datum of that type, or the extent of that type.

filetype A *filetype* describes data distribution, and is defined as an MPI datatype built from etypes and etype sized holes. The distribution is a mapping of the ordering of etypes in the file to a new ordering. This mapping can be arbitrarily complex. For example, relative to the file ordering, the new ordering: might not contain all etypes, might contain the etypes in a different order, and might contain multiple copies of the same etype.

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 (see Figure 10.1). Views can be changed by the user during execution. The default view is a linear byte stream (displacement is zero, etype and filetype equal to MPI_BYTE).

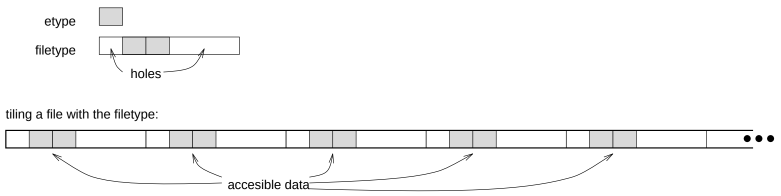

Figure 10.1: Etypes and filetypes

A group of processes can use complementary views to achieve a global data distribution such as a scatter/gather pattern (see Figure 10.2).

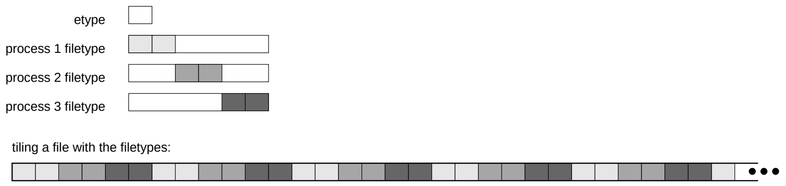

Figure 10.2: Partitioning a file among parallel processes

offset An *offset* is a position relative to the current view, expressed as a count of etypes. Holes in the view's filetype are ignored when calculating this position. For example, an offset of 2 for process 1 in Figure 10.2 denotes the 7th etype in the file. An "explicit offset" is an offset that is used as a formal parameter in explicit data access routines.

file pointer A *file pointer* is an implicit offset maintained by the system. "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.

10.2 File Manipulation

10.2.1 Opening a File

MPI_OPEN(comm, filename, amode, info, fh)

IN	comm	communicator (handle)
IN	filename	name of file to open (string)
IN	amode	file access mode (integer)
IN	info	info object (handle)
OUT	fh	new file handle (handle)

```
int MPI_Open(MPI_Comm comm, char *filename, int amode, MPI_Info info,
             MPI_File *fh)
```

```
MPI_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR)
    CHARACTER FILENAME(*)
    INTEGER COMM, AMODE
    INTEGER INFO, FH, IERROR
```

MPI_OPEN opens the file identified by the file name filename on all processes in the comm communicator group. MPI_OPEN is a collective routine; all processes in the communicator group must call MPI_OPEN, with identical values for comm, filename, and amode. (Values for info may vary.) A process can open a file independently of other processes by using the MPI_COMM_SELF communicator. The file handle returned, fh, can be subsequently used to access the file until the file is closed using MPI_CLOSE or MPI_ICLOSE. Note that the communicator comm is unaffected by MPI_OPEN, and continues to be usable in all MPI routines (e.g., MPI_SEND). Further, use of comm will not interfere with I/O behavior.

Initially, all processes view the file as a linear byte stream. (POSIX files are linear byte streams.) The file view can be changed via the MPI_FILE_SET_VIEW routine.

The following access modes, specified via amode, are supported:

- MPI_RDONLY - read only,
- MPI_RDWR - reading and writing,
- MPI_WRONLY - write only,
- MPI_CREATE - create the file if it does not exist,
- MPI_EXCL - error if creating file that already exists,
- MPI_DELETE_ON_CLOSE - delete file on close,
- MPI_DATA_NATIVE - file data is in native format (see section 10.5),
- MPI_DATA_INTERNAL - file data is in an implementation dependent format which ensures interoperability within this MPI environment (see section 10.5),
- MPI_DATA_CANONICAL - file data is in canonical format (see section 10.5).

The modes MPI_RDONLY, MPI_RDWR, MPI_WRONLY, MPI_CREATE, and MPI_EXCL have identical semantics to their POSIX counterparts. These can be combined in C by using the bitwise OR operator, and in FORTRAN by adding them.

> *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.*)

> *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.*)

The info argument is used to provide information regarding file access patterns and file system specifics (e.g. optimization, see section 10.2.9). The constant MPI_INFO_NULL refers to the null info, and can be used when no info needs to be specified.

The format for specifying the file name in the filename parameter is implementation dependent.

> *Advice to implementors.* An implementation may require the user to prefix the actual filename with a string that indicates the type of file system (e.g., ufs:), or even a remote hostname if the implementation supports remote files (e.g., machine.univ.edu:ufs:). (*End of advice to implementors.*)

Files are opened by default using non-atomic mode file consistency semantics (see section 10.6.1). The more stringent atomic mode consistency semantics, required for atomicity of overlapping accesses, can be set using MPI_FILE_SET_ATOMICITY.

10.2.2 Nonblocking Open

MPI_IOPEN(comm, filename, amode, info, fh, request)

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)
OUT	request	request object (handle)

```
int MPI_Iopen(MPI_Comm comm, char *filename, int amode, MPI_Info info,
              MPI_File *fh, MPI_Request *request)
```

```
MPI_IOPEN(COMM, FILENAME, AMODE, INFO, FH, REQUEST, IERROR)
    CHARACTER FILENAME(*)
    INTEGER COMM, AMODE, INFO, FH, REQUEST, IERROR
```

MPI_IOPEN is a nonblocking version of the MPI_OPEN interface. MPI_IOPEN is a collective routine. MPI_IOPEN associates a request handle request with the open request.

The request handle can be used later to query the status of the open (via the MPI_TEST routines), or wait for its completion (via the MPI_WAIT routines).

Once the open completes, the returned file handle fh can be used in all I/O routines (as in MPI_OPEN). It is erroneous to use fh before the open completes.

As with MPI_OPEN, the communicator comm can be used in all MPI routines without affecting I/O behavior. In particular, comm can be used to send and receive messages without waiting for the open to complete.

10.2.3 Closing a file

MPI_CLOSE(fh)

INOUT	fh	file handle (handle)

```
int MPI_Close(MPI_File *fh)
```

```
MPI_CLOSE(FH, IERROR)
    INTEGER FH, IERROR
```

MPI_CLOSE first synchronizes file state (equivalent to performing an MPI_FILE_SYNC), then closes the file associated with fh. The file is deleted if it was opened with access moded MPI_DELETE_ON_CLOSE (equivalent to performing an MPI_DELETE). MPI_CLOSE is a collective routine.

> *Advice to users.* If the file is deleted on close, and there are other processes currently accessing the file, the status of the file and the behavior of future accesses by these processes are implementation dependent. (*End of advice to users.*)

The user is responsible for ensuring that all outstanding requests associated with fh have completed before calling MPI_CLOSE.

The MPI_CLOSE routine sets fh to MPI_FILE_NULL.

10.2.4 Nonblocking Close

MPI_ICLOSE(fh, request)

INOUT	fh	file handle (handle)
OUT	request	request object (handle)

```
int MPI_Iclose(MPI_File *fh, MPI_Request *request)
```

```
MPI_ICLOSE(FH, REQUEST, IERROR)
    INTEGER FH, REQUEST, IERROR
```

MPI_ICLOSE is a nonblocking version of the MPI_CLOSE interface. MPI_ICLOSE is a collective routine. Semantics for outstanding requests, MPI_DELETE_ON_CLOSE, and the setting of fh to MPI_FILE_NULL act as if MPI_CLOSE was called.

MPI_ICLOSE associates a request handle request with the close request. The request handle can be used later to query the status of the close (via the MPI_TEST routines), or wait for its completion (via the MPI_WAIT routines).

10.2.5 Deleting a file

MPI_DELETE(filename)

IN	filename	name of file to delete (string)

```
int MPI_Delete(char *filename)
```

```
MPI_DELETE(FILENAME, IERROR)
    CHARACTER FILENAME(*)
    INTEGER IERROR
```

MPI_DELETE deletes the file identified by the file name filename, provided it is not currently open by any process. If the file does not exist, MPI_DELETE returns an error in the class MPI_ERR_NO_SUCH_FILE.

If a process currently has the file open (via the file handle fh), the behavior of any access to fh (as well as the behavior of any outstanding accesses) is implementation dependent. In addition, whether an open file is removed or not is also implementation dependent. If the file is not removed because it is currently open, an error in the class MPI_ERR_FILE_NOT_DELETED is returned.

10.2.6 Resizing a file

MPI_RESIZE(fh, size)

IN	fh	file handle (handle)
IN	size	size to truncate or expand file (integer)

```
int MPI_Resize(MPI_File fh, MPI_Offset size)
```

```
MPI_RESIZE(FH, SIZE, IERROR)
    INTEGER FH, IERROR
    INTEGER*8 SIZE
```

MPI_RESIZE resizes the file associated with the file handle fh truncating or preallocating as necessary. MPI_RESIZE 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 (from the beginning of the file and measured in bytes). File blocks located beyond that position are deallocated. If size is larger than the current file size, additional file blocks are allocated and the file size becomes size. File preallocation behaves as if the newly allocated region in the file has been written using MPI_WRITE_EXPLICIT with undefined data. MPI_RESIZE does not affect the individual file pointers or the shared file pointer.

Advice to users. It is possible for the file pointers to point beyond the end of file after a MPI_RESIZE operation truncates a file. This is legal, and equivalent to seeking beyond the current end of file. (*End of advice to users.*)

For pending nonblocking read and write requests that have not been completed (e.g., via MPI_WAIT), the value of each byte of data accessed will be equivalent to either the value which would have been accessed before the MPI_RESIZE was called, or the value which would have been accessed after the MPI_RESIZE completes. The exact behavior (which of these two choices) of each byte of each outstanding operation is implementation dependent. For example, an outstanding nonblocking read of a truncated region in a file might return the data (that existed before truncation), or an MPI_ERR_EOF error, or, possibly, both.

Advice to users. The user is responsible for ensuring that all outstanding nonblocking requests of interest are completed before calling MPI_RESIZE. (*End of advice to users.*)

As far as consistency semantics are concerned, MPI_RESIZE is a data access operation (see Section 10.6.1).

10.2.7 Querying the size of a file

MPI_FILE_GET_SIZE(fh, size)

IN	fh	file handle (handle)
OUT	size	size of the file in bytes (integer)

```
int MPI_File_get_size(MPI_File fh, MPI_Offset *size)
```

```
MPI_FILE_GET_SIZE(FH, SIZE, IERROR)
    INTEGER FH, IERROR
    INTEGER*8 SIZE
```

MPI_FILE_GET_SIZE returns, in size, the current size in bytes of the file associated with the file handle fh.

10.2.8 Querying File Parameters

MPI_FILE_GET_COMM(fh, comm)

IN	fh	file handle (handle)
OUT	comm	copy of comm used to open the file (handle)

```
int MPI_File_get_comm(MPI_File fh, MPI_Comm *comm)
```

```
MPI_FILE_GET_COMM(FH, COMM, IERROR)
    INTEGER FH, COMM, IERROR
```

MPI_FILE_GET_COMM returns the communicator associated with the open file fh. MPI_FILE_GET_COMM is a collective routine. A duplicate of the communicator used to open the file is returned in comm. The user is responsible for freeing comm.

MPI_FILE_GET_AMODE(fh, amode)

IN	fh	file handle (handle)
OUT	amode	file access mode used to open the file (integer)

```
int MPI_File_get_amode(MPI_File fh, int *amode)
```

```
MPI_FILE_GET_AMODE(FH, AMODE, IERROR)
    INTEGER FH, AMODE, IERROR
```

MPI_FILE_GET_AMODE returns, in amode, the mode associated with the open file fh.

10.2.9 File Info

Hints specified via info (see section 4.5.1) allow a user to provide information regarding file access patterns and file system specifics to direct optimization. Providing hints may enable an implementation to deliver increased I/O performance or minimize the use of system resources. However, hints do not change the semantics of any of the interfaces. In other words, an implementation is free to ignore all hints. Hints are specified on a per file basis at MPI_OPEN, MPI_IOPEN, or MPI_FILE_SET_VIEW time via the opaque info object.

> *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. In particular, setting or inquiring values to unsupported hints, should be handled as if these hints were not specified. Needless to say, no hint can be mandatory. However, for each hint used by a specific implementation, a default value must be provided in order to be used when the user does not specify a value for this hint. (*End of advice to implementors.*)

MPI_FILE_SET_INFO(fh, info)

IN	fh	file handle (handle)
IN	info	info object (handle)

```
int MPI_File_set_info(MPI_File fh, MPI_Info info)
```

```
MPI_FILE_SET_INFO(FH, INFO, IERROR)
    INTEGER FH, INFO, IERROR
```

MPI_FILE_SET_INFO sets new values for the hints associated with the open file fh. MPI_FILE_SET_INFO is a collective routine. The info object may be different on each node, but any info entries that an implementation requires to be the same on all nodes must appear in each node'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 after the file is open. Thus, an implementation may ignore hints issued in this call that it would have accepted in an open call. (*End of advice to users.*)

MPI_FILE_GET_INFO(fh, info_used)

IN	fh	file handle (handle)
OUT	info_used	new info object (handle)

```
int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)
```

```
MPI_FILE_GET_INFO(FH, INFO_USED, IERROR)
    INTEGER FH, INFO_USED, IERROR
```

MPI_FILE_GET_INFO returns a new info object containing the hints associated with the open file fh. The current setting of all hints actually used by the system related to this open file is returned in info_used. The user is responsible for freeing info_used via MPI_INFO_FREE.

Advice to users. The info object returned in info_used will contain all hints currently active for this file. This set of hints may be greater or smaller than the set of hints passed in to MPI_OPEN, MPI_IOPEN, MPI_FILE_SET_VIEW, and MPI_FILE_SET_INFO, as the system may not recognize some hints set by the user, and may recognize other hints that the user has not set. (*End of advice to users.*)

Reserved File Hints

Some potentially useful hints are outlined below. Specific implementations are free to define these hints as well as additional hints. However, in order to maintain compatibility between implementations, additional hints should not reuse the same hint names for other purposes than these listed here.

These hints are mainly concerned with layout of data on parallel I/O devices, and with access patterns. For each hint name introduced, we describe the purpose of the hint, and the type of the hint value. If specified, the "**[SAME]**" annotation stipulates 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 **(list of comma separated strings):** This hint specifies the manner in which the file will be accessed until the file is closed. 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 will benefit from collective buffering (an optimization performed on collective accesses which coalesces small requests into large disk accesses). Legal values for this key are true and false. Collective buffering parameters are further directed via additional hints.

cb_block_size **(integer) [SAME]:** This hint specifies the block size used for collective buffering file access. Target nodes access data in chunks of this size. The chunks are distributed to target nodes in a round-robin (CYCLIC) pattern.

cb_buffer_size **(integer) [SAME]:** This hint specifies the total buffer space 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 used for collective buffering. All file accesses are performed by the target nodes.

chunked **(list of comma separated integers):** This hint specifies whether the application will benefit from a chunked array layout on disk (an optimization supporting large, multidimensional, out-of-core applications). The value for this hint is a comma separated list of array dimensions denoting the size of the global array stored in the file.

chunked_primary **(list of comma separated strings):** This hint specifies the primary storage order for chunked array layout, i.e., how the data should be stored within a chunk. Each string is one of C_order (i.e., row-major order) or FORTRAN_order (i.e., column-major order).

chunked_secondary **(list of comma separated strings):** This hint specifies the secondary storage order for chunked array layout, i.e., how the chunks should be ordered on disk. Each string is one of row_major (i.e., C order) or column_major (i.e., FORTRAN order).

chunked_size **(list of comma separated integers):** This hint specifies the size of the average chunk accessed (extent in each dimension).

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_OPEN, MPI_IOPEN, MPI_FILE_SET_VIEW, and MPI_FILE_SET_INFO.

file_perm **(string) [SAME]:** This hint specifies the file permissions to use for file creation. This hint is only useful when passed to MPI_OPEN or MPI_IOPEN when amode includes MPI_CREATE. The value with this key is implementation dependent.

io_node_list **(list of comma separated strings) [SAME]:** This hint specifies the list of I/O devices that should be used to store a file. This hint is relevant only when the file is created.

nb_proc **(integer) [SAME]:** This hint specifies the number of parallel processes that will typically be assigned to run programs that access this file. This hint is most relevant when the file is created.

num_io_nodes **(integer) [SAME]:** This hint specifies the number of I/O devices in the system. Used to specify the ideal number of I/O devices for this application.

striping_factor **(integer) [SAME]:** This hint specifies the number of I/O devices that the file should be striped across, and is relevant only when the file is created.

striping_unit **(integer) [SAME]:** This hint specifies the suggested striping unit to be used for this file. The striping unit is the amount of consecutive data taken from one I/O device before progressing to the next device, when striping across a number of devices; it is expressed in bytes. This hint is relevant only when the file is created.

10.3 File Views

MPI_FILE_SET_VIEW(fh, disp, etype, filetype, info)

IN	fh	file handle (handle)
IN	disp	displacement (integer)
IN	etype	elementary datatype (handle)
IN	filetype	filetype (handle)
IN	info	info object (handle)

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
             MPI_Datatype filetype, MPI_Info info)
```

```
MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, INFO, IERROR)
    INTEGER FH, ETYPE, FILETYPE, INFO, IERROR
    INTEGER*8 DISP
```

The MPI_FILE_SET_VIEW routine changes the process's view of the data in the file—the beginning of the data accessible in the file through that view is set to disp; the type of data is set to etype; and the distribution of data to processes is set to filetype. In addition, MPI_FILE_SET_VIEW resets the independent file pointers and the shared file pointer to zero. MPI_FILE_SET_VIEW is collective; all processes in the group must pass identical values for fh and etype; values for disp, filetype, and info may vary. It is erroneous to use the shared file pointer data access routines unless identical values for disp and filetype are also given (see section 10.4.4). The datatypes passed in etype and filetype must be committed.

Advice to implementors. It is expected that a call to MPI_FILE_SET_VIEW will immediately follow the MPI_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 10.3). Separate views, each defined using a displacement and a filetype, can be used to access each segment.

(*End of advice to users.*)

The etype argument specifies the elementary datatype used to construct the filetype, and also the datatype used in read/write operations. Offsets into the view are measured in units of etype.

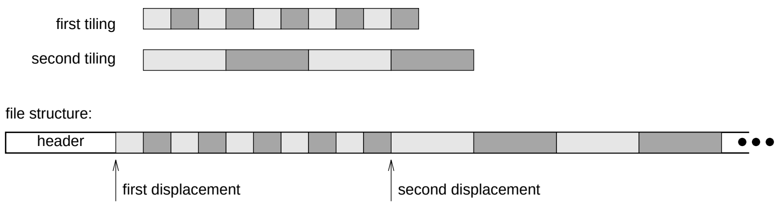

Figure 10.3: Displacements

The filetype argument describes what part of the data in the file is being accessed. Conceptually, the file starting from disp is tiled by repeated copies of filetype, until the end. If filetype has holes in it, then the data in the holes is inaccessible by 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.

The info argument is used to provide information regarding file access patterns and file system specifics to direct optimization (see section 10.2.9). The constant MPI_INFO_NULL refers to the null info, and can be used when no info needs to be specified.

The MPI_FILE_SET_VIEW routine does not affect outstanding data access operations. Every data access operation on a file handle (fh), blocking or nonblocking, is performed using the file view of fh in effect when the operation was initiated.

MPI_FILE_GET_VIEW(fh, disp, etype, filetype)

IN	fh	file handle (handle)
OUT	disp	displacement (integer)
OUT	etype	elementary datatype (handle)
OUT	filetype	filetype (handle)

```
int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype,
             MPI_Datatype *filetype)
```

```
MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, IERROR)
    INTEGER FH, ETYPE, FILETYPE, IERROR
    INTEGER*8 DISP
```

The MPI_FILE_GET_VIEW returns the process's view of the data in the file. The current values of the displacement, etype, and filetype are returned in disp, etype, and filetype respectively.

The user is responsible for freeing derived datatypes returned in etype and filetype via MPI_TYPE_FREE.

> *Advice to users.* The datatypes returned by MPI_FILE_GET_VIEW are plain MPI datatypes, and can be used for I/O and communication. However, it is implementation dependent whether the returned types are committed. For this reason, if either etype or filetype are to be used in communication, the user is responsible for committing them. (*End of advice to users.*)

10.4 Data Access

10.4.1 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 coordination (non-collective vs. collective). All combinations of these data access routines, including two types of file pointers, individual and shared, are provided.

The data access routines are summarized in the following table:

positioning	**synchronism**	**coordination**	
		non-collective	*collective*
explicit offsets	*blocking* (synchronous)	MPI_Read_explicit MPI_Write_explicit	MPI_Read_explicit_all MPI_Write_explicit_all
	nonblocking (asynchronous)	MPI_Iread_explicit MPI_Iwrite_explicit	MPI_Iread_explicit_all MPI_Iwrite_explicit_all
individual file pointers	*blocking* (synchronous)	MPI_Read MPI_Write	MPI_Read_all MPI_Write_all
	nonblocking (asynchronous)	MPI_Iread MPI_Iwrite	MPI_Iread_all MPI_Iwrite_all
shared file pointer	*blocking* (synchronous)	MPI_Read_shared MPI_Write_shared	MPI_Read_shared_ordered MPI_Write_shared_ordered
	nonblocking (asynchronous)	MPI_Iread_shared MPI_Iwrite_shared	MPI_Iread_shared_ordered MPI_Iwrite_shared_ordered

POSIX read() and write() are blocking, non-collective operations, and use individual file pointers. The MPI equivalents are MPI_READ and MPI_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. However, the MPI_FILE_SYNC routine provides the only guarantee that data has been written to "permanent" storage.

Positioning

MPI provides three types of positioning for data access routines: explicit offsets, individual file pointers, and shared file pointers. The different positioning methods may be mixed within the same program, and do not affect each other.

The data access routines which accept explicit offsets contain _EXPLICIT in their name (e.g., MPI_XXX_EXPLICIT). Explicit offset operations act as atomic seek-and-read or seek-and-write operations. Operations with explicit offsets are described in Section 10.4.2. The names of the individual file pointer routines contain no positional qualifier (e.g., MPI_XXX). Operations with individual file pointers are described in Section 10.4.3. The data access routines which use shared file pointers contain _SHARED in their name (e.g., MPI_XXX_SHARED). Operations with shared file pointers are described in Section 10.4.4.

The main semantic issues with system-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 was accessed.

More formally,

$$new\_file\_position = old\_position + \frac{size(datatype) \times count}{size(etype)}$$

where *count* is the number of elements of type *datatype* to be accessed and where $size(X)$ gives the number of bytes of actual data (excluding holes) that composes the MPI datatype X.

For both performance and thread safety reasons, MPI always updates the file pointer at the outset of an operation by the amount of data requested. Waiting for an access to complete before updating the file pointer would restrict the ability to initiate additional accesses by the same process (for both types of file pointers) and by other processes (for the shared file pointer).

Updating the file pointer at the outset of an operation yields identical semantics to UNIX except in one rare case: reading beyond the end of file and subsequently writing. In all cases, however, it is more efficient and provides better support for threads. Consider the following scenario:

```
MPI_Iread(fh, buf, count, datatype, &status);
MPI_Iwrite(fh, buf, count, datatype, &status);
```

If the first read reaches the end of the file before completing, the file pointer will be incremented by the amount of data *requested* rather than the amount of data read. Therefore, the file pointer will point beyond the current end of the file, and the write will leave a hole in the file.

Synchronism

The standard supports blocking and nonblocking I/O routines.

A *blocking* I/O call will block until the I/O request is completed.

A *nonblocking* I/O call initiates an I/O operation, but does not wait for it to complete. Given suitable hardware, this allows the transfer of data out/in the user's buffer to proceed concurrently with computation. A separate *request complete* call (MPI_WAIT, MPI_TEST, or any of their variants) is needed to complete the I/O request, i.e., to certify that data has been read/written, and it is safe for the user to reuse the buffer. The nonblocking versions of the routines are named MPI_IXXX, where the I stands for immediate.

Once a nonblocking data access operation is initiated, accessing the local buffer of the operation, or using it as the source or target of other communications, before the operation completes is erroneous.

Coordination

Every non-collective data access routine MPI_XXX has a collective counterpart, MPI_XXX_ALL or MPI_XXX_ORDERED. The only difference between collective operations and their non-collective counterparts is potential synchronization. The completion of a non-collective 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.

Collective operations may perform much better than their non-collective counterparts, as global data accesses have significant potential for automatic optimization.

10.4.2 Data Access With Explicit Offsets

MPI_READ_EXPLICIT(fh, offset, buf, count, datatype, status)

IN	fh	file handle (handle)
IN	offset	file offset (integer)
OUT	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	status	status object (Status)

```
int MPI_Read_explicit(MPI_File fh, MPI_Offset offset, void *buf, int count,
              MPI_Datatype datatype, MPI_Status *status)
```

```
MPI_READ_EXPLICIT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER*8 OFFSET
```

MPI_READ_EXPLICIT attempts to read from the file associated with fh (at the offset position) a total number of count data items having datatype type into the user's buffer buf. The offset is in etype units relative to the current view. The data is taken out of those parts of the file specified by the current view. MPI_READ_EXPLICIT stores the number of datatype elements actually read in status. All other fields of status are undefined.

MPI_READ_EXPLICIT_ALL(fh, offset, buf, count, datatype, status)

```
int MPI_Read_explicit_all(MPI_File fh, MPI_Offset offset, void *buf,
              int count, MPI_Datatype datatype, MPI_Status *status)
```

```
MPI_READ_EXPLICIT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER*8 OFFSET
```

MPI_READ_EXPLICIT_ALL is a collective version of the blocking MPI_READ_EXPLICIT interface. This routine is semantically identical to MPI_READ_EXPLICIT, except that this routine is collective.

MPI_WRITE_EXPLICIT(fh, offset, buf, count, datatype, status)

IN	fh	file handle (handle)
IN	offset	file offset (integer)
IN	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	status	status object (Status)

```
int MPI_Write_explicit(MPI_File fh, MPI_Offset offset, void *buf, int count,
              MPI_Datatype datatype, MPI_Status *status)
```

```
MPI_WRITE_EXPLICIT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER*8 OFFSET
```

MPI_WRITE_EXPLICIT attempts to write into the file associated with fh (at the offset position) a total number of count data items having datatype type from the user's buffer buf. The offset is in etype units relative to the current view. The data is written into those parts of the file specified by the current view. MPI_WRITE_EXPLICIT stores the number of datatype elements actually written in status. All other fields of status are undefined.

MPI_WRITE_EXPLICIT_ALL(fh, offset, buf, count, datatype, status)

```
int MPI_Write_explicit_all(MPI_File fh, MPI_Offset offset, void *buf,
              int count, MPI_Datatype datatype, MPI_Status *status)
```

```
MPI_WRITE_EXPLICIT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER*8 OFFSET
```

MPI_WRITE_EXPLICIT_ALL is a collective version of the blocking MPI_WRITE_EXPLICIT interface. This routine is semantically identical to MPI_WRITE_EXPLICIT, except that this routine is collective.

MPI_IREAD_EXPLICIT(fh, offset, buf, count, datatype, request)

IN	fh	file handle (handle)
IN	offset	file offset (integer)
OUT	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	request	request object (handle)

```
int MPI_Iread_explicit(MPI_File fh, MPI_Offset offset, void *buf, int count,
              MPI_Datatype datatype, MPI_Request *request)
```

```
MPI_IREAD_EXPLICIT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
    INTEGER*8 OFFSET
```

MPI_IREAD_EXPLICIT is a nonblocking version of the MPI_READ_EXPLICIT interface. MPI_IREAD_EXPLICIT associates a request handle request with the I/O request. The request handle can be used later to query the status of the read request (via MPI_TEST), or wait for its completion (via MPI_WAIT).

The nonblocking read call indicates that the system can start to read data into the supplied buffer. MPI_IREAD_EXPLICIT attempts to read from the file associated with fh (at the offset position), a total number of count data items having datatype type into the user's buffer buf. The offset is in etype units relative to the current view. The number of datatype elements actually read can be extracted from the MPI_TEST or MPI_WAIT return status, using the routines MPI_GET_COUNT or MPI_GET_ELEMENTS. In addition, the MPI_ERROR field of the return status will contain MPI_SUCCESS if the data access was successful, or a specific error code indicating failure. All other fields of the return status are undefined.

MPI_IREAD_EXPLICIT_ALL(fh, offset, buf, count, datatype, request)

```
int MPI_Iread_explicit_all(MPI_File fh, MPI_Offset offset, void *buf,
              int count, MPI_Datatype datatype, MPI_Request *request)
```

```
MPI_IREAD_EXPLICIT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
    INTEGER*8 OFFSET
```

MPI_IREAD_EXPLICIT_ALL is a collective version of the nonblocking MPI_IREAD_EXPLICIT interface. This routine is semantically identical to MPI_IREAD_EXPLICIT, except that this routine is collective.

MPI_IWRITE_EXPLICIT(fh, offset, buf, count, datatype, request)

IN	fh	file handle (handle)
IN	offset	file offset (integer)
IN	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	request	request object (handle)

```
int MPI_Iwrite_explicit(MPI_File fh, MPI_Offset offset, void *buf,
             int count, MPI_Datatype datatype, MPI_Request *request)
```

```
MPI_IWRITE_EXPLICIT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
    INTEGER*8 OFFSET
```

MPI_IWRITE_EXPLICIT is a nonblocking version of the MPI_WRITE_EXPLICIT interface. MPI_IWRITE_EXPLICIT associates a request handle request with the I/O request. The request handle can be used later to query the status of the write request (via MPI_TEST), or wait for its completion (via MPI_WAIT).

The nonblocking write call indicates that the system can start to write data from the supplied buffer. MPI_IWRITE_EXPLICIT attempts to write into the file associated with fh (at the offset position), a total number of count data items having datatype type from the user's buffer buf. The offset is in etype units relative to the current view. The number of datatype elements actually written can be extracted from the MPI_TEST or MPI_WAIT return status, using the routines MPI_GET_COUNT or MPI_GET_ELEMENTS. In addition, the MPI_ERROR field of the return status will contain MPI_SUCCESS if the data access was successful, or a specific error code indicating failure. All other fields of the return status are undefined.

MPI_IWRITE_EXPLICIT_ALL(fh, offset, buf, count, datatype, request)

```
int MPI_Iwrite_explicit_all(MPI_File fh, MPI_Offset offset, void *buf,
             int count, MPI_Datatype datatype, MPI_Request *request)
```

```
MPI_IWRITE_EXPLICIT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
    INTEGER*8 OFFSET
```

MPI_IWRITE_EXPLICIT_ALL is a collective version of the nonblocking MPI_IWRITE_EXPLICIT interface. This routine is semantically identical to MPI_IWRITE_EXPLICIT, except that this routine is collective.

10.4.3 Data Access With Individual File Pointers

The system maintains one individual file pointer per process per collective MPI_OPEN. The current value of this pointer can be used to implicitly specify the offset of data accesses, by using the routines described in this section. These routines only use and update the individual file pointers maintained by the system. The shared file pointer is not used nor updated.

The individual file pointer routines have the same semantics as the data access with explicit offset routines described in Section 10.4.2 with the following modification:

- the offset is defined to be the current value of the system-maintained individual file pointer.

After an individual file pointer operation is initiated, the individual file pointer is updated to point to the next elementary data item after the last one requested (regardless of how many items are actually accessed). The file pointer is updated relative to the current view of the file.

MPI_READ(fh, buf, count, datatype, status)

IN	fh	file handle (handle)
OUT	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	status	status object (Status)

```
int MPI_Read(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
             MPI_Status *status)
```

```
MPI_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

MPI_READ is semantically identical to MPI_READ_EXPLICIT, with offset set to the current individual file pointer position maintained by the system.

MPI_READ_ALL(fh, buf, count, datatype, status)

```
int MPI_Read_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
             MPI_Status *status)
```

```
MPI_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

MPI_READ_ALL is a collective version of the blocking MPI_READ interface. This routine is semantically identical to MPI_READ, except that this routine is collective.

MPI_WRITE(fh, buf, count, datatype, status)

IN	fh	file handle (handle)
IN	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	status	status object (Status)

```
int MPI_Write(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
              MPI_Status *status)
```

```
MPI_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

MPI_WRITE is semantically identical to MPI_WRITE_EXPLICIT, with offset set to the current individual file pointer position maintained by the system.

MPI_WRITE_ALL(fh, buf, count, datatype, status)

```
int MPI_Write_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
              MPI_Status *status)
```

```
MPI_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

MPI_WRITE_ALL is a collective version of the blocking MPI_WRITE interface. This routine is semantically identical to MPI_WRITE, except that this routine is collective.

MPI_IREAD(fh, buf, count, datatype, request)

IN	fh	file handle (handle)
OUT	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	request	request object (handle)

```
int MPI_Iread(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
              MPI_Request *request)
```

```
MPI_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
```

MPI_IREAD is semantically identical to MPI_IREAD_EXPLICIT, with offset set to the current individual file pointer position maintained by the system.

MPI_IREAD_ALL(fh, buf, count, datatype, request)

```
int MPI_Iread_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
             MPI_Request *request)
```

```
MPI_IREAD_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
```

MPI_IREAD_ALL is a collective version of the blocking MPI_IREAD interface. This routine is semantically identical to MPI_IREAD, except that this routine is collective.

MPI_IWRITE(fh, buf, count, datatype, request)

IN	fh	file handle (handle)
IN	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	request	request object (handle)

```
int MPI_Iwrite(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
             MPI_Request *request)
```

```
MPI_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
```

MPI_IWRITE is semantically identical to MPI_IWRITE_EXPLICIT, with offset set to the current individual file pointer position maintained by the system.

MPI_IWRITE_ALL(fh, buf, count, datatype, request)

```
int MPI_Iwrite_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
             MPI_Request *request)
```

```
MPI_IWRITE_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
```

MPI_IWRITE_ALL is a collective version of the blocking MPI_IWRITE interface. This routine is semantically identical to MPI_IWRITE, except that this routine is collective.

MPI_SEEK(fh, offset, whence)

IN	fh	file handle (handle)
IN	offset	file offset (integer)
IN	whence	update mode (integer)

```
int MPI_Seek(MPI_File fh, MPI_Offset offset, int whence)
```

```
MPI_SEEK(FH, OFFSET, WHENCE, IERROR)
    INTEGER FH, WHENCE, IERROR
    INTEGER*8 OFFSET
```

MPI_SEEK updates the individual file pointer according to whence, which could have the following possible values:

- MPI_SEEK_SET: the pointer is set to offset
- MPI_SEEK_CUR: the pointer is set to the current pointer position plus offset
- MPI_SEEK_END: the pointer is set to the end of the view plus offset

The offset can be negative, which allows seeking backwards. It is erroneous to seek to a negative position in the view. The end of the view is defined to be the position of the next elementary data item, relative to the current view, following the last whole elementary data item accessible.

MPI_FILE_GET_POSITION(fh, offset, byte_offset)

IN	fh	file handle (handle)
OUT	offset	offset of individual pointer (integer)
OUT	byte_offset	displacement of individual pointer (integer)

```
int MPI_File_get_position(MPI_File fh, MPI_Offset *offset,
            MPI_Offset *byte_offset)
```

```
MPI_FILE_GET_POSITION(FH, OFFSET, BYTE_OFFSET, IERROR)
    INTEGER FH, IERROR
    INTEGER*8 OFFSET, BYTE_OFFSET
```

MPI_FILE_GET_POSITION returns, in offset, the current position of the individual file pointer in etype units relative to the current displacement and filetype. It also returns, in byte_offset, the absolute position in bytes of the individual file pointer, ignoring the displacement and filetype.

Advice to users. The offset can be used in a future call to MPI_SEEK using MPI_SEEK_SET to return to the current position. The byte_offset is useful for setting the displacement to the current position via MPI_FILE_SET_VIEW. (*End of advice to users.*)

10.4.4 Data Access with Shared File Pointers

The system maintains exactly one shared file pointer per collective MPI_OPEN (shared among processes in the communicator group). The current value of this pointer can be used to implicitly specify the offset of data accesses, by using the routines described in this section. These routines only use and update the shared file pointer maintained by the system. The individual file pointers are not used nor updated.

The shared file pointer routines have the same semantics as the data access with explicit offset routines described in Section 10.4.2 with the following modifications:

- the offset is defined to be the current value of the system-maintained shared file pointer,
- the effect of multiple calls to shared file pointer routines is defined to behave as if the calls were serialized, and
- the use of shared file pointer routines is erroneous unless all processes use the same file view.

The serialization ordering is not deterministic. The user needs to use other synchronization means to enforce a specific order.

After a shared file pointer operation is initiated, the shared file pointer is updated to point to the next elementary data item after the last one requested (regardless of how many items are actually accessed). The file pointer is updated relative to the current view of the file.

> *Advice to users.* The shared file pointer routines only make sense if all the processes can access the same data items. This means that all the processes must use the same file view. For example, consider a single MPI_FILE_SET_VIEW in which process 0 uses filetype MPI_CHAR and process 1 uses filetype MPI_DOUBLE. If process 0 reads a single character from the file, the shared file pointer points to the second character relative to process 0, but points into the middle of the first double relative to process 1. At this point, the effect of a shared pointer operation on process 1 would be ill-defined. (*End of advice to users.*)

Non-collective Operations

MPI_READ_SHARED(fh, buf, count, datatype, status)

IN	fh	file handle (handle)
OUT	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	status	status object (Status)

```
int MPI_Read_shared(MPI_File fh, void *buf, int count,
            MPI_Datatype datatype, MPI_Status *status)
```

```
MPI_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

MPI_READ_SHARED is semantically identical to MPI_READ_EXPLICIT, with offset set to the current shared file pointer position maintained by the system. Shared file pointer calls within a single file handle communicator group are guaranteed to be sequentially consistent.

MPI_WRITE_SHARED(fh, buf, count, datatype, status)

IN	fh	file handle (handle)
IN	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	status	status object (Status)

```
int MPI_Write_shared(MPI_File fh, void *buf, int count,
             MPI_Datatype datatype, MPI_Status *status)
```

```
MPI_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

MPI_WRITE_SHARED is semantically identical to MPI_WRITE_EXPLICIT, with offset set to the current shared file pointer position maintained by the system. Shared file pointer calls within a single file handle communicator group are guaranteed to be sequentially consistent.

MPI_IREAD_SHARED(fh, buf, count, datatype, request)

IN	fh	file handle (handle)
OUT	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	request	request object (handle)

```
int MPI_Iread_shared(MPI_File fh, void *buf, int count,
             MPI_Datatype datatype, MPI_Request *request)
```

```
MPI_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
```

MPI_IREAD_SHARED is semantically identical to MPI_IREAD_EXPLICIT, with offset set to the current shared file pointer position maintained by the system. Shared file pointer calls within a single file handle communicator group are guaranteed to be sequentially consistent.

MPI_IWRITE_SHARED(fh, buf, count, datatype, request)

IN	fh	file handle (handle)
IN	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	request	request object (handle)

```
int MPI_Iwrite_shared(MPI_File fh, void *buf, int count,
              MPI_Datatype datatype, MPI_Request *request)
```

```
MPI_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
```

MPI_IWRITE_SHARED is semantically identical to MPI_IWRITE_EXPLICIT, with offset set to the current shared file pointer position maintained by the system. Shared file pointer calls within a single file handle communicator group are guaranteed to be sequentially consistent.

Collective Operations

The semantics of a collective access using a shared file pointer is that the accesses to the file will be in the order determined by the ranks of the processes within the group associated with the communicator. In addition, in order to prevent subsequent shared offset accesses by the same processes from interfering with this collective access, the call can only return after all the processes within the group have initiated their accesses. When the call returns, the shared file pointer points to the next byte accessible, according to the filetype used by all processes, after the last byte requested.

> *Advice to users.* There may be some programs in which all processes in the communicator group associated with the open file 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_WRITE_SHARED_ORDERED rather than MPI_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_READ_SHARED_ORDERED(fh, buf, count, datatype, status)

IN	fh	file handle (handle)
OUT	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	status	status object (Status)

```
int MPI_Read_shared_ordered(MPI_File fh, void *buf, int count,
              MPI_Datatype datatype, MPI_Status *status)
```

```
MPI_READ_SHARED_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

MPI_READ_SHARED_ORDERED is a collective version of the MPI_READ_SHARED interface. This routine must be called by all processes in the communicator group associated with the file handle fh. Each process may pass different argument values for the datatype, and count arguments. After all processes of the group have issued their respective calls, each process attempts to read, from the file associated with fh, a total number of count data items having datatype type into the user's buffer buf. For each process, the location in the file at which data is read is the position at which the shared file pointer would be after all processes whose ranks within the group are less than that of this process had read their data. MPI_READ_SHARED_ORDERED returns the number of datatype elements read in status. The shared file pointer is updated by the amounts of data requested by all processes of the group.

MPI_WRITE_SHARED_ORDERED(fh, buf, count, datatype, status)

IN	fh	file handle (handle)
IN	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	status	status object (Status)

```
int MPI_Write_shared_ordered(MPI_File fh, void *buf, int count,
              MPI_Datatype datatype, MPI_Status *status)
```

```
MPI_WRITE_SHARED_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

MPI_WRITE_SHARED_ORDERED is a collective version of the blocking MPI_WRITE_SHARED interface. All processes in the communicator group associated with the file handle fh must call MPI_WRITE_SHARED_ORDERED. Each process may pass different argument values for the datatype and count arguments. After all processes of the group have issued their respective calls, each process attempts to write, into the file associated with fh, a total number of count data items having datatype type contained in the user's buffer buf. For each process, the location in the file at which data is written is the position at which the shared file pointer would be after all processes whose ranks within the group are less than that of this process had written their data. MPI_WRITE_SHARED_ORDERED returns the number of datatype elements written in status. The shared file pointer is updated by the amounts of data requested by all processes of the group.

MPI_IREAD_SHARED_ORDERED(fh, buf, count, datatype, request)

IN	fh	file handle (handle)
OUT	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	request	request object (handle)

```
int MPI_Iread_shared_ordered(MPI_File fh, void *buf, int count,
             MPI_Datatype datatype, MPI_Request *request)
```

```
MPI_IREAD_SHARED_ORDERED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
```

MPI_IREAD_SHARED_ORDERED is a collective version of the nonblocking MPI_IREAD_SHARED interface. All processes in the communicator group associated with the file handle fh must call MPI_IREAD_SHARED_ORDERED. Each process may pass different argument values for the datatype and count arguments. After all processes of the group have issued their respective calls, each process attempts to read, from the file associated with fh, a total number of count data items having datatype type into the user's buffer buf. For each process, the location in the file at which data will be read is the position at which the shared file pointer would be after all processes whose ranks within the group are less than that of this process had read their data. MPI_IREAD_SHARED_ORDERED returns immediately and does not wait for the data to be present in the user's buffer. MPI_IREAD_SHARED_ORDERED associates an individual request handle request to the I/O request for each process. The request handle can be used later by a process to query the status of its individual read request or wait for its completion. On each process, MPI_IREAD_SHARED_ORDERED completes when the individual request has completed (i.e., a process does not have to wait for all other requests to complete). The user should not access any part of the receiving buffer after a nonblocking read is called, until the read completes. The shared file pointer is updated by the amounts of data requested by all processes of the group. For each process, the number of datatype elements actually read by that process can be extracted from the MPI_TEST or MPI_WAIT return status, using the routines MPI_GET_COUNT or MPI_GET_ELEMENTS.

MPI_IWRITE_SHARED_ORDERED(fh, buf, count, datatype, request)

IN	fh	file handle (handle)
IN	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	request	request object (handle)

```
int MPI_Iwrite_shared_ordered(MPI_File fh, void *buf, int count,
              MPI_Datatype datatype, MPI_Request *request)
```

```
MPI_IWRITE_SHARED_ORDERED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
```

MPI_IWRITE_SHARED_ORDERED is a collective version of the nonblocking MPI_IWRITE_SHARED interface. All processes in the communicator group associated with the file handle fh must call MPI_IWRITE_SHARED_ORDERED. Each process may pass different argument values for the datatype and count arguments. After all processes of the group have issued their respective calls, each process attempts to write, into the file associated with fh, a total number of count data items having datatype type contained in the user's buffer buf. For each process, the location in the file at which data will be written is the position at which the shared file pointer would be after all processes whose ranks within the group are less than that of this process had written their data. MPI_IWRITE_SHARED_ORDERED returns immediately and does not wait for the data to be copied out of the user's buffer. MPI_IWRITE_SHARED_ORDERED associates an individual request handle request to the I/O request for each process. The request handle can be used later by a process to query the status of its individual write request or wait for its completion. On each process, MPI_IWRITE_SHARED_ORDERED completes when the individual write request has completed (i.e., a process does not have to wait for all other requests to complete). The user should not access any part of the supplied buffer after a nonblocking write is called, until the write is completed. The shared file pointer is updated by the amounts of data requested by all processes of the group. For each process, the number of datatype elements actually written by that process can be extracted from the MPI_TEST or MPI_WAIT return status, using the routines MPI_GET_COUNT or MPI_GET_ELEMENTS.

Seek

MPI_SEEK_SHARED(fh, offset, whence)

IN	fh	file handle (handle)
IN	offset	file offset (integer)
IN	whence	update mode (integer)

```
int MPI_Seek_shared(MPI_File fh, MPI_Offset offset, int whence)
```

```
MPI_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR)
```

```
    INTEGER FH, WHENCE, IERROR
    INTEGER*8 OFFSET
```

MPI_SEEK_SHARED updates the shared file pointer according to whence, which could have the following possible values:

- MPI_SEEK_SET: the pointer is set to offset
- MPI_SEEK_CUR: the pointer is set to the current pointer position plus offset
- MPI_SEEK_END: the pointer is set to the end of the view plus offset

MPI_SEEK_SHARED is collective; all the processes in the communicator group associated with the file handle fh must call MPI_SEEK_SHARED with the same offset and whence. All processes in the communicator group are synchronized with a barrier before the shared file pointer is updated.

The offset can be negative, which allows seeking backwards. It is erroneous to seek to a negative position in the view. The end of the view is defined to be the position of the next elementary data item, relative to the current view, following the last whole elementary data item accessible.

MPI_FILE_GET_POSITION_SHARED(fh, offset, byte_offset)

IN	fh	file handle (handle)
OUT	offset	offset of shared pointer (integer)
OUT	byte_offset	displacement of shared pointer (integer)

```
int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset,
            MPI_Offset *byte_offset)
```

```
MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, BYTE_OFFSET, IERROR)
    INTEGER FH, IERROR
    INTEGER*8 OFFSET, BYTE_OFFSET
```

MPI_FILE_GET_POSITION_SHARED returns, in offset, the current position of the shared file pointer in etype units relative to the current displacement and filetype. It also returns, in byte_offset, the absolute position in bytes of the shared file pointer, ignoring the displacement and filetype.

> *Advice to users.* The offset can be used in a future call to MPI_SEEK_SHARED using MPI_SEEK_SET to return to the current position. The byte_offset is useful for setting the displacement to the current position via MPI_FILE_SET_VIEW. (*End of advice to users.*)

10.5 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 via flags to MPI_OPEN.

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 10.6.1) provided that it would have been possible to start the two processes simultaneously and have them reside in a single MPI_COMM_WORLD. Further, 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, the file view, and in cases where heterogeneous message passing is supported, regardless of the hardware architecture. It should be noted that without single environment file interoperability, the I/O facilities in MPI would be next to useless.

There are three aspects to file interoperability outside of a single environment:

- transferring the bits,
- converting between different file structures, and
- converting between different machine representations.

The first two aspects of file interoperability are beyond the scope of this standard, as both are highly machine dependent. However, transferring the bits of a file into and out of the MPI environment (e.g., by writing a file to tape) is required to be supported by all MPI implementations. In particular, an implementation must specify how familiar operations equivalent to UNIX `cp`, `rm`, `mv`, etc. can be performed on the file. Further, it is expected that the facility provided maintains the correspondence between absolute byte offsets (e.g., after possible file structure conversion, the data bits at byte offset 102 in the MPI environment are at byte offset 102 outside the MPI environment). As an example, a simple off-line conversion utility which transfers and converts files between the native file system and the MPI environment would suffice, provided it maintained the offset coherence mentioned above. A high quality implementation of MPI will provide an offset-coherent, native file interface to the files contained in the MPI environment.

The remaining aspect of file interoperability, converting between different machine representations, is not only desired, but also well-supported by the typing information specified in the etype and filetype. This facility allows the information in files to be shared between any two applications, regardless of whether they use MPI, and regardless of the machine architectures on which they run.

MPI supports three data representations: native, internal, and canonical. The native and internal data representations are implementation dependent, while the canonical representation is "common" to all MPI implementations and facilitates file interoperability. The data mode is specified in the *amode* argument when a file is opened.

> *Advice to users.* MPI does not support self-describing files (e.g., HDF), and retains no knowledge of the data mode used during file creation. Therefore, an MPI application is responsible for specifying the same data mode on each file open in order to correctly retrieve the file data. (*End of advice to users.*)

MPI_DATA_NATIVE This mode states that no data type conversions are performed during read and write operations. The data is represented on the storage medium exactly as it is in memory. The advantages to this mode are that data precision and I/O performance are not lost in type conversions, in a purely homogeneous environment.

The disadvantage is the loss of transparent interoperability within a heterogeneous MPI environment.

Advice to users. This mode 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 insure that the message routines do not perform any type conversions on the data. (*End of advice to implementors.*)

MPI_DATA_INTERNAL In specifying this mode, the application is stating that it wants automatic data type conversions on I/O operations within a heterogeneous MPI environment, but it does not care in which representation the data is actually stored.

This mode has the advantage that data type conversion is not necessary if the file data is stored in the same representation as the I/O requester. The disadvantage is that files written in this mode may not be usefully exported from the MPI implementation without knowledge of the data types used in writing the file.

An example implementation of MPI_DATA_INTERNAL is one that always stores data in the representation of the machine that requires the best I/O performance.

Advice to implementors. MPI_DATA_CANONICAL is a superset of the functionality required by MPI_DATA_INTERNAL. An implementation may therefore choose to implement MPI_DATA_INTERNAL as MPI_DATA_CANONICAL. (*End of advice to implementors.*)

MPI_DATA_CANONICAL This mode states that read and write operations convert all data from and to its canonical representation respectively. The data on the storage medium is always in its canonical representation, and the data in memory is always in the representation of the process that placed it there.

This mode has a couple of advantages. First, all nodes 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 out of the original MPI implementation and imported into another MPI implementation with the guarantee that all automatic data type conversions will continue to function.

The disadvantages of this mode are that data precision and I/O performance may be lost in the 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 canonical 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.*)

10.5.1 Datatype Size and Extent in Files

Data representations in MPI_DATA_CANONICAL mode and MPI_DATA_INTERNAL mode may be smaller or larger than the equivalent native representations. For this reason, it is important to avoid absolute file addressing. Displacements and seek offsets should be calculated by using the routines MPI_FILE_GET_TYPE_SIZE and MPI_FILE_GET_TYPE_EXTENT which return the size and extent of an MPI datatype in an open file.

MPI_FILE_GET_TYPE_SIZE(fh, datatype, size)

IN	fh	file handle (handle)
IN	datatype	datatype (handle)
OUT	size	datatype size (integer)

```
int MPI_File_get_type_size(MPI_File fh, MPI_Datatype datatype,
             MPI_Aint *size)
```

```
MPI_FILE_GET_TYPE_SIZE(FH, DATATYPE, SIZE, IERROR)
    INTEGER FH, DATATYPE, IERROR
    INTEGER(KIND = MPI_ADDRESS_SIZE) SIZE
```

Returns the size of datatype in the file fh.

MPI_FILE_GET_TYPE_EXTENT(fh, datatype, extent)

IN	fh	file handle (handle)
IN	datatype	datatype (handle)
OUT	extent	datatype extent (integer)

```
int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype,
             MPI_Aint *extent)
```

```
MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERROR)
    INTEGER FH, DATATYPE, IERROR
    INTEGER(KIND = MPI_ADDRESS_SIZE) EXTENT
```

Returns the extent of datatype in the file fh.

10.5.2 Canonical Data Representation

Editors note: The forum is currently working on defining the data representation for MPI_DATA_CANONICAL.

10.6 Consistency and Semantics

10.6.1 File Consistency

Consistency semantics define the outcome of multiple accesses to a single file. All file accesses in MPI are relative to a specific file handle created from a collective open. MPI provides three levels of consistency: sequential consistency among all accesses using a single file handle, sequential consistency among all accesses using file handles created from a single collective open with atomic mode enabled, and weak consistency among accesses not handled above via the MPI_FILE_SYNC routine. Sequential consistency means the behavior of a set of operations will be as if the operations were performed in some serial order; each access appears atomic, although the exact ordering of accesses is unspecified.

First, all operations on a single file handle are sequentially consistent. (Note that all operations on a single file handle are performed by exactly one process.) As far as consistency is concerned, nonblocking operations behave identically to blocking operations. The data accessed in nonblocking operations is identical to the data that would have been accessed for the equivalent blocking operation.

Second, consider accesses using file handles created from a single collective open. Let FH_1 be the set of file handles created from one particular collective open of the file FOO, and FH_2 be the set of file handles created from a different collective open of FOO. Note that nothing restrictive is said about FH_1 and FH_2: the sizes of FH_1 and FH_2 may be different, the set of processes used for each open may not overlap, the file handles in FH_1 may be destroyed before those in FH_2 are created, etc.

MPI guarantees that all file accesses using file handles in FH_1 (where the file regions accessed do not overlap) are sequentially consistent, and hence, appear to the application as atomic. (Note that this is automatically true, as it is impossible to produce non-sequentially consistent behavior among accesses that do not overlap.) However, unlike UNIX, the default semantics for overlapping accesses does not guarantee sequential consistency. This default mode (set when a file is opened) is referred to as *non-atomic* mode. In non-atomic mode, all data in regions of the file which had overlapping accesses is undefined, unless MPI_FILE_SYNC is used to ensure weak consistency. Atomic access can be guaranteed for overlapping accesses by enabling atomic mode for the file via the MPI_FILE_SET_ATOMICITY routine.

Changing the consistency semantics for an open file only affects new data accesses. All completed data accesses are guaranteed to abide by the consistency semantics in effect during their execution. Nonblocking data accesses which have not completed (e.g., via MPI_WAIT) are only guaranteed to abide by non-atomic mode consistency semantics.

> *Advice to implementors.* Since the semantics guaranteed by atomic mode are stronger than those guaranteed by non-atomic mode, an implementation is free to adhere to the more stringent atomic mode semantics for outstanding requests. (*End of advice to implementors.*)

Finally, consider access to a single file using file handles from distinct collective opens (e.g., fh_1 from FH_1 and fh_2 from FH_2), or from a single collective open with non-atomic mode set. The default behavior does not guarantee sequential consistency. In order to guarantee sequential consistency, MPI_FILE_SYNC must be used (both opening and closing a file implicitly perform an MPI_FILE_SYNC).

Let SEQ_{fh} be a sequence of data access operations on a single file handle, bracketed by MPI_FILE_SYNCs of that file handle. SEQ_{fh} is a "write sequence" if any of the data access operations in the sequence are writes. Given two sequences, SEQ_1 and SEQ_2, we say they do not overlap if one sequence is guaranteed to completely precede the other. 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 overlaps SEQ_1.

MPI_FILE_SET_ATOMICITY(fh, flag)

IN	fh	file handle (handle)
IN	flag	true to set atomic mode, false to set non-atomic mode (boolean)

```
int MPI_File_set_atomicity(MPI_File fh, int flag)
```

```
MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR)
    INTEGER FH, FLAG, IERROR
```

Let FH be the set of file handles created by one collective open. The consistency semantics for data access operations using FH is set by collectively calling MPI_FILE_SET_ATOMICITY on FH. MPI_FILE_SET_ATOMICITY is collective; all processes in the group must pass identical values for fh and flag. If flag is true, atomic mode is set, if flag is false, non-atomic mode is set.

MPI_FILE_GET_ATOMICITY(fh, flag)

IN	fh	file handle (handle)
OUT	flag	true if atomic mode, false if non-atomic mode (boolean)

```
int MPI_File_get_atomicity(MPI_File fh, int *flag)
```

```
MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR)
    INTEGER FH, FLAG, IERROR
```

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, non-atomic mode is enabled.

MPI_FILE_SYNC(fh)

IN	fh	file handle (handle)

```
int MPI_File_sync(MPI_File fh)
```

```
MPI_FILE_SYNC(FH, IERROR)
    INTEGER FH, IERROR
```

MPI_FILE_SYNC causes either: all previous writes to fh to be written to permanent storage, or all updates made by other processes to permanent storage to be visible in subsequent reads of fh. MPI_FILE_SYNC is a collective operation.

MPI_FILE_SYNC guarantees that all *completed* write requests have been flushed to permanent storage. However, for pending nonblocking read and write requests that have not been completed (e.g., via MPI_WAIT), the value of each byte of data accessed will be equivalent to either the value which would have been accessed before the MPI_FILE_SYNC was called, or the value which would have been accessed after the MPI_FILE_SYNC completes. The exact behavior (which of these two choices) of each byte of each outstanding operation is implementation dependent.

Advice to users. The user is responsible for ensuring that all outstanding nonblocking requests of interest are completed before calling MPI_FILE_SYNC. (*End of advice to users.*)

10.6.2 Progress

The progress rules of MPI are both a promise to users, and a set of constraints on implementors. In cases where the progress rules restrict possible implementation choices more than the interface specification alone, the progress rules take precedence.

All blocking routines must complete in finite time unless an exceptional condition (such as resource exhaustion) causes an error.

Nonblocking data access routines inherit the following progress rule from nonblocking point to point communication: a nonblocking write is equivalent to a nonblocking send for which a receive is eventually posted, and a nonblocking read is equivalent to a nonblocking receive for which a send is eventually posted.

For the nonblocking file manipulation routines (MPI_IOPEN and MPI_ICLOSE), an implementation is free to delay progress until the request associated with the nonblocking file manipulation routine is tested or waited upon (e.g., by a call to MPI_TEST).

Rationale. The less stringent progress rule for MPI_IOPEN and MPI_ICLOSE permits the optimization of file staging. For example, an implementation can bundle numerous open operations into a single request, allowing the mass storage system to stage files in from tape simultaneously. (*End of rationale.*)

Finally, an implementation is free to delay progress of collective routines until all processes in the group associated with the collective call have invoked the routine. Once all processes in the group have invoked the routine, the progress rule of the equivalent noncollective routine must be followed.

10.6.3 Type Matching

The type matching rules for I/O mimic the type matching rules for communication with one exception: the type MPI_BYTE matches any type in a data access. In general, the etype of data items written must textually 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. An etype or data access buffer type constructed entirely from MPI_BYTE matches all types.

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.*)

10.6.4 Logical vs. Physical File Layout

This standard 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 will be necessary in order to optimize disk layout. This information is provided as *hints* specified via *info* when a file is created (see section 10.2.9).

10.7 Filetype Constructors

10.7.1 Introduction

Common I/O operations (e.g., rank-ordered blocks, structured array accesses, etc.) are easily expressed using the previously defined read/write operations and carefully defined filetypes. In order to simplify generation of common filetypes, the following MPI datatype constructors are provided. Note that these datatype constructors create defined datatypes, not intrinsic or fundamental datatypes. In other words, datatypes decoded using the interface MPI_GET_TYPE_ENVELOPE (see section 7.7.1), will not yield "combiners" in this section.

Each new filetype created newtype consists of zero or more copies of the base type oldtype, possibly separated by holes. Oldtype can be any MPI basic or derived datatype, and need not be contiguous. The extent of the new filetype is a nonnegative integer multiple of the extent of the base type. All filetype constructors return a success or failure code.

Although it is expected that these routines will be commonly used in a collective fashion, all the filetype constructors are independent. This facilitates using these constructors for defined datatypes as well as filetypes.

10.7.2 Scatter / Gather Type Constructors

MPI_TYPE_SCATTER_GATHER(size, rank, oldtype, newtype)

IN	size	size of communicator used in MPI_Open (nonnegative integer)
IN	rank	rank in communicator used in MPI_Open (nonnegative integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_scatter_gather(int size, int rank, MPI_Datatype oldtype,
              MPI_Datatype *newtype)
```

```
MPI_TYPE_SCATTER_GATHER(SIZE, RANK, OLDTYPE, NEWTYPE, IERROR)
    INTEGER SIZE, RANK, OLDTYPE, NEWTYPE, IERROR
```

The newly created filetype newtype allows each process of a group to access a distinct block of the file in rank order. The blocks are identical in size and datatype; each item is of type oldtype.

To achieve the scatter or gather operation, this routine should be used collectively, with each process passing identical values for size and oldtype. The size of the group should be passed in size, the process rank in rank, and the base type in oldtype. The file must be opened with the returned filetypes, and subsequently accessed with collective read or write operations with identical offsets.

MPI_TYPE_SCATTERV_GATHERV(size, rank, count, oldtype, newtype)

IN	size	size of communicator used in MPI_Open (nonnegative integer)
IN	rank	rank in communicator used in MPI_Open (nonnegative integer)
IN	count	number of elements of oldtype in this block (nonnegative integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_scatterv_gatherv(int size, int rank, int count,
             MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
MPI_TYPE_SCATTERV_GATHERV(SIZE, RANK, COUNT, OLDTYPE, NEWTYPE, IERROR)
    INTEGER SIZE, RANK, COUNT, OLDTYPE, NEWTYPE, IERROR
```

The newly created filetype **newtype** allows each process of a group to access a distinct *block* of the file in rank order as in MPI_Type_scatter_gather, except that the block sizes and types may be different. Each block is defined as **count** contiguous copies of the passed datatype **oldtype**.

10.7.3 Local Array Filetype Constructor

MPI_TYPE_SUBARRAY(ndims, sizes, subsizes, starts, order, oldtype, newtype)

IN	ndims	number of array dimensions (nonnegative integer)
IN	sizes	size of the full array in each dimension (array of nonnegative integers)
IN	subsizes	size of the sub-array in each dimension (array of nonnegative integers)
IN	starts	starting coordinates of the sub-array in each dimension (array of nonnegative integers)
IN	order	array storage order flag (integer)
IN	oldtype	array element datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_subarray(int ndims, int *sizes, int *subsizes, int *starts,
             int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
MPI_TYPE_SUBARRAY(NDIMS, SIZES, SUBSIZES, STARTS, ORDER, OLDTYPE, NEWTYPE,
             IERROR)
```

```
INTEGER NDIMS, SIZES(*), SUBSIZES(*), STARTS(*), ORDER, OLDTYPE,
NEWTYPE, IERROR
```

The local sub-array filetype constructor creates an MPI datatype describing an n-dimensional sub-array of a n-dimensional array. The sub-array may be situated anywhere within the full array, and may be of any size up to the size of the larger array as long as it is confined within this array. This type constructor facilitates accessing arrays distributed in blocks among processes to a single file that contains the global array.

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 names are only meant to specify the default language behavior.

To specify in which order the array is stored, the order argument is set to one of the following:

- MPI_C_ORDER - The ordering used by C arrays, (i.e., row-major order)
- MPI_FORTRAN_ORDER - The ordering used by FORTRAN arrays, (i.e., column-major order)

Although arrays in C and C++ are indexed starting at 0, and arrays in FORTRAN are indexed starting at 1, the starts argument is always indexed starting at 0.

10.7.4 Distributed Array Filetype Constructors

The distributed array filetype constructors support HPF-like [11] 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 these filetype constructors as follows. Complementary filetypes are created by having every process of a group call a 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). Using this view, a collective data access operation (with identical offsets) will yield an HPF-like distribution pattern. (*End of advice to users.*)

MPI_TYPE_DARRAY(size, rank, ndims, dsizes, distribs, dargs, pdims, psizes, order, oldtype, newtype)

IN	size	size of communicator used in MPI_Open (nonnegative integer)
IN	rank	rank in communicator used in MPI_Open (nonnegative integer)
IN	ndims	number of array dimensions (nonnegative integer)
IN	dsizes	size of distributed array in each dimension (array of nonnegative integer)
IN	distribs	distribution of array in each dimension, e.g., BLOCK or CYCLIC (array of integer)
IN	dargs	distribution argument of dimension, e.g., BLOCK(dargs) or CYCLIC(dargs) (array of integer)
IN	pdims	number of processor grid dimensions (nonnegative integer)
IN	psizes	size of processor grid in each dimension (array of nonnegative integer)
IN	order	array storage order flag (integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_darray(int size, int rank, int ndims, int *dsizes, int
            *distribs, int *dargs, int pdims, int *psizes, int order,
            MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
MPI_TYPE_DARRAY(SIZE, RANK, NDIMS, DSIZES, DISTRIBS, DARGS, PDIMS, PSIZES,
            ORDER, OLDTYPE, NEWTYPE, IERROR)
    INTEGER SIZE, RANK, NDIMS, DSIZES(*), DISTRIBS(*), DARGS(*), PDIMS,
    PSIZES(*)
    INTEGER ORDER, OLDTYPE, NEWTYPE, IERROR
```

MPI_TYPE_DARRAY generates the filetypes corresponding to the distribution of an ndims-dimensional array of oldtype elements onto a pdims-dimensional grid of logical processors. For a call to MPI_TYPE_DARRAY to be correct, the equation $\prod_{i=0}^{pdims-1} psizes[i] = size$ must be satisfied.

Each dimension of the array is distributed in one of three ways:

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

To specify a default distribution argument, the constant MPI_DISTRIBUTE_DFLT_ARG is used. The distribution argument for a dimension that is not distributed is ignored.

For example, ARRAY(CYCLIC(15)) corresponds to MPI_DISTRIBUTE_CYCLIC with a distribution argument of 15, and ARRAY(BLOCK) corresponds to MPI_DISTRIBUTE_BLOCK with a distribution argument of MPI_DISTRIBUTE_DFLT_ARG.

The order argument is used as in MPI_TYPE_SUBARRAY to specify the storage order for multidimensional arrays. Therefore, arrays described by this type constructor may be stored in row-major or column-major order. Valid values for order are MPI_FORTRAN_ORDER and MPI_C_ORDER.

For example, let us generate the filetypes corresponding to the HPF distribution:

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

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

```
    ndims = 3
    dsizes(1) = 100
    distribs(1) = MPI_DISTRIBUTE_CYCLIC
    dargs(1) = 10
    dsizes(2) = 200
    distribs(2) = MPI_DISTRIBUTE_NONE
    dargs(2) = 0
    dsizes(3) = 300
    distribs(3) = MPI_DISTRIBUTE_BLOCK
    dargs(3) = MPI_DISTRIBUTE_DFLT_ARG
    pdims = 2
    psizes(1) = 2
    psizes(2) = 3
    call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
    call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
    call MPI_TYPE_DARRAY(size, rank, ndims, dsizes, distribs, dargs,
   $     pdims, psizes, MPI_FORTRAN_ORDER, oldtype, newtype, ierr)
```

MPI_TYPE_DARRAY_BLOCK(size, rank, dsize, darg, oldtype, newtype)

IN	size	size of communicator used in MPI_Open (nonnegative integer)
IN	rank	rank in communicator used in MPI_Open (nonnegative integer)
IN	dsize	size of distributed array (nonnegative integer)
IN	darg	distribution argument, e.g., BLOCK(darg) (integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_darray_block(int size, int rank, int dsize, int darg,
             MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
MPI_TYPE_DARRAY_BLOCK(SIZE, RANK, DSIZE, DARG, OLDTYPE, NEWTYPE, IERROR)
    INTEGER SIZE, RANK, DSIZE, DARG, OLDTYPE, NEWTYPE, IERROR
```

MPI_TYPE_DARRAY_BLOCK generates the filetypes corresponding to the BLOCK distribution of a one-dimensional dsize element array of oldtype.

This call is a shorthand for:

```
MPI_Comm_size(comm, &size);
MPI_Comm_rank(comm, &rank);
distrib = MPI_DISTRIBUTE_BLOCK;
MPI_Type_darray(size, rank, 1, dsize, distrib, darg, 1, size,
                 order, oldtype, &newtype);
```

Note that order is not specified for this constructor, nor for MPI_TYPE_DARRAY_CYCLIC, because it is not meaningful for 1D arrays.

MPI_TYPE_DARRAY_CYCLIC(size, rank, dsize, darg, oldtype, newtype)

IN	size	size of communicator used in MPI_Open (nonnegative integer)
IN	rank	rank in communicator used in MPI_Open (nonnegative integer)
IN	dsize	size of distributed array (nonnegative integer)
IN	darg	distribution argument, e.g., CYCLIC(darg) (integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_darray_cyclic(int size, int rank, int dsize, int darg,
            MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
MPI_TYPE_DARRAY_CYCLIC(SIZE, RANK, DSIZE, DARG, OLDTYPE, NEWTYPE, IERROR)
    INTEGER SIZE, RANK, DSIZE, DARG, OLDTYPE, NEWTYPE, IERROR)
```

MPI_TYPE_DARRAY_CYCLIC generates the filetypes corresponding to the CYCLIC distribution of a one-dimensional dsize element array of oldtype.

This routine is a shorthand for:

```
MPI_Comm_size(comm, &size);
MPI_Comm_rank(comm, &rank);
distrib = MPI_DISTRIBUTE_CYCLIC;
MPI_Type_darray(size, rank, 1, dsize, distrib, darg, 1, size,
                 order, oldtype, &newtype);
```

10.8 I/O Error Classes

The implementation dependent error codes returned by the I/O routines can be converted into the following error classes:

- MPI_ERR_FILE - Invalid file handle

- MPI_ERR_NOT_SAME - Collective argument not identical on all processes, or collective routines called in a different order by different processes
- MPI_ERR_NO_SUCH_FILE - File does not exist
- MPI_ERR_FILE_EXISTS - File exists
- MPI_ERR_BAD_FILE - Invalid file name (e.g., path name too long)
- MPI_ERR_ACCESS - Permission denied
- MPI_ERR_NO_SPACE - Not enough space
- MPI_ERR_QUOTA - Quota exceeded
- MPI_ERR_READ_ONLY - Read-only file or file system
- MPI_ERR_EOF - End of file or end of view
- MPI_ERR_FILE_NOT_DELETED - File not deleted, currently open by some process
- MPI_ERR_IO - Other I/O error

10.9 Examples

10.9.1 Scatter / Gather Filetype Constructor Example

Suppose that we have a stream of telemetry data coming in containing a series of identically formatted blocks that will be analyzed in a round-robin sequence by several processes. To demonstrate a simple form of a general MPI datatype, consider a block of four integers, the third of which we want to disregard because the sensor has failed. To create the filetype for each process, the following FORTRAN code fragment could be used:

```
      integer telemetry_data(3)
      integer size, rank, count, telemetry_type, scatter_filetype, ierror
      integer array_of_blocklengths(2), array_of_displacements(2)

      count = 2
      array_of_blocklengths(1) = 2
      array_of_blocklengths(2) = 1
      array_of_displacements(1) = 0
      array_of_displacements(2) = 3

      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)

      call MPI_TYPE_INDEXED(count, array_of_blocklengths,
     $        array_of_displacements,
     $        MPI_INTEGER, telemetry_type, ierror)

      call MPI_TYPE_SCATTER_GATHER(size, rank, telemetry_type,
     $        scatter_filetype, ierror)
```

The equivalent C code would be:

```
MPI_Datatype telemetry_type, scatter_filetype;
int telemetry_data[3];
int size, rank, count;
int array_of_blocklengths[2], array_of_displacements[2];

count = 2;
array_of_blocklengths[0] = 2;
array_of_blocklengths[1] = 1;
array_of_displacements[0] = 0;
array_of_displacements[1] = 3;

MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements,
                 MPI_INTEGER, &telemetry_type);

MPI_Type_scatter_gather(size, rank, telemetry_type, &scatter_filetype);
```

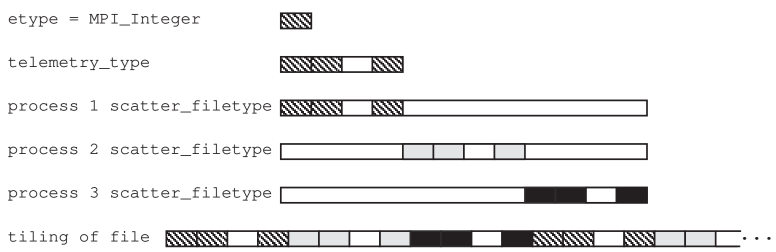

Figure 10.4: Example of Scatter Filetype for Three Processes

First the telemetry type is generated to represent the useful data in each block. Then the scatter filetype is generated using the scatter/gather constructor. The generated filetype scatters the data to the individual processes with holes for the discarded data and data used by other processes. Figure 10.4 shows the filetype generation for three processes.

10.9.2 Local Array Filetype Constructor Example

Assume we are writing out a 100x100 2D array of double precision floating point numbers that is distributed among 4 processes such that each process has a block of 25 columns (e.g., process 0 has columns 0-24, process 1 has columns 25-49, etc.; see Figure 10.5). To create the filetypes for each process one could use the following C program:

```
double subarray[100][25];
MPI_Datatype filetype;
int sizes[2], subsizes[2], starts[2];
int rank;
```

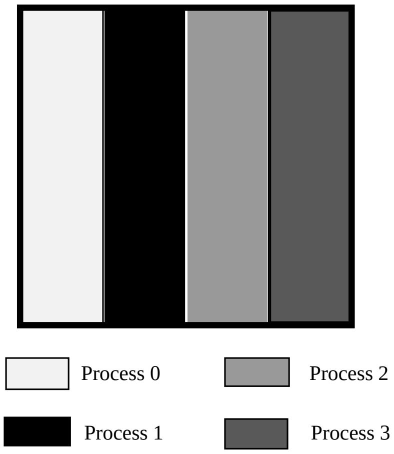

Figure 10.5: Example array file layout

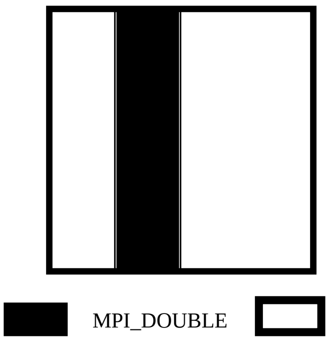

Figure 10.6: Example subarray filetype for process 1

```
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_subarray(2, sizes, subsizes, starts, MPI_C_ORDER,
                  MPI_DOUBLE, &filetype);
```

Or, equivalently in FORTRAN:

```
    double precision subarray(100,25)
    integer filetype, rank, ierror
    integer sizes(2), subsizes(2), starts(2)

    call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
    sizes(1)=100
    sizes(2)=100
    subsizes(1)=100
    subsizes(2)=25
    starts(1)=0
    starts(2)=rank*subsizes(2)

    call MPI_TYPE_SUBARRAY(2, sizes, subsizes, starts,
  $            MPI_FORTRAN_ORDER, MPI_DOUBLE_PRECISION,
  $            filetype, ierror)
```

The generated filetype will then describe the portion of the file contained within the process' subarray with holes for the space taken by the other processes. Figure 10.6 shows the filetype created for process 1.

Part III

MPI Journal of Development

Chapter 11

Real-Time MPI

Real-Time proposals here are work in progress. The real-time working group unanimously voted to move the chapter into the JOD to gain more time for preliminary implementations and further improvements since the work here standardize the area that does not have common existing practice. MPI Forum had not voted on any parts of this chapter.

11.1 Introduction

The goal of real-time MPI (MPI/RT) is to provide the middleware for programmers to create real-time applications with performance portability. MPI/RT is to provide a consistent set of extensions and, in some cases, restrictions of MPI. The goal of MPI/RT is to add greater predictability and schedulability to message-passing programming.

Portability in the MPI/RT sense is defined to mean:

- Ability to move a code from one platform to another, or from one configuration to another, and compile successfully without changing the syntax of the parallel program.
- The qualities of service achieved by the real-time application may not be guaranteed in a new configuration or port. By fine tuning the quality of service parameters, the program may be made to run in the new configuration or platform. These changes insure timing correctness, provided that sufficient resources are present to support the minimum application requirements.
- Upgrading from one generation of a platform to the next is generally expected to work seamlessly, even though resource utilization could be lower on the upgraded platform.

The design philosophy of MPI/RT is as follows:

- MPI/RT will *not* determine implementation policy, but will instead provide middleware to implement real-time paradigms such as:
  - Time-driven,
  - Event-driven,
  - Priority-driven,
  - Ad hoc, and
  - Resource constrained.

Discussion: We have omitted an important real-time paradigm in this document: dataflow. **Resolution:** We voted to omit the dataflow model from MPI/RT, because it can be accomplished using the other models we support.

- MPI/RT will make minimal changes to MPI, so that MPI/RT programs can benefit from existing MPI libraries, at least for non-critical parts of real-time applications,
- Efforts will be made to make existing MPI libraries work as seamlessly as possible within these real-time profiles.
- MPI/RT must allow not only code portability but also performance portability, insofar as possible.
- MPI/RT will not replace the native runtime system or scheduler, but will provide a portable means to communicate with these systems.

  Figure 11.1 show the model of software abstraction layers for MPI/RT.

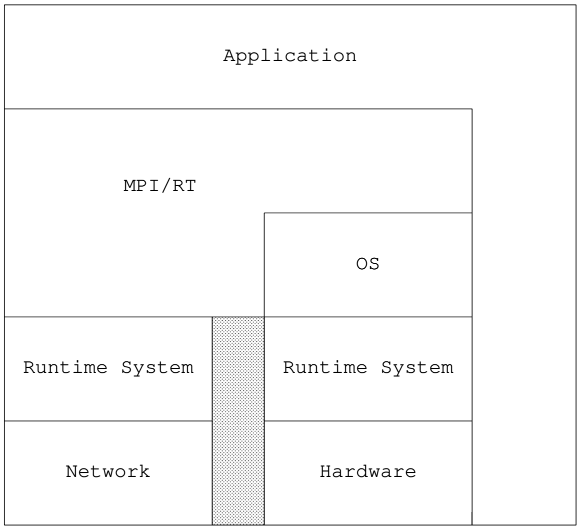

Figure 11.1: Model of Software Abstraction Layers for MPI/RT.

- For resource constrained users, the advantages of a layered approach indicate that "subset profiles" bear consideration as part of this effort. The smallest subsets should be minimal to allow the widest possible embedding of MPI with and without specific real-time features. A feature-driven rather than call-driven approach to such profiles is indicated. Some real-time, embedded kernels might choose to add the smallest set of features to the kernel, with layering of additional features for less constrained situations.
- The results of the real-time MPI effort will be a single set of real-time extensions, restrictions, and recommendations, suitable for realization as an MPI/RT implementation. It is expected that users will select operations in order to support the real-time paradigms of their choosing.
- The placement of this chapter in the Journal of Development (JOD) indicates that MPI/RT will be adopted separately from the MPI-2 but is loosely related to it wherever

possible. That is, not all MPI-2 implementations will need to support the real-time and resource constrained features described here.

Real-time MPI will serve as an open standard for a wide variety of high-performance, real-time, embedded, and heterogeneous parallel computing systems encompassing a diverse mix of computational paradigms. We intend that MPI/RT will make MPI relevant to the real-time community as well as enable development of vendor-independent real-time applications. It should also assist in bridging the gap between theory and implementation in parallel real-time computing and communications.

11.2 Real-time Message-Passing Requirements

Although developers have an intuitive sense of what they mean by a real-time system, definitions vary widely. The distinction between real-time computer systems and general purpose computer systems lies not in their performance specifications, but in the relative importance of the system timing considerations. In real-time computing, the correctness of a computation depends not only on the results of a computation, but also on the time at which the results of the computation are generated.

The measures of merit in a real-time system include:

- Timing correctness as well as program correctness.
- Deterministic operation – even at the cost of performance.
- Predictable response to urgent events.
- High degree of schedulability. *Schedulability* is defined as the degree of resource utilization at or below which the timing requirements of all tasks can be guaranteed [].
- Stability under transient load. When the system is overloaded by events and meeting all deadlines is impossible, the deadlines of selected critical tasks must still be guaranteed.
- Dependability – fault tolerance, traded off with the cost of performance.
- Portability, in the case of MPI/RT, with minimal impact on performance.

Timing correctness for a real-time system includes the following properties:

- A real-time program can proceed, in a meaningful way, only if all previous steps are time-correct.
- MPI/RT should have analogous functions to MPI that guarantee message passing in a timely fashion.

11.3 RT Profiles

Discussion: Added at the September 19 meeting.

We define several profiles of the real-time MPI. These profiles are independent of the programming paradigms that this chapter provides. All implementations are required to provide all profiles, though implementors will concentrate their efforts on the profile or profiles most relevant to their systems. This will provide for wide portability, and allow for appropriate investment in particular system niches.

> *Advice to implementors.* Such profiles will be handled at compile/link time. We have an expectation that "smart linking" will be used to minimize the overhead associated with unused functions. This is compatible with the MPI-1 expectation of profiling libraries, that essentially puts one function per source file in most implementations. (*End of advice to implementors.*)

The profiles to be recognized by MPI/RT are as follows:

- Resource constrained MPI 1.2, defined below,
- Resource constrained MPI 1.2, plus RT features described in this chapter,
- MPI 1.2 (classic MPI), plus RT features,
- MPI-2, plus RT features.

11.3.1 Resource Constrained MPI 1.2

This profile is to include all of the functionality of the revised MPI-1 reference document, with the following exceptions and restrictions:

- Derived datatypes are omitted, but built-in datatypes are preserved.
- Virtual topologies are eliminated.
- All buffered send modes are eliminated.
- Explicit operations with groups are eliminated.
- Users are recommended to assume zero buffering in writing their programs[1]

The expectation is that all resource constrained implementations will utilize smart linking to avoid unnecessary code bloat, and will take advantage of simplifications that arise from the restriction to built-in datatypes.

11.3.2 Other profiles

The profiles, defined above, are essentially self explanatory. The details of MPI-2 are to be revisited upon its completion. Furthermore, there is an expectation that the meaning of "RT features" will be broaden to include all relevant RT versions of the MPI-2 functionality rather than MPI-1 functionality only. For instance, I/O will require a real-time analog.

[1] MPI is not mandated to assume buffering, but most non-real-time implementations provide it. This is a strong advice to users and implementors of the resource constrained version to expect programs that use any buffering to deadlock or otherwise fail.

11.4 RT Initialization/Termination

Discussion: At the October meeting, it was agreed that further discussion is needed, on these points:

- In a mixed environment, with one "world," some processes may need RT features and others not
- It is desired to have MPI_INIT start all the MPI processes, and have some exploit real-time behavior according to the way they were linked and/or executed.
- Further arguments are needed about how to achieve the functionality of starting the real-time mode of operation, while allowing other processes to avoid expensive synchronization. It was pointed out that admission tests, with some processes uninvolved in real-time, might be specious at best.

11.5 Clocks

Discussion: It was stated at the October meeting that it will be difficult for implementations to provide appropriate bounds with respect to Accuracy and Access Time of clocks. Various caveats will be added to allow implementors to provide finite bounds, with some residual value to users.

11.5.1 Synchronization of Clocks

Most platforms that support real-time applications provide tightly synchronized system clocks, that are dependent upon special hardware support. There are several compelling reasons for having highly synchronized clocks [17]:

1. Fine-grained, accurate instrumentation is needed for all approaches to real-time message-passing systems, and even for performance measurements in non real-time systems.
2. Real-time applications require precise timing correctness. These systems also require demonstrations of this correctness, and often require delicate tuning for optimal performance. Well-synchronized clocks are necessary to support these requirements in a parallel environment.
3. Applications should be able to adjust scheduled times for portability, based upon the quality of the synchronized clocks. For example, time padding will be needed to adapt scheduled message-passing operations on heterogeneous processing nodes.
4. The primary goal of time-driven real-time MPI is to support application specification of resource usage. For time-driven MPI/RT, all resources that are used for communication need to be scheduled in order to achieve predictable behavior. These scheduled resources can include:
   (a) Distributed Memory,
   (b) Shared Memory,
   (c) Communication Bandwidth,

(d) Computation.

as well as others, such as platform specific resources related to inter-process communication.

11.5.2 Description of the Clocks

The processing nodes that comprise a real-time parallel processing system may have access to several clocks. We designate one of the clocks as a globally synchronized clock. For each process in an MPI/RT program, this clock will the be one read by the MPI_WTIME call. There is an underlying assumption that each process is associated with a fixed processing node (which may contain one or more CPUs). The process accesses its globally synchronized clock through its associated node.

For most platforms, each node has a local clock, and this is periodically corrected in order for it to serve as a synchronized clock for all MPI/RT processes active at that node. However, other alternatives are not precluded. For example, there may even be a single clock at one of the nodes or even completely outside of the participating nodes, which all the processing nodes access over a network (which may also be used for regular data transfer operations) to get a synchronized clock value.

A fundamental assumption is made that the system clocks will be monotonically non-decreasing. We also assume that the underlying operating system will hide any artifacts due to system clock counters overflowing.

11.5.3 Clock Synchronization Parameters

In this section we present several parameters that describe the synchronized clocks, all of which must be accessible at run time and compile time. The values of all these parameters are expected to be double precision floating-point values measured in seconds, with exception of the drift, which is dimensionless.

Resolution (Tick) Resolution represents the time between two successive clock ticks. The resolution of the various synchronized clocks in a heterogeneous system may differ.

Advice to implementors. We have proposed a subtle modification to MPI_WTICK, so that the call will return the resolution of the synchronized clock of the processing node associated with the calling process. In order to support tightly synchronized system clocks, we expect clock resolution to be at most one millisecond. (*End of advice to implementors.*)

Drift Drift indicates, for each synchronized clock, a guaranteed upper bound on the error in the rate of the clock. That is, if the drift is δ, then the clock rate (measured in seconds per actually elapsed second) is guaranteed to be between $1 - \delta$ and $1 + \delta$. Drift is, as stated above, dimensionless.

Drift can be effectively used to bound the accuracy of measurement of small time intervals, when they are measured by the difference between two readings of the same synchronized clock.

Advice to implementors. An implementation of MPI/RT should reset the drift parameter when global system clocks are synchronized. (*End of advice to implementors.*)

Skew Skew is a maximum bound on the absolute value of the difference between simultaneous values of the synchronized clocks in distinct nodes. Note that this refers to ideal values, not the result of any real reading operations.

Accuracy Accuracy is a maximum bound on the absolute value of the difference between simultaneous values of the synchronized clock and an ideal clock started at the synchronized clock hypothetical starting time (some critical instant in time).

If synchronized clocks are periodically corrected in order to deal with drifts and other inaccuracies, the calculation of the interval accuracy based upon drift will be too pessimistic for large intervals.

This value can also be used for cross-platform synchronization.

Access Time Access Time is a maximum bound on the time to execute a call of MPI_WTIME, This, of course, assumes that the execution of the call is not interrupted by the operating system. This undesirable and ambiguous caveat is necessary with the current operating systems (especially those with virtual memory) because an absolute guarantee would be so long as to be practically unusable.

Implementations that are based on accessing the same global clock instead of a local synchronized one will have the same resolution for all nodes, with zero skew, no drift, and identical accuracy (subject to the propagation delay of the clock signal) for all processing nodes. However, the access times could still vary at each processing node and across all the processing nodes. A bound on this variability is needed.

11.5.4 MPI/RT Clock Attributes

In order for MPI/RT to allow portable applications across a wide variety of parallel real-time systems, it is necessary to encapsulate the clock synchronization parameters of section 11.5.3 in a system independent way. MPI provides a caching facility that allows an application to attach arbitrary pieces of information, called **attributes**, to both intra- and intercommunicators. This information is retrieved by referencing a *key*. A set of attributes that describe the execution environment is attached to the communicator MPI_COMM_WORLD when MPI is initialized. The value of these attributes can be inquired by using the function MPI_ATTR_GET. At initialization time, MPI/RT adds the following keys to MPI_COMM_WORLD:

- MPI_WTIME_DRIFT (DOUBLE)
- MPI_WTIME_SKEW (DOUBLE)
- MPI_WTIME_ACCURACY (DOUBLE)
- MPI_WTIME_ACCESS_TIME (DOUBLE)

Discussion: Should the format of these parameters match the Posix real-time specification?

The values of the listed parameters do not depend on what applications are running but rather on the parallel environment. These parameters represent constraints on the changes to the environment. For example, the skew should not be increased when new processes are added while an application is running.

Discussion: That suggests that dynamic process management ideas are impacted.

11.6 Real-time Buffer Pools

This section proposes an interface for associating a list of buffers with a real-time channel.

11.6.1 Buffer Pool Object Creation and Destruction

Buffer pools are created via single constructor that specifies their entire contents. The participating buffers can not be modified after creation. The buffer pool handle is used in the creation of a real-time channel, following a specific constructor (`MPIRT_CHANNELS_BUFPOOL_INIT`).

All the buffers in a buffer pool are the same length and contain the same datatype elements. The datatypes are relative, while addresses for individual buffers are absolute. The buffer pool can be shared between several channels but all channels must use the same queuing strategy.

MPIRT_BUFFER_POOL_CREATE(count, datatype, system_queue_strategy, bufcount, bases, bufpool)

IN	count	the number of elements of datatype in a buffer (nonnegativeinteger)
IN	datatype	datatype of each element (handle)
IN	system_queue_strategy	System's designated approach to managing the buffer pool when it is used in an association with a channel (handle).
IN	bufcount	the number of buffers in the buffer pool (nonnegative integer)
INOUT	bases	array of length `bufcount` with the beginning addresses of each buffer in the pool (choice)
OUT	bufpool	buffer pool object (handle)

The `system_queue_strategy` argument indicates the desired buffer management strategy.

Discussion: Two specific values are currently under consideration: `MPI_BUFFER_CIRCULAR_WAIT` and `MPI_BUFFER_CIRCULAR_NOWAIT`. This indicates that the system should traverse the buffers in a circular order when trying to locate an available buffer.

With each buffer we identify a unique index. When the MPI creates all buffers it creates an index between 0 and `bufcount-1` for each buffer. These numbers are assigned in accordance with the order in the array `bases` above.

The following function destroys a buffer pool handle:

MPIRT_BUFFER_POOL_HANDLE_FREE(bufpool)

INOUT	bufpool	buffer pool object (handle)

```
int MPIRT_Buffers_free(MPIRT_Buffers* buffers)
```

```
MPIRT_BUFFERS_FREE(BUFFERS, IERR)
    INTEGER BUFFERS, IERR
```

11.6.2 User Buffer Access

In considering the consumption and release of buffers as a channel's state evolves under message transmission, both the system's strategy, and the user's emphasis on message age come into play. The desired system strategy for handling the buffer pools upon creation has been specified by MPIRT_BUFFER_POOL_CREATE. There is also the need for an application to specify "newer," "older" or "intermediate" age data, as a function of the semantics of the meaning of such data to the application.

Buffers in the buffer pool are either available or used. Buffers become used as the system consumes them. Used buffers can be made available to the system again by the following function:

MPIRT_BUFFER_AVAIL(index, bufpool)

IN	index	buffer index (integer)
INOUT	bufpool	buffer pool object (handle)

```
int MPIRT_Buffers_avail(int index, MPIRT_Buffers* buffers)
```

```
MPIRT_BUFFERS_AVAIL(INDEX, BUFFERS, IERR) INTEGER INDEX, BUFFERS, IERR
```

The index value MPIRT_ALL_BUFFER indicates that all buffers in the buffer pool available for reuse by MPI.

The following operation allows the user to get access to the buffers:

MPIRT_BUFFER_GET(bufpool,user_strategy,count,index,channel)

IN	bufpool	buffer pool object (handle)
IN	user_strategy	user strategy for buffer management (handle)
IN	count	count of the buffer element (integer)
OUT	index	the index of the buffer (integer)
OUT	channel	the name of the channel on which the message arrived (request)

Three specific values of the MPIRT_USER_STRATEGY are currently defined: MPI_BUFFER_NEWEST, MPI_BUFFER_OLDEST, and MPI_BUFFER_NEXTAVAIL. Other options may be added. For example MPIRT_BUFFER_GET(bufpool,NEWEST,1,index,channel) returns the index of the buffer with the last received message. To obtain a buffer suitable for use on the sending sideMPI_BUFFER_NEXTAVAIL has been provided.

Discussion: A frozen user view of the buffer queue may be required for some applications (atomic access to the buffer pool queue). For example user may need to access three consecutive receive messages. Simple repetition of MPIRT_BUFFER_GET(bufpool,NEWEST,1,index,channel) is not sufficient, since new messages may arrive between these three operations and the queue will be modified. Hence, the following operations are proposed: MPIRT_QUEUE_LOCK, MPIRT_QUEUE_UNLOCK. An implementation can have a shadow queue so it can continue to receive and send messages, while maintaining the frozen buffer queue snapshot for the user. Since additional resources may be required, a flag can be added to the API for the buffer pool at creation time to notify implementation. This flag provides information for establishing QOS for a channel.

Discussion: This is an old example that will be modified later.

C example:

```
MPIRT_Request  request;
MPIRT_Status   status;
MPIRT_Buffers  buffers;
int            data0[1024], data1[1024], data2[1024], data3[1024];
int           *new_data;
...

MPIRT_Buffers_create( &buffers );
MPIRT_Buffer_add( data0, 1024, MPI_INT, data0, &buffers);
MPIRT_Buffer_add( data1, 1024, MPI_INT, data1, &buffers);
MPIRT_Buffer_add( data2, 1024, MPI_INT, data2, &buffers);
MPIRT_Buffer_add( data3, 1024, MPI_INT, data3, &buffers);
MPIRT_Buffers_manage( MPIRT_BUFFERS_CIRCULAR, &buffers );
...

MPIRT_Buffers_wait( MPIRT_BUFFERS_NEWEST, &buffers, &request, &status );
new_data = (int *)status.MPI_TAG;
Update_Data( new_data );
...
```

FORTRAN example:

```
      INTEGER index
      INTEGER request, status,buffers
      INTEGER data(1024,4)
      INTEGER ierr
      INTEGER new_index

      call MPIRT_Buffers_create( buffers, ierr )
      do index=1,4
         call MPIRT_Buffers_add( data(1,index), 1024, MPI_INTEGER, index,
     &                           buffers, ierr )
```

```
        enddo
        call MPIRT_Buffers_manage( MPIRT_BUFFERS_CIRCULAR, buffers, ierr )
...

        MPIRT_Buffers_wait( MPIRT_BUFFERS_NEWEST, buffers, request, status, ierr )
        new_index = status(MPI_TAG)
        call Update_Data( data( 1, new_index ) )
..
```

11.7 Channels

Persistent communications can provide scheduled sends and receives. MPI persistent communications allow any receive to match a persistent send, and any send to match a persistent receive []. These persistent communication calls operate as prenegotiated communication endpoints.

In MPI/RT, persistent channels offer the functionality of a virtual channel within the framework of the MPIstandard. The motivations for having virtual channels in MPI/RT are as follows:

- Ability to exploit persistent communication
- Deadlock avoidance
- Once established, virtual channels should guarantee properties critical for timing correctness such as:
  - Bounds on end-to-end delay
  - Jitter control
  - Minimum bandwidth
  - Buffer space

11.7.1 Point-to-point Channels

Discussion: *Proposal* It was proposed at the last meeting that the channel specification in the RT chapter be revised. During the RT subcommittee meeting a motion was made to eliminate the non-bufpool variants of the channel creation and modification routines. That implies that all of the standard buffer versions be deleted and that the bufpool versions be renamed by removing the word "BUFPOOL" from the name. (e.g., the current `MPIRT_CHANNELS_INIT()` would be deleted, and `MPIRT_CHANNELS_BUFPOOL_INIT()` would be renamed to `MPIRT_CHANNELS_INIT()`)

The buffer pool section states that buffers in the buffer pool can not be modified once the buffer pool is associated with a channel. However, the current functionality in this section still provides channel modification functions to change the buffers in the buffer pool. This problem will be fixed by deleting `MPIRT_CHANNELS_MODIFY` and by only allowing one buffer pool to replace another buffer pool.

As part of this proposal, it was recommend that MPI/RT will provide no other buffering for a channel other than that explicitly provided by the user via the bufpool. (For example, if all of the receive buffers have been exhausted, any further requests at the send side will not complete until one of the receive buffers is consumed.) By eliminating extra buffering in an implementation, real-time constraints can be met; especially the needs of resource constrained systems.

This proposal will be voted on at the next meeting in December in Boston.

Collective operations for point-to-point channels negotiation are as follows:

Discussion: The buffer version of the MPIRT_CHANNELS_INIT may be removed after the vote at the next meeting (see above proposal).

MPIRT_CHANNELS_INIT(bufs, counts, types, nchannels, flags, qoss, fns, comm, requests, errors)

IN	bufs	Array of send or receive buffers for channels local to process (choice)
IN	counts	integer array (of length group size) specifying the number of elements to transfer for each channel
IN	types	data type of transfer buffer for each channel (handle).
IN	nchannels	Length of all arrays used locally (number of channels being opened)
IN	flags	array of length `nchannels`, with 1,-1 values, indicating endedness of ith channel
IN	ranks	array of length `nchannels`, indicating the ranks of the remote channel endpoints
INOUT	qoss	array of quality of service parameters per channel (integer array of length `nchannels`).
IN	fns	array of function names for handlers on channels
IN	comm	communicator (handle).
OUT	requests	request objects, one per channel (persistent request)
OUT	errors	specifies success, non-success, for each channel (int array of length `nchannels`)

```
int MPI_Channels_init(void **bufs, int *counts, MPI_Datatypes *types, int
             nchannels, int *flags, int *ranks, int *qoss, MPI_Comm comm,
             MPI_Request *requests, int *errors)
```

```
MPI_CHANNELS_INIT(BUFS, COUNTS, TYPES, NCHANNELS, FLAGS, RANKS, QOSS, COMM,
             REQUESTS, ERRORS, IERR)
    INTEGER BUFS(*), NCHANNELS, COUNTS(*), FLAGS(*), RANKS(*), QOSS(*),
    COMM, REQUESTS(*), ERRORS(*), IERR
```

The alternative form, which makes use of the buffer pool strategy, is as follows:

Discussion: The buffer pool version of the MPIRT_CHANNELS_BUFPOOL_INIT may be renamed to MPIRT_CHANNELS_INIT after the vote at the next meeting (see above proposal).

MPIRT_CHANNELS_BUFPOOL_INIT(bufpools, nchannels, flags, qoss, fns, comm, requests, errors)

IN	bufpools	array of transfer buffer pool addresses (handle)
IN	nchannels	Length of all arrays used locally (number of channels being opened)
IN	flags	array of rank length, with 1,-1 values, indicating endedness of channels
INOUT	qoss	array of quality of service parameters per channel (handle).
IN	fns	array of function names to be used as handlers for errors on channels
IN	comm	communicator (handle).
OUT	requests	request objects, one per channel (persistent request)
OUT	errors	specifies success, non-success, for each channel (int array)

This operation yields as many channels as specified by the user, with the potential for duplicate ranks both for sending and receiving as well as multiple channels involving the same pair.

Discussion: The disambiguation of multiple channels involving a pair of the same processes must be defined.

Each process specifies the sender (1), receiver (-1), in the flags array.

These individual requests may be used to transmit point-to-point data between processes, independently. The call is collective in order to establish required quality of service in the face of shared resources. This call does not initiate any communication, but only sets up such communication.

The quality of service parameters (per channel) include specification of hard vs. best-effort requirement for meeting the QOS request made. By hard, we mean that if the QOS cannot be offered exactly as specified, then an error is returned, and the channel is not established. Best effort constitutes the nearest level of service a system can provide. It is less than the application requested as specified in the QOS In-parameter, and is returned in the QOS as an Out-parameter.

Both ends of each channel must specify the same quality of service requirements.

The error function handler array specifies functions that are called in case of handler errors. These are needed to support one-sided and no-sided communication uses of the channels, as there is no MPI call to bind a function to an operation on a per-use basis. This is especially critical for the periodic uses. We may create more than one error handle for a channel. One is needed to handle timeout situations, especially for no-sided communication. More maybe needed to handle other types, like buffer pool overflow, communicator errors and others, that some applications may want to handle prior to a timeout expiration. The error condition array specifies `MPI_SUCCESS` for successful channels, and offers specific error codes, to be determined, for non-successful channel creations.

Discussion: *Proposal:* The complicated mappings between error codes and error handlers should be addressed in a future. Masking errors, providing mechanisms to define and find out about "fatal" errors should be address at the same time. This is still based on assumption that most of errors for this section, including, communicators, buffer pools, QoS and so on, will be associated with the channels.

Advice to users. The same buffer pool can be used for more than one channel. While for some paradigms user can easily manage to avoid problems with the shared buffers, in general it is a very dangerous scenario that can lead to various errors and create problem for maintaining QoS for the channels that share the buffer pool. (*End of advice to users.*)

Discussion: *Proposal:* It is generally agreed that we should modify this function to build up to $N(N-1)$ channels (both directions), rather than require two calls.

MPIRT_CHANNELS_DELETE(comm, flag, nchannels, requests)

IN	comm	communicator (handle).
IN	flag	flags abrupt versus "close" type channel semantics
IN	nchannels	number of channels to be deleted
IN	requests	request objects, one per channel (persistent request)

This operation frees the specified requests previously allocated, and is a collective operation over the group of the communicator. This may be all the channels allocated on a single or multiple calls, or just a subset thereof, according to user preference.

Any pending operations are cancelled, if the flag is set to MPIRT_DELETE, and are completed, if the flag is set to MPIRT_CLOSE.

In order to allow channels to late bind their behavior, a modification capability is offered:

Discussion: *Proposal:* The buffer version of the `MPIRT_CHANNELS_MODIFY` may be removed after the vote at the next meeting (see above proposal).

MPIRT_CHANNELS_MODIFY(bufs, counts, types, nchannels, flags, qoss, fns, comm, requests, errors)

IN	bufs	array of buffers
IN	counts	integer array (of length group size) specifying the number of elements to transfer for each channel
IN	types	data type of transfer buffer for each channel (handle).
IN	nchannels	number of channels to be modified (integer)
IN	flags	if MPIRT_FLAGS_IGNORE, channel is left unchanged, otherwise if MPIRT_MODIFY, then specific parameters are renegotiated for this channel.
INOUT	qoss	array of quality of service parameters per channel (handle).
IN	fns	Error handler functions
IN	comm	communicator (handle).
INOUT	requests	request objects, one per channel (persistent request)
OUT	errors	specifies success, non-success, for each channel (int array)

Discussion: The buffer pool version of the `MPIRT_CHANNELS_BUFPOOL_MODIFY` may be renamed to `MPIRT_CHANNELS_MODIFY` after the vote at the next meeting (see above proposal).

MPIRT_CHANNELS_BUFPOOL_MODIFY(bufpools, nchannels, flags, qoss, fns, comm, requests, errors)

IN	bufpools	array of transfer buffer pool handles (handle)
IN	nchannels	number of channels to be modified (integer)
IN	flags	if MPIRT_FLAGS_IGNORE, channel is left unchanged, otherwise if MPIRT_MODIFY, then specific parameters are renegotiated for this channel.
INOUT	qoss	array of quality of service parameters per channel (handle).
IN	fns	Error handler functions
IN	comm	communicator (handle).
INOUT	requests	request objects, one per channel (persistent request)
OUT	errors	specifies success, non-success, for each channel (int array)

When a falg is set to MPIRT_MODIFY the remaining arguments become significant. On both ends of a channel, the flags must be set consistently. For each channel to be renegotiated, each parameter is checked. A special value of "ignore" will be defined for each of these, to keep the currently set value. If not "ignored," then the quantity will be modified:

- bufpool: MPIRT_BUFPOOL_IGNORE,
- qos parameter: MPIRT_QOS_IGNORE,
- error fn parameter: MPIRT_HANDLER_IGNORE,

If an error occurs on a channel renegotiation, the original channel setup continues to persist. Setting channels to have negligible usefulness of quality of service does not in any event cause a channel to disappear.

Discussion: *Proposal:* There is no easy way to change any parameters of a buffer pool. We can only disconnect the old buffer pool with a channel and connect a new one. Maybe we should add another modify operation, to modify buffers in the existing buffer pool, while they are attached to a channel.

Single Channel Operations

Discussion: *Proposal:* The buffer pool version of the `MPIRT_CHANNEL_MODIFY` may be deleted after the vote at the next meeting (see above proposal).

A single channel may be modified as follows, with the strict requirement that only the end-point processes call this operation. It is not collective over the entire group of the communicator (binary-collective):

MPIRT_CHANNEL_MODIFY(buf, count, type, qos, fn, comm, request)

IN	buf	new buffer (choice)
IN	count	new integer count or "ignore"
IN	type	new data type or "ignore" (handle).
INOUT	qos	new quality of service parameter or "ignore" (handle).
IN	fn	error handler function
IN	comm	communicator (handle).
INOUT	request	modified request object (persistent request)

Discussion: *Proposal:* The buffer pool version of the `MPIRT_CHANNEL_BUFPOOL_MODIFY` may be renamed to `MPIRT_CHANNEL_MODIFY` after the vote at the next meeting (see above proposal).

MPIRT_CHANNEL_BUFPOOL_MODIFY(bufpool, qos, fn, comm, request)

IN	bufpool	new buffer pool handle or "ignore" (handle)
INOUT	qos	new quality of service parameter or "ignore" (handle).
IN	fn	Error handler function
IN	comm	communicator (handle).
INOUT	request	modified request object (persistent request)

For an individual channel, one may only modify in the sense of reducing the quality of service demanded, not by increasing it.

Discussion: When sets of channels are modified one at a time, there is a potential for deadlock. The calls must be made consistently across sets of processes, to avoid this potential.

Furthermore, a single channel may be deleted as follows, with the strict requirement that only the end-point processes call this operation. It is not collective over the entire group of the communicator (binary-collective):

Discussion: *Proposal:* This operation maybe removed provided that the existing MPI-2 channel deletion is sufficient. This functionality is currently in the collective chapter in the JOD.

MPIRT_CHANNEL_DELETE(comm, flag, request)

IN	comm	communicator (handle).
IN	flag	"abrupt" vs. "close" semantics (int)
INOUT	request	modified request object (persistent request)

This functionality is a superset of what MPI_REQUEST_FREE would do. As above, the flag specifies either MPIRT_DELETE or MPIRT_CLOSE.

Discussion: When sets of channels are deleted one at a time, there is a potential for deadlock. The calls must be made consistently across sets of processes, to avoid this potential.

11.7.2 Collective Operations with Quality of Service

For each operation in MPI-2 and collective operations in the journal of development (future MPI-3), we have defined a persistent variant, that is initiated with asynchronous (MPI_START) and synchronous (MPI_DO) mechanisms. For each such operation, the real-time variant is specified as follows: A quality of service specification, is added as an additional parameter, directly before the communicator parameter. Currently, we have both the MPI-1 and MPI-2 style functionality (illustrated for broadcast):

Discussion: *Proposal:* The non-buffer pool version of the collective channel initialization and modification may be added after the vote at the next meeting (see above proposal).

```
MPI_BCAST(bufpool, root, comm);
MPI_BCAST_INIT(bufpool, root, comm, request);
```

For each collective operation MPI/RT provides an additional form.

```
MPIRT_BCAST_INIT(bufpool, root, qos, comm, request);
```

The quality of service parameter applies to the collective operation. The hard vs. best effort nature of the quality of service specification is as for point-to-point channels. The specification of quality of service may differ from that used for point-to-point channels, and may also differ for various collective operations. For modification (late binding), the following type of service is to be provided for each collective operation, illustrated here for broadcast:

```
MPIRT_BCAST_MODIFY(bufpool, root, qos, comm, request);
```

Each parameter will either be "ignorable" or specify a new value. The modify is itself a collective operation over the group of the communicator. Quality of services may be either increased or decreased. For deletion of a collective persistent operation in the Collective Extensions chapter, it is assumed that MPI_REQUEST_FREE is sufficient. Here, we add a specific collective destructor:

Discussion: *Proposal:* This operation may be removed provided that the existing MPI-2 channel deletion is sufficient. This functionality is currently in the collective chapter in the JOD.

MPIRT_CHANNEL_COLL_DELETE(comm, flag, request)

IN	comm	communicator (handle).
IN	flag	flags abrupt versus "close" type channel semantics
IN	request	request object representing real-time collective operation

This operation frees all request previously allocated, and is a collective operation over the group of the communicator. Any pending operations are cancelled, if the flag is set to MPIRT_DELETE, and are completed, if the flag is set to MPIRT_CLOSE.

Discussion: An equivalent, alternative model is to have two calls, and no flag.

Advice to implementors. If real-time collective operations are layered on top of point-to-point channels, then a set of channels used to create a collective operation could be built up with the point-to-point channels, and deleted en masse using the point-to-point channel deletion defined above.

However, the quality of service parameter to the collective operation, as specified by the user, will have to be translated to appropriate channel quality of service parameters by the implementation. The implementation is also free to select one of several algorithms (poly-algorithm) to accomplish a given operation. (*End of advice to implementors.*)

11.8 Quality of Service

The quality of service parameters span point-to-point persistent channel, collective persistent channels and all paradigms of real-time discussed in this chapter. The QOS parameters are the most important distinction between regular MPI and real-time MPI.

It was agreed that only paradigm-specific QOS parameters would be part of MPI/RT for point-to-point channels as well as for collective channels, and that any lower-level QOS parameters would only be discussed as advice to implementors. See Appendix.

Discussion: Some of the lower-level quality of service may be revisited for multiple real-time paradigms mixed in the same application, but that is beyond the current scope of this draft. This will also recur when we consider implementation interoperability.

For three paradigms, here are the current proposed contents of the quality of service parameters:

Time-Driven The QOS parameters for time-driven paradigm include:

- time interval for message transmission,
- period, for periodic message transmission, including the relationship between the time interval and the period, usually defined by a release time and a deadline,
- Starting time within the period (transmission need not start at the beginning of the period, or end within the period).

Event-Driven The current expected QOS for event-driven paradigm is the bound required on the activation time of an event handler for a delivered event.

Low-level event-driven paradigm will consider local event bounds, whereas high-level event-driven will consider "global" event bounds.

Priority-Driven The QOS parameters for priority driven paradigm include:

- Preemptability flag (yes or no),
- Integer Priority of the persistent channel for message transmission (integer),
- Priority class of the persistent channel (integer) (provides second level of priority)
- Some measure of the preemption quantum desired in bytes (allows a user to calculate timing bounds for guaranteed delivery)

Currently, there are no deadline associated with the message transmission for the priority-driven or event-driven paradigms.

Note that the aggregate size of the transmission is needed as part of quality of service, but this is provided separately through the buffer, datatype, and count parameters of the standard form of the channel initialization, and as part of the buffer objects.

Discussion: It was agreed that it is appropriate for an application to specify event-driven quality of service with either time-driven or priority-driven quality of service parameters but not both. It is quite often the case that an application uses event-driven with other paradigms.

It is still under discussion what is the meaning of the mixture of time-driven and priority-driven paradigms with their associate quality of services.

Discussion: *Proposal:* For some applications the hard real-time guarantees, sometime at the expense of the performance, are too restrictive. The working group voted to add a QOS version that does not require hard guarantees. The details of the "softer" quality are still under discussion.

11.9 Time-driven MPI/RT

The primary goal of the time-driven approach to MPI/RT is to allow the real-time application sufficient control of the environment in which it is running so that it can explicitly schedule its message-passing activities and resource usage. Since MPI is designed as a message-passing library, it can not schedule by itself, but must depend upon the operating system and communication and network protocols to enforce specified schedules.

An application using time-driven MPI/RT will be able to specify time intervals to bound the resource usage of communication operations using globally synchronized clock values, and the implementation of time-driven MPI/RT will fulfill these requirements with minimal changes to the MPI standard.

For practicality, the work of our efforts has been to minimize the number of new functions introduced into MPI/RT profiles to provide these real-time features. MPI supports both blocking and non-blocking sends and receives, as well buffered, synchronous, and ready versions of each. We decided it was important not add new versions of each

send and receive mode, increasing the number of MPI calls multiplicatively. Rather, we sought solutions that would involve additive additions in the number of MPI functions.

11.9.1 Scheduling Message Transfers

The existing MPI message transfer operations lack two parameters which we consider critical for real-time applications, especially for the time-driven programming paradigm. These are a starting time of the operation and a timeout for completion of the operation. The starting time of an operation should be considered as a special case of an event. While certain applications (especially embedded ones) prefer an even finer granularity of control, we tried to strike a balance between the feasibility of an implementation and what time-driven application designers want to use. For example, there is a hard lower bound for the starting time, but no hard upper bound on the starting time.

One distinctive characteristic of the time-driven approach to real-time message-passing is its lack of need for queues and system buffers. Applications use *ready mode* message-passing implicitly. A ready-mode send may be started *only* if the matching receive has already been posted []. On many systems, this allows the removal of a hand-shake operation and results in improved performance. Since a parallel time-driven program must globally schedule all message transmissions, the message receiver always knows to expect an incoming message. Thus, for reasons of efficiency and simplicity, a time-driven MPI/RT implementation should not do any handshaking (as many of the existing implementations do). It is up to the application to specify times (for start and timeout) to ensure that the sender/receiver (local/remote) pairs are working in synchrony.

Another distinctive feature is a potentially more efficient way of using notifications, which can be more minimal than with other approaches. A time-driven MPI/RT application does not need to be notified when a message is transmitted successfully and on time; instead it is notified only when an error occurs (e.g., a timeout expires). A matter of significant discussion in the MPI/RT group concerns precisely what should happen to messages left on the network when a timeout expires.

11.9.2 Schedulable Time Intervals

An activity interval, specified by a starting time and a timeout, is an input parameter for a scheduled message send. The purpose of this parameter is to ensure that the system resources required to satisfy this operation will not be used outside of specified interval. These resources can be narrowly interpreted to refer to the interprocess communications network. A broader interpretation would include memory accesses, node busses, network interface cards, and so on. Again, while we prefer a finer granularity of control, we have tried to strike a balance between the feasibility of an implementation and what time-driven schedule designers want to use.

The starting time and timeout are somewhat symmetric. The starting time ensures that the resources needed for a data transfer operation will be available at the specified start time. The timeout parameter, in contrast, would ideally specify the time when all resources used by the message transfer operation are no longer in use. That is, after the time specified in the timeout, irrespective of whether the operation completed successfully or not, all system resources (physical network, network interface cards, node buses, message buffers, etc.) have been released and can be used for subsequent message-passing operations.

Unfortunately, in practice these guarantees often cannot be met. The MIP/RT timeout therefore specifies that the message transfer should be stopped and the calling application should be notified if the operation has not completed by the timeout time. Since the message may be progressing through a multi-stage network, a time-driven MPI/RT implementation may need to send a message from the receiver node to the sender to indicate that the timeout has occurred. The resulting error messages may not be received by the timeout deadline, and they may use resources after the timeout. Thus the application may need to reserve resources to handle such events. It should not be the responsibility of the MPI/RT implementation to provide this bound, since any guarantees which can be given from the perspective of a user-level message-passing library would be too naive to be useful. The application itself is in a much better position to know timing and performance details relevant to establishing such a bound, including details of the platform and knowledge of the run-time patterns of communication. Even for the application, it may be extremely difficult to establish such bounds, especially if the real-time performance characteristics of the operating system or the underlying runtime system are poorly known or highly variable.

11.9.3 MPI/RT Time Handle

The starting times and timeouts of the activity interval in time-driven MPI data transfer operation calls are specified by a structure called a TIME_OBJECT(one instance of the structure is used for each). A TIME_OBJECT has two fields, TIME_OBJECT_TYPE and TIME_OBJECT_TIME. TIME_OBJECT_TYPE must have one of two values, ABSOLUTE, or RELATIVE. When a starting time is replaced by MPI_TIME_IGNORE, then there is no hard constraint on when the operation should start. Implicitly, it should start as soon as possible, just as with the current MPI calls. Similarly, when a timeout is given by a MPI_TIME_IGNORE, there is to be no hard constraint on when the operation should end. Furthermore, the second field of a TIME_OBJECT is not significant if the first field is set to MPI_TIME_IGNORE.

For a TIME_OBJECT whose first field is ABSOLUTE or RELATIVE, the second field

(TIME_OBJECT_TIME) should be a field containing a double precision floating point number. In either case (ABSOLUTE or RELATIVE) the TIME_OBJECT_TIME refers to the global synchronized clock, but in the relative case, an actual constraint is to be derived at run-time by adding TIME_OBJECT_TIME to the value returned by as fresh as possible a read of the global synchronized clock. Note that even in the ABSOLUTE form, an actual time requirement cannot necessarily be constructed until the value of the time object at the time of the execution of the call is known.

Thus, the activity interval for time-driven MPI/RT message transfers is specified by two parameters: starting time and timeout, each specified by a time handle of the form TIME_OBJECT.

When using a late binding call, MPI_TIME_NOOVERRIDE should be used to retain built-in QOS specification, whereas MPI_TIME_IGNORE would replace the specified times with an "ignore," thereby making the call work as if no time interval were required.

In C/C++, the TIME_OBJECT is semi-opaque, like the MPI_STATUS. For Fortran, a strategy will be worked out similar to the MPI_STATUS solution as well.

11.9.4 Time-Driven Channel Calls

Using persistent channels to schedule a time-driven message transaction involves the addition of an MPI_Timeobj parameter to the MPI_START() call, which activates a persistent communications handle. MPI_START, MPI_STARTALL and MPI_WAIT/TEST operate as with other non-blocking operations to convert a request into a status and mark as complete the operation.

MPI_START_TIME(request, index, start, timeout, period, f)

IN	request	primitive request object (persistent request)
IN	index	index of the buffer for data transfer from the buffer pool associated with the channel (integer)
IN	start	message timing parameter (TIME_OBJECT handle)
IN	timeout	message timing parameter (TIME_OBJECT handle
IN	period	optional periodic reinvocation (TIME_OBJECT handle)
IN	f	void function to call on failure

The meanings of start and timeout are defined above. The meaning of period is either: non-periodic, if MPI_TIME_IGNORE is specified, or else the length of the period between automatic restarts of the request specified. When period is specified, both start and timeout are taken as relative to multiples of the period.

As an example of this approach, the following code schedules a send at a particular time:

Discussion: The QOS bindings used in the example are not defined in this draft. No conclusions regarding QOS bindings and their parameters should be drawn from this example

```
MPI_Timeobj start, stop;
MPI_Timeobj period = MPI_TIME_IGNORE; // not periodic here!
int counts[1];
MPI_Datatype types[1];
MPI_QOS qoss[1];
MPI_Aint bufs[1];
MPI_Request requests[1];
MPI_Buffer_Strategy system_strategy = MPI_BUFFER_CIRCULAR_NOWAIT;
MPI_Buffer_Strategy user_strategy_receive = MPI_BUFFER_NEWEST;
MPI_Buffer_Strategy user_strategy_send = MPI_BUFFER_NEXTAVAIL;
int flags[1];
extern void f();
void error_fns[1];
MPI_Buffer_Pool bufpool_handles[1];
MPI_Error errors[1];
MPI_Request channel;
int index;

// ...
MPI_Comm comm; // assume a communicator with exactly (2 processes)
// ...
int rank;
MPI_Comm_rank(comm, &rank);

// time in nanoseconds:
start.type =  MPI_RELATIVE;
start.value = 3.0e6;
stop.type   = MPI_RELATIVE;
stop.value  = 5.0e5;

count = 100;

   qoss[0] = MPI_QOS_IGNORE;
   // use QOS constructor:
   MPI_QOS_time(start, stop, period, &qoss[0]);
   bufs[0] = (int *)malloc(count*sizeof(int));
   types[0] = MPI_INT;
   counts[0] = count;
   error_fns[0] = f;

if (rank == 0)
   flags[0] = 1; // head
else
   flags[0] = -1; // tail

// set up buffer pool

   MPIRT_Buffer_Pool_Create(count[0], types[0], system_strategy, 1,
```

```
                         bufs, &bufpool_handles[0]);

// set up a single channel between two processes, utilizing
// time-based QOS, and using the functions in array ``fns'' to handle errors
// on either the sender or the receiver:...
   MPIRT_Channels_Bufpool_Init(bufpool_handles, 1, flags, qoss, error_fns,
                           comm, &requests, &errors);

// MPI_TIME_NO_OVERRIDE = use times in QOS parameter, don't change them:

if (rank == 0) {

// get the send buffer

   MPIRT_BUFFER_GET(bufpool_handles[0], user_strategy_send, 1, &index,
                    &channel);

// put the message into the buffer
// send the buffer

   MPIRT_Start_Time(request[0], MPI_TIME_NO_OVERRIDE, MPI_TIME_NO_OVERRIDE,
                     MPI_TIME_NO_OVERRIDE, MPI_HANDLER_NOOVERRIDE);
}
else
{

// after waiting for start.value + stop.value + MPI_WTIME_SKEW +
//                   MPI_WTIME_DRIFT + MPI_WTIME_TICK
// get the receive buffer for the newly arrived message
   MPIRT_BUFFER_GET(bufpool_handles[0], user_strategy_receive, 1, &index,
                    &channel);
}
// ...

MPIRT_Channel_coll_delete(comm, MPI_CH_CLOSE, requests); // end it!
```

For time-driven MPI/RT, it is sufficient to map a request into a function call specification (per invocation), so that no status is formed, and no special action is taken, unless a timeout occurs. Other violations could occur, in the channel situation, and an analogous mechanism to the timeout event must be provided.

11.10 Event-Driven MPI/RT

The low level and high level event-driven real-time MPI paradigms provide functionality for local and global events, respectively. The low level event-driven paradigm provides a mechanism for scheduling with QOS an application handler upon the completion of a data transfer operation. The high level event-driven paradigm provides a mechanism for scheduling with QOS any application activity, including MPI data transfer, application functions and so on upon the system, the application or the MPI event occurrence. Both paradigms allow users to manage MPI, system and user resources using events.

11.10.1 Low-Level, Event-Driven Real-Time MPI

Request handlers are an ideal mechanism for implementing the event-driven paradigm. Handlers were introduced in MPI-2. The functionality of this paradigm can be used with either MPI or MPI/RT operation's requests. To help users better manage resources, two events for the data transfer completions are introduced. One event specifies the local completion of the data transfer, i.e. the message buffer can be reused, that is currently available on most platforms. The other specifies the global completion of the data transfer, i.e. the channel resources can be reused.

Request handlers

The low-level mechanism for event-driven MPI/RT is the request completion/resource release handler, shown below. (NOTE: This mechanism is closely based on the MPI_POST_HANDLER routine currently in the external interfaces chapter.)

MPI_REQUEST_POST_HANDLER(request, request_cond, cond_handler_fn, failure_fn, extra_state, qos)

IN	request	MPI request (handle).
IN	request_cond	request condition (integer)
IN	cond_handler_fn	request condition handler (function)
IN	failure_fn	qos failure handler (function)
IN	extra_state	user supplied state (pointer)
IN	qos	event-driven qos (handle)

Once MPI_REQUEST_POST_HANDLER has been called, the handler function cond_handler_fn is to be called within the qos parameter after the given request reaches the condition specified by the request_cond argument. Recall, that the event-driven qos specifies the bound required for the activation time of an event handler after the request_cond is reached. When the handler is called, it is passed the request, the status of the request, and the extra_state. If the condition handler cannot be called within qos specified (the allotted time in the case of either absolute or relative times, or the specified time has already passed in the case of absolute times only), then the failure handler failure_fn is called. Like the request handler, the failure routine is passed the request argument, that request's status, and the extra_state argument.

Discussion: The definition of MPI_REQUEST_POST_HANDLER is not precise. The exact meaning of *the handler function to be called* need to be defined. One definition may specify that the function should be in the ready queue of the operating system, while another may specify that the function should start its execution. These and other choices need to be discussed and voted on at December meeting.

Discussion: *Proposal for low level event-driven QOS*

There are two exceptional cases for the time argument: First, if qos is a relative time and its value is zero, then the handler is to be called "as soon as possible." (How soon is an implementation quality issue. The desired goal is an interrupt-like functionality.) If the time is MPI_TIME_IGNORE, the request handler will be called at some later time but not necessarily "immediately." Since an instantaneous response time is not practically achievable in the first case and since the response time is unspecified in the second case, the failure handler will never be called for either case.

If the request has reached the specified condition when the MPI_REQUEST_POST_HANDLER call is made, the handler is scheduled for execution (unless the qos specified time is absolute and has already passed, in which case the failure routine is called). Notice that for normal nonblocking calls, it may often be the case that the request has already completed. In such circumstances the user may wish to use a persistent version of the call generating the request, if it is available. This would allow the handler to be specified before the request is started. (The lack of late binding poses something of a problem, however. See the example given below.)

Note that a request can have only one handler for each of its conditions. If the user wishes to have a callback list for each condition, this must be implemented manually by having a high-level handler that calls the individual handlers one by one. If MPI_REQUEST_POST_HANDLER is called for a request and a condition for which a handler has already been specified and the handler has not yet been invoked or the request is a persistent one, then the old handler is replaced by the new handler. If the new handler is a null pointer, then a handler will no longer be called for the specified condition on that request.

The request conditions currently specified are as follows:

- MPI_REQUEST_COMPLETE The associated handler is called when the request in question has been marked complete. For example, if the handler is associated with a nonblocking or persistent send, then the handler is called after the send buffer is available for reuse. (Note that the handler may run concurrently with the process if the process was blocked on an MPI_WAIT on the same request at the time the handler was invoked. If the user wishes to avoid this, he/she must provide explicit synchronization.)
- MPI_REQUEST_RELEASE The associated handler is called when all resources associated with the last execution of the request are free. Continuing the above example, if the handler is associated with a nonblocking or persistent send, then this handler is called when all local buffers and network resources have been released. (The overall semantics of this condition are admittedly fuzzy. The condition is necessary, however, in order to guarantee real-time performance in certain circumstances. For example, in the case of the above example, one might want the handler to initiate another send request, guaranteeing that the two sends do not contend for system resources.)

Discussion: As an alternative to the MPI_REQUEST_RELEASE condition, we may wish to strengthen the notion of request completion for real-time systems to include the release of all system resources–not just the user buffer.

The request handler is assumed to be "full-weight." That is, it can execute any MPI call or system-specific synchronization call and may run for an indeterminate amount of time. (I.e., it is not restricted like a signal handler.) Also, handlers do not implicitly "consume" their request(s). The request passed to a handler can still be waited on or freed by the process before or after the handler is called, unless the handler itself explicitly frees the request. A request is not actually freed by MPI/RT until all of the handlers associated with that request have been called. Even if the process or another handler has called MPI_REQUEST_FREE on the request prior to the execution of the handler, the request is still valid and can be queried using the appropriate calls. One complication is MPI_CANCEL: If a request is cancelled prior to the execution of the handlers, the handlers for each condition are called in turn may note the fact that the request has been cancelled via MPI_TEST_CANCELLED.

Examples

MPI based example The following example demonstrates the use of a request handler with MPI functionality. A handler is posted on receive request to add messages to a linked list to be consumed by the local process. The semaphore routines are as per POSIX.1b. (Note: A persistent request could not be used because the buffer cannot be late bound. Thus, the handler must repost itself.)

```
sem_t wait,
      lock;
list_t buflist;
MPI_Request request;
MPI_Status status;


void buf_recv(MPI_Request *request, MPI_Status *status, buffer_t *oldbuf)
{
    buffer_t *newbuf;
    int flag;

    /* Check if request was cancelled */
    MPI_Test_cancelled(status, &flag);
    if (!flag)
    {

        /* Free the old request */
        MPI_Request_free(request);

        /* Add old buffer to the list and prepare to receive a new buffer */
        /* (with mutual exclusion)                                        */
        sem_wait(&lock);
        list_append(&buflist, oldbuf);
        newbuf = (buffer_t *)malloc(sizeof(buffer_t));
        MPI_Irecv(newbuf, sizeof(buffer_t), MPI_BYTE, MPI_ANY_SOURCE, 0,
                  MPI_COMM_WORLD, request);
        MPI_Request_Post_Handler(request, MPI_REQUEST_COMPLETE, buf_recv, NULL,
                                 newbuf, MPI_TIME_IGNORE);
```

```
        sem_post(&lock);
    }

    /* Signal the process that the handler is done */
    sem_post(&wait);

    return;
}


void main(int argc, char *argv[])
{
    buffer_t *firstbuf,
             *nextbuf;
    int done;

    /* Initialize MPI */
    MPI_Init(argc, argv);

    /* Initialize semaphores */
    sem_init(&wait, 0, 0);
    sem_init(&lock, 0, 1);

    /* Create first buffer and post a handler to append it to the list */
    firstbuf = (buffer_t *)malloc(sizeof(buffer_t));
    MPI_Irecv(firstbuf, sizeof(buffer_t), MPI_BYTE, MPI_ANY_SOURCE, 0,
              MPI_COMM_WORLD, &request);
    MPI_Request_Post_Handler(&request, MPI_REQUEST_COMPLETE, buf_recv, NULL,
                             firstbuf, MPI_TIME_IGNORE);

    /* Loop until buf_consume() routine says we're done */
    do
    {
        /* Wait for a handler to signal a buffer's arrival */
        sem_wait(&wait);

        /* Remove a buffer from the list (with mutual exclusion) */
        sem_wait(&lock);
        list_pop(&buflist, &nextbuf);
        sem_post(&lock);

        /* Consume and free the buffer */
        buf_consume(nextbuf, &done);
        free(buf);
    }
    while (!done);

    /* Cancel current request (waiting for handler to post a new request) */
```

```
    sem_wait(&lock);
    MPI_Cancel(&request);
    sem_post(&lock);

    /* Wait for handler to complete for the last time */
    sem_wait(&wait);

    /* Destroy semaphores */
    sem_destroy(&wait);
    sem_destroy(&lock);

    /* Finalize MPI */
    MPI_Finalize();

    exit(0);
}
```

MPI/RT based example This example shows how request handlers can be used with channels and buffer pools. In the code below, data is sent from process 0 to process 1. The receipt of data at process 0 triggers a handler that responds by sending data back to process 0.

```
MPIRT_Buffer_pool bufpools[2];
MPI_Request chnlreqs[2];
buffer_t *bufs[2][5];

void buf_reply(MPI_Request *chnlreq, MPI_Status *status, void *unused)
{
    int i;

    MPIRT_Buffer_get(bufpools[1], MPI_BUFFER_NEXTAVAIL, 1, &i, &chnlreqs[1]);
    buf_fill(bufs[1][i]);
    MPIRT_Buffer_avail(i, bufpools[1]);
    MPI_Do(&chnlreqs[1]);
    return;
}


void main(int argc, char *argv[])
{
    int flags[2],
        qoss[2],
        errors[2],
        myrank,
        done,
        i;
    void *fns[2];

```

```
/* Initialize MPI and get this process's global rank */

MPI_Init(argc, argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

/* Create send and receive buffer pools and channels */
for (i = 0; i < 5; i++)
{
    bufs[0][i] = (buffer_t *)malloc(sizeof(buffer_t));
    bufs[1][i] = (buffer_t *)malloc(sizeof(buffer_t));
}
MPIRT_Buffer_pool_create(sizeof(buffer_t), MPI_BYTE,
                         MPI_BUFFER_CIRCULAR_WAIT, 5, bufs, &bufpools[0]);
MPIRT_Buffer_pool_create(sizeof(buffer_t), MPI_BYTE,
                         MPI_BUFFER_CIRCULAR_WAIT, 5, bufs, &bufpools[1]);
if (myrank == 0)
{
    flags[0] = 1;
    flags[1] = -1;
}
else if (myrank == 1)
{
    flags[0] = -1;
    flags[1] = 1;
}
qoss[0] = MPI_QOS_IGNORE;
qoss[1] = MPI_QOS_IGNORE;
fns[0] = NULL;
fns[1] = NULL;
MPIRT_Channels_bufpool_init(bufpools, 2, flags, qoss, fns, MPI_COMM_WORLD,
                            chnlreqs, errors);

/* Have process 0 fill buffers and send them to process 1 then wait for
   a buffer to be sent back in response */
if (myrank == 0)
{
    do
    {
        MPIRT_Buffer_get(bufpools[0], MPI_BUFFER_NEXTAVAIL, 1, &i,
                         &chnlreqs[0]);
        buf_fill(bufs[0][i]);
        MPIRT_Buffer_avail(i, bufpools[0]);
        MPI_Do(&chnlreqs[0]);
        MPI_Do(&chnlreqs[1]);
        MPIRT_Buffer_get(bufpools[1], MPI_BUFFER_OLDEST, 1, &i);
        buf_consume(bufs[1][i], &done);
        MPIRT_Buffer_avail(i, bufpools[1]);
    }
```

```
        while (!done);
    }
    else if (myrank == 1)
    {
        do
        {
            MPI_Request_Post_Handler(&chnlreqs[0], MPI_REQUEST_COMPLETE,
                                     buf_reply, NULL, NULL, MPI_TIME_IGNORE);
            MPI_Do(&chnlreqs[0]);
            MPIRT_Buffer_get(bufpools[0], MPI_BUFFER_OLDEST, 1, &i);
            buf_consume(bufs[0][i], &done);
            MPIRT_Buffer_avail(i, bufpools[0]);
        }
        while (!done);
    }

    /* Destroy channels and finalize MPI */
    flag[0] = MPIRT_CLOSE;
    flag[1] = MPIRT_CLOSE;
    MPIRT_Channels_Delete(MPI_COMM_WORLD, flags, 2, chnlreq);
    MPI_Finalize();

    exit(0);
}
```

11.10.2 High-Level, Event-Driven Real-Time MPI

Introduction

Discussion: This whole section is the first draft of the proposal that was only partially discussed by the working group.

This section provides a limited functionality that support only some of many existing event-driven models currently in use by real-time and embedded systems.

The main goal of the event-driven approach to real-time MPI is to help the application to control the run-time environment in which it is running with explicitly scheduling, for now, of MPI and computation activities and resource usage. Coordination is required between MPI, the operating system, communication and network protocols to enforce the schedules.

In a nutshell, an application using event-driven MPI will be able to specify intervals *guarded* by the specified events in order to bound the resource usage of communication and computation activities.

The limited functionality presented here contains:

1. MPIRT_Guarded_Function and MPIRT_Guarded_Channel_Use that provide computation and communication activities bounded by intervals specified by events,
2. MPIRT_Register_Event_Names and MPIRT_Register_Generated_Event_Names that allow to create event-response lists for processes,

3. MPIRT_Generate_Event that allows application to generate user events.

Currently many application "wait" on system events or user control messages to schedule a handler, that in turn, schedules several application activities: functions, processes, threads, and data transfers. The model for high level event-driven paradigm presented in this section establishes the direct coupling between events and application activities without user handlers.

Events

Just like the MPI provides the interface for data flow, the high level event-driven section provides the interface for control flow.

An event is a discrete atomic instantaneous state transition without any liveliness. Three types of events are available: system events, communication events and user events. Each event is identified with a name. The name is associated with a persistent event. The type of the event indicates the type of the resource that generate the event. System events are generated by the platform environment, for example operating system. Communication events are coupled with persistent channels that are introduced in earlier sections, and are generated or captured by MPI. User events are dedicated to the synchronization of the resource usage among different processes (nodes) on the platform, and are generated by the application.

For the high level event-driven paradigm events are not necessarily local to the process or even a node. Each process registers with MPI the persistent event names that it wants MPI to "monitor" and the persistent event names that the process will generate.

Discussion: Due to a preliminary nature of this proposal no system events are presented in this section. In the other parts of the MPI-2 MPI_PROCESS_DIED is defined as the only system event. The system events will be added later in the full version of the proposal.

In this preliminary version all the communication events are associated with the MPI/RT channel use. This proposal contains only two events associated with the channel that are introduced in the low level event-driven section: MPI_Request_Complete and MPI_Handler_Complete. They represent local and global completion of the MPI channel data transfer. In order to properly match these events with the guarded activities, MPI associates a persistent global name with a channel. The channels name can be either provided to an implementation by the application or the implementation will assign a name to a channel. Hence, there are two persistent event names associated with the channel. For a channel named α they are: α_local_complete that is equivalent to MPI_Request_Complete for a data transfer for the channel α, and α_global_complete that is equivalent to MPI_Handler_Complete for a data transfer for the channel α.

The followign change to MPIRT_CHANNELS_BUFPOOL_INIT is proposed.

MPIRT_CHANNELS_BUFPOOL_INIT(bufs, nchannels, flags, qoss, fns, comm, channels_names, requests, errors)

IN	bufs	Array of transfer buffer pool addresses (handle)
IN	nchannels	Length of all arrays used locally (number of channels being opened)
IN	flags	array of length `nchannels`, with 1,-1 values, indicating endedness of ith channel
INOUT	qoss	array of quality of service parameters per channel handle)
IN	fns	array of function names for handlers on channels (function)
IN	comm	communicator (handle)
INOUT	channels_names	array of names of the channels that are used in event lists (pointer)
OUT	requests	request objects, one per channel (persistent request)
OUT	errors	specifies success, non-success, for each channel (int array)

The user can provide the channel names and MPI will assign them to the channels, or the user can specify MPIRT_IGNORABLE and MPI will provide the channel names and return them as an out parameter in channels_names.

Discussion: An analogous changes to the other versions of the channel initialization will be added later.

The user events has meaning only to the application. MPI is just a mechanism to match user events and responses as well as the mechanism for event delivery and response triggers. An application assigns a persistent name to a user event and notifies MPI about which process generates this event. This is the only event type that is generated by the user. The events of other two event types are generated by MPI and system.

MPI delivers all the events to the processes that are registered for them and then triggers application functions or data transfers according to the events that guard the activity.

Events and Responses

For any function or communication operation an application can specify events that trigger its start and that trigger its termination if it is not finished yet.

Discussion: *Proposal for merging event-driven and time-driven paradigms*

The time interval of the time-driven real-time MPI contains two events that are specified by the time stamps. From this perspective time-driven paradigm is just a subset of the event-driven one. There is, however, one critical difference that lie in the ability of the application to schedule its non MPI activities. For the time-driven paradigm there are existing facilities to start non-MPI activities using OS timers, spin-locks and others. That and the synchronized clocks allows the application to coordinate all of its activities, MPI and non MPI, local and global. There are no analogous mechanisms for event-driven paradigm, and event delivery/monitoring across the entire platform requires application action and sufficient communication support. This is the place where MPI can really help.

The main purpose of the events is to *guard* the interval when the activity may use resources. This is analogous to the time-driven paradigm where no resources will be used by an MPI/RT data transfer operation prior to its starting time of the operation time interval and, to the best of the MPI implementation effort, no resources will be used after timeout of the operation time interval.

Discussion: *Relationship to the existing* MPI *functionality*
Currently, in the MPI-2 there is already a notion of the event in the dynamic chapter (3.4.3); there are also event handlers in Section 9.8 of the draft. The only event specified there is MPI_PROCESS_DIED, which is a system-level event that an application can monitor. There are two relevant operations: MPI_SIGNAL and MPI_MONITOR, appearing in Chapter 3. However, only the monitor operation makes use of events.

Guards and Guarded Activities

Events "guard" a liveliness interval within which the activity can use resources. While many different activities are of interest for real-time and embedded system for this proposal we concentrated on two activities: application functions and MPI/RT data transfer operations over a channel.

Discussion: *Proposal to extend activity definition* For a future almost any application and MPI functionality should be considered for the activity. These may include jobs, processes, threads, operations, system configuration, and all MPI operations.

The guards are two lists. The first one is the list of events whose conjuncture trigger the activity. The second one is the list of events, such that any event on the list stops the activity if it is not finished by itself yet. If we need more comprehensive arithmetic of actions we can add it later. For completeness we may add action *IGNORABLE* for the empty action list.

Discussion: *Proposal for adding time specification for events* For merging event-driven and time-driven paradigms special event type called timers should be allowed. The timer will take a specification of the MPI_TIMEOBJ used by the time-driven paradigm. That will allow to specify the time for the beginning and/or timeout for the activity as well as mixed time/event guards.

Discussion: *Alternative event list specification* There is one other possibility is to specify guards as a list of intervals, where each interval is specified by starting event (time), and by finishing event (timeout). But this specification requires the number of starting events to be equal to the number of finishing events (we can still use ignorable to stab it out).

MPI is responsible to deliver events and to trigger (start or stop) an activity if it is eligible. As stated before for now only two activities are considered: application functions and MPI/RT data transfer operations. The API for application guarded function is

MPIRT_GUARDED_FUNCTION(start_list, stop_list, guarded_fn, extra_state, qos, failure_fn)

IN	start_list	list of the events that trigger the start of the function (pointer)
IN	stop_list	list of the events that trigger the stop of the function (pointer)
IN	guarded_fn	requested guarded function (function)
IN	extra_state	user supplied state (pointer)
IN	qos	event-driven qos for the delivery of events (handle)
IN	failure_fn	qos failure handler (function)

The first arguments start_list and stop_list define the liveliness interval for the guarded function. The guarded_fn provides the name of the guarded function, and extra_state provides the list of the input arguments for the guarded function. The qos specifies the event QOS, that provides the bound for the start time for the guarded function after the triggering event happened, regardless if it happened locally or remotely.

Analogously for MPI/RT guarded data transfer operation over a channel API is

MPIRT_GUARDED_CHANNEL_USE(start_list, stop_list, channel_request, qos, failure_fn)

IN	start_list	list of the events that trigger the start of the function (pointer)
IN	stop_list	list of the events that trigger the stop of the function (pointer)
IN	channel_request	persistent channel request (handle)
IN	qos	event-driven qos for the delivery of events (handle)
IN	failure_fn	qos failure handler (function)

Discussion: *Extensions* The issue of periodicity and its effect on event names will need to be discussed further. The existing MPI_SIGNAL can be used in an MPI implementation to send an event to a proper place. Alternatively MPI_MONITOR can be enhanced to include events discussed above. Notice also, that while action list specified only by the "sending" side, both sending and receiving sides may be aware of the action. For example, the function failure_fn of the receiving side will be triggered if the finishing event is triggered.

Event Registration

Each application process registers event names it wants MPI to monitor and event names it will generate. Since an application can only generate user events, only user event names that application will generate need to be registered with MPI. MPI is already aware where and how system and communication events are generated.

Discussion: All the event names that the process registers for appears in the start or stop list that guards an activity within the process. So in reality there is no need for registration, a compiler can pick the event names itself. Currently, the full registration of all events that are used to guard some process activity is presented.

The following operation registers event names that are used as guards in a process activities.

MPIRT_Register_Event_Names(event_list)

IN	event_list	list of the events (pointer)

The following operation registers user event names that process generates.

MPIRT_Register_Generated_Event_Names(event_list)

IN	event_list	list of the events (pointer)

An single operation for deregistration is also provided.

MPIRT_Deregister_Event_Names(event_list)

IN	event_list	list of the events (pointer)

Event Generation

The issue of how the events are delivered to the guarded activity is left to the implementation.

Discussion: There were several discussions on the topic of the event delivery. The following methods were suggested:

1. Between all the nodes establish point-to-point channels. MPI will use these channels to delivery events. Using the registration operations together with the system and MPI generated events, MPI creates matching tables for all registered events. Upon receiving messages on these channels, MPI checks if the event is sufficient to activate/deactivate an activity that is guarded by the received event(s).
2. Similar to the above but uses signals (MPI or system) for event delivery.
3. Hardware assisted event delivery.

As mentioned before system and communication events are triggered by system and MPI. The following operation is provided for the MPI notification of the user event.

MPIRT_Generate_User_Event(event_name)

IN	event_name	event name (handle)

EXAMPLE

```
MPI_Name data_transfer_start, channels_names[1];
MPI_Timeobj stop;
MPI_Timeobj period=MPI_TIME_IGNORE;
int counts[1];
MPI_Datatype types[1];
MPI_QOS qoss[1];
MPI_QOS qos_event = 1.0e3; // Initialize QOS for event delivery to 1 microsecond
MPI_GUARD start_guards[1], stop_guards[1];
MPI_EVENT_NAME monitor_list[1], generate_list[1]
MPI_int bufs[1];
MPI_Request[1];
MPI_Buffer_Strategy system_strategy       = MPI_BUFFER_CIRCULAR_NOWAIT,
                    user_strategy_sent    = MPI_BUFFER_NEXTAVAIL,
                    user_strategy_receive = MPI_BUFFER_NEWEST;

int flags[1];
extern void f();
extern void message_consumer();
void error_fns[1];
MPI_Buffer_Pool bufpool_handles[1];
MPI_Error errors[1];
MPI_Request channel;
int index;

//...
MPI_Comm comm; // Assume a communicator with exactly twp processes
//...

//time in nanoseconds
stop.type = MPI_RELATIVE;
stop.value = 8.0e6; // 8 milliseconds
stop_guards[0] = stop;

// Initialize all parameters of the channel and buffer pools

count = 100;

qoss[0] = MPI_QOS_IGNORE;  // do not set up any QOS
bufs[0] = (int *) malloc(count*sizeof(int));
types[0] = MPI_INT;
counts[0] = count;
error_fns[0] = f;

if (rank ==0)
 flags[0] = 1; // head
else
 flags[0] = -1; // tail
```

```
// set up buffer pool


    MPIRT_Buffer_Pool_Create (count[0], types[0], system_strategy, 1, bufs,
                              &bufpool_handles[0]);

// Set up a single channel between two processes
// without any QOS for channel use.
// The QOS will be set up by the guards for the data transfer
// that uses of the channel.
// This operation also assigns the name of the channel that
// will be used later as a guard to activate the message processing
// function.

   MPIRT_Channels_Bufpool_Init (bufpool_handles, 1, flags, qoss, error_fns,
                                comm, &channels_names, &requests, &errors);

if (rank == 0)
{
//...
//  This Activity Generates a Message.
// First, get the buffer.
 MPIRT_BUFFER_GET(bufpool[0], user_strategy_sent, 1,
                  &index, &channel);
//...
...
//...
// This Activity Sends a Message.
//...
// This activity uses the name of the channel. MPI implementation
// should monitor event data_transfer_start and the QOS for the delivery
// of this event specified by qos_event. This activity does not generate any
// user events.

    monitor_list[0] = "data_transfer_start";

    MPIRT_Register_Event_Names(monitor_list);

//  Transfer the message when the receiver is ready.
//  The message should be transferred within 8 milliseconds.

    start_guards[0] = "data_transfer_start";

    MPIRT_Guarded_Channel_Use(request[0], start_guards, stop_guards,
                              qos_event, MPI_HANDLER_NOOVERRIDE);

}
else
```

```
//...
// This activity receives the messages
//...
...
//...
// This activity processes the message
//....
// Register the activity events

monitor_list[0] = channel_name[0].global_complete;

MPIRT_Register_Event_Names(monitor_list);

generated_list[0] = "data_transfer_start";

MPIRT_Register_Generated_Event_Names(generated_list);

//...
// The body of the message processing
// ...

start_guards[0] = channel_name[0].global_finish;

MPIRT_Guarded_Funcetion(start_guards[0], MPI_GUARD_IGNORE,
                      message_consumer, MPI_FUNCTION_PARAMETER_IGNORE,
                      qos_event, MPI_HANDLER_IGNORE);

}

MPIRT_Channel_Coll_Delete(comm, MPI_CH_CLOSE, requests);  // end of main body

void message_consumer()
{
//...
// The body of message processing
//...
// First, receive the message
 MPIRT_BUFFER_GET(bufpool[0], user_strategy_receive, 1,
                  &index, &channel);
//...
// When message processing completes generate the event

  MPIRT_User_Generate_Event("data_transfer_start");

}
```

11.11 Priority-Driven MPI/RT

In the priority-based real-time programming paradigm, process and message priorities are used to meet timing specifications. However, because process priorities can be handled in a variety of ways, it is extremely difficult to provide portable mechanisms to specify them, and MPI/RT does not directly address this issue.

11.11.1 Message Priority

In MPI/RT priorities are specified and fixed per channel by a field in the QOS argument to the channel creation calls. As with other QOS parameters, the processes at the input and output end of the channel must provide the same priority or an error occurs. (NOTE: We still need to work out the range of process priorities as well as the mechanism for setting the process priority field of the QOS block.)

Because varying platforms may provide different levels of support for message priority at the OS level and below, MPI/RT specifies very little about how message priorities are implemented. In addition to passing message priority information to the appropriate OS and hardware layers, a high-quality MPI/RT implementation will order operations internally according to priority information. For example, given the choice between performing two different communication operations (such as receiving one message or another), the higher priority communication should be performed first. If the high priority communication blocks or stalls, lower priority communication may be initiated. Notice that in the general case, this implies that communication may need to be preempted. For example, if the user initiates a low-priority nonblocking send, the begins a high-priority send, the low-priority send would be stalled in favor of the high-priority send.

11.11.2 Process Priority

As stated above, MPI/RT makes no attempt to correllate process and message priorities, and indeed, has nothing to do with process priorities whatsoever (with the possible exception of calls in the dynamic process chapter, which may be passed priority information via the "info" argument, and the case of request handlers, mentioned below). Such functionality should instead be provided by domain-specific middleware. Such middleware could, for example, set the priority of a process to the priority of the messages it receives. In the absence of middleware, the application developer must manage process priorities explicitly. For example, if the programmer wishes for processes receiving high priority messages to have a high process priority, he/she must set the process priority explicitly. Similarly, in the case of request handlers associated with a prioritized message, the user of MPI/RT must explicity correlate the priority of the handler to that of the message by creating a thread, for example, that has a priority corresponding to that of the message, and allowing that thread to then handle the message. Alternatively, the request handler mechanism could be used to alter the priority of the receiving process, providing an implementation of the middleware mechanism described above. Notice that this implies that request handlers must almost certainly be run at a higher priority that any MPI/RT process or thread. Thus, while an MPI/RT *implementation* may need to concern itself with process priorities, the *interface* itself does not.

11.12 Ad Hoc MPI/RT

Ad hoc real-time systems have no explicit deadlines and are interactive. Interactive systems try to achieve adequate response times. Interactive parallel visualization is an example of an ad hoc real-time application. These interactive systems have hard and soft deadlines, with interactive behavior with the user. A mechanism for handling this mixture of deadlines as well as providing QOS for the interactive portion must be available for ad hoc implementations. Ad hoc MPI/RT is currently under investigation and more proposals will be made in the near future.

Discussion: Interactive real-time systems can take advantage of the notion of priority preferences. By assigning priorites and weighting the priorities by preferences, interactive QOS may be achieved. More discussion is required.

11.13 Soft MPI/RT

The correctness of real-time systems depends on the logical result of the computation and the time at which results are produced. For soft real-time systems, the user has flexibility with respect to hard deadlines. If certain deadlines are missed, the system continues to function correctly. Any system that tolerates intermittent delays may be considered soft. The degree that the system is allowed to miss deadlines is the key factor in defining the soft real-time system beavior.

Discussion: Proposal QOS: Quality of service parameters to define the deviation from the hard deadline could be considered. This gives flexibility to the system, where specific deadlines are hard and others are soft, the hard deadlines may be a reference point for defining the offset for the soft deadlines. One mechanism to achieve QOS is guarantee to tolerate a percentage of timeouts within a period. Another mechanism is to have priorites with preferences, where the constraints on preferences is tighter than priority preferences for ad hoc systems.

11.14 Issues in Resource Constrained Systems

For resource constrained systems, the following issues become more important than in other uses of MPI:

- Small amounts of program space (code bloat unacceptable),
- Small amount of data space (buffering limited, maximum message sizes may be limited),
- Simplified programming environments as compared to full-blown OS's,
- Static loading environments more nearly compatible with MPI-2's view of the world.

Such systems may also have real-time requirements.

MPI is already migrating into resource constrained prototype systems; hence it is interesting to offer suggestions, and possibly additional profile language, to address this particular space of applications and systems. One particular approach will be to offer a set of subset implementation profiles for resource constrained systems, so that if subsetting should occur, it can be done according to a systematic convention documented in the MPI-2 standard.

11.14.1 Resource Constrained MPI

There are users within the resource constrained computing arena that would like to use MPI, but are constrained by extreme limitations on local memory and storage space. These users often run codes stored in non-volatile memory, such as as FLASH, or ROM. While the use of an efficient linker can dramatically reduce the size of MPI libraries, the resulting binaries are still far too large to be incorporated into firmware. There are three particular areas of concern.

- Large executable size due to the many functions in MPI and their interdependence in many cases. (I.e., resource constrained MPI/RT may only be able to support a subset of MPI/RT functionality.)
- The amount of buffer space on the receiver side. (I.e., the illusion of infinite slack may be particularly untenable.)
- The amount of buffer space on the sender side. (I.e., the ability to pack derived datatypes may be restricted.)

As a result, we recommend that only a core set of MPI/RT functionality be required for resource constrained MPI/RT. (Exactly which routines should be preserved remains to be discussed.) In fact, because buffer space may be limited or nonexistent, the user should only expect that the synchronous and ready send operations are available (these routines require no explicit buffer space at the receiver) and that only the built-in datatypes are provided (neither requires buffer space at the sender or the receiver). The combination of these two restrictions would seem to completely eliminate the need for buffer space (except, perhaps, for the case of collective communication, which may require space for intermediate values). Alternatively, the user could be expected to provide buffer space explicitly via routines such as MPI_BUFFER_ATTACH and be restricted to using buffered send operations (as well as synchronous and ready ones). Also, direct memory transfer between source and destination buffers may be implementable only when the source of a message is explicitly specified in a receive call, so the use of MPI_ANY_SOURCE may be precluded.

Ten function calls, supporting only elemental MPI datatypes, provide sufficient cover for a wide variety of parallel programming paradigms, without requiring the library overhead of derived datatypes and topologies.

11.15 Instrumentation

Instrumentation is an essential aspect in providing application developers with the metrics needed to monitor quality of service assurances and fine tune specific hardware configurations. These metrics support performance portability, maintainability and fault tolerance. Real-time instrumentation includes, but is not restricted to monitoring application performance and monitoring MPI/RT performance. Other performance monitoring directly related to the overhead of MPI/RT operations will be implementation dependent. An important benefit from performance monitoring is the ability to capture global and local resource utilization information.

Currently, there is little information available to the user concerning internal MPI events. In some circumstances where timing is critical, an application could benefit from information about times of resources use by internal MPI (and native) communication events (and states). Real-time instrumentation must be sensitive to the impact of monitoring on the timing behavior of an application. To minimize this impact, MPI/RT instrumentation will include interfaces for existing monitoring instruments and/or event loggers. This design provides implementation independence.

Real-time instruments will not duplicate efforts provided in profiling tools, although some of the information collected may be duplicated. The distinction between profiling and instrumentation will be defined by global and local resource requirements and impact to timing requirements. Implementors may choose to use "profiling" hooks when applicable if performance can be achieved.

Discussion: A proposal to clarify performance monitoring: A proposal was made to research the definition of application performance monitoring. Most real-time applications have performance instruments to monitor resource usage associated with computations such as FFT's. Research is needed to determine how to integrate these instruments into MPI real-time instruments whenever applicable. More investigation is required.

As part of this general proposal, it was decided that real-time instrumentation for MPI/RT must consider monitoring to support quality of service assurances. Defaults must be identified that are implementation independent. These defaults must be provided as monitoring options and made available to the user. For this reason, bindings for monitoring are still under investigation.

11.15.1 MPI/RT Monitoring

Run time instruments support performance monitoring, decision analysis and fault tolerance. The MPI instrument monitoring API is designed to monitor, collect and output metrics. For systems that have a generic performance monitoring capability, MPI/RT monitoring may be integrated into the existing capability. Metrics that are obtained from performance monitoring may provide decision analysis criteria for conditional heuristics. These heuristics guide fault tolerance policies and support load balancing schemes.

Performance monitoring for both application specific information, and MPI/RT specific information are accommodated. The standard will not dictate this level of detail, but instead provide recommendations to implementors and users.

Runtime instruments are created and deleted. Activation of real-time instrument monitoring will be defined by a start and end statement to capture a snapshot of any size. Some inaccuracies will occur when MPI monitoring is turned off and some communications have not completed. MPI real-time monitoring is specified as follows:

Discussion: Proposal to add initialization: It was proposed that initialization should be independent of the create/delete.

MPI/RT Monitoring Management

The creation and deletion of monitoring is designed to decouple the burden of overhead for initialization and finalization from the ability to start and stop monitoring. More than one monitor may execute simultaneously. The execution of some monitors may be mutually exclusive.

MPIRT_MONITOR_CREATE(monitor_info, monitor_handle, errcode)

IN	monitor_info	monitor information (info)
OUT	monitor_handle	monitor handle (handle)
OUT	errcode	indicates reason for failure (integer)

MPIRT_MONITOR_DELETE(monitor_handle)

IN	monitor_handle	monitor handle (handle)

MPI/RT Monitoring Control

Monitoring control includes starting and stopping monitoring. Monitoring may be started and stopped at any time after `MPI_MONITOR_CREATE` has been called and before `MPI_MONITOR_DELETE` is completed. When monitoring is stopped, all flags are set to the default settings. When monitoring is started, the user may specify which specific elements (options) are to be monitored during the session. For MPI/RT monitoring, a set of defaults will be provided. For "other" performance monitoring, the implementor will provide the defaults as a non-portable aspect of monitoring with an option to return a warning upon execution if not supported.

Discussion: Proposal to Start and Stop Monitoring: A proposal was made to have more than one type of start and stop monitoring bindings to accommodate the differences between MPI/RT performance monitoring, and other performance monitoring. In MPI/RT, monitoring may be coupled to channels and some provision must be made to accommodate this monitoring seamlessly.

MPI/RT Output Monitoring Results

MPIRT_MONITOR_RESULTS(request, monitor_handle, f)

IN	request	request to output monitoring (handle)
IN	monitor_handle	handle for monitoring (handle)
IN	f	function to execute (function)

This function is provided for systems that do not provide a mechanism for outputting the performance information automatically and for support of decision analytic hueristics.

Discussion: The monitoring information is returned locally to the user. This function may not be called before MPI_MONITOR_STOP and must be called before MPI_MONITOR_DELETE is called.

No provisions are planned to allow cancellation of monitoring.

MPI/RT Monitoring Parameters

Discussion: This section will be defined when monitor stop and start are defined.

Monitoring Other Software Services

In addition to the MPI/RT instruments described in this section, an option for distinguishing other services related to MPI/RT is available.

MPI calls are supported with software services. For each MPI implementation, there is a software layer between the hardware and the MPI application layer bindings. This layer may differ for each architecture and may consists of operating system services, vendor specific MPI services and/or other services. The overhead from software services with respect to an MPI call may be of interest to the user and/or implementor. Some systems are not capable of accurately collecting timing information for these services. Implementations that can not support this option, return a warning message when the option is selected.

This option can provide a level of granularity to the MPI/RT instruments defined in this section. Typical information derived from monitoring other services might be the time spent in operating system calls. These operating system calls would be confined to those that support MPI/RT.

Advice to implementors. Examples of information of interest for MPI performance measures may include execution times, timeouts, counts and workloads.

1. Execution times

   Total MPI/RT execution time is time spent executing services for MPI/RT from a defined start point to a defined end point.

   ```
   execution time = (end time - start time)
   ```

   Some general categories for MPI/RT execution times may include time spent in a channel operation, a collective operation or a communicator group.

2. Counts

   The number of processes, number of times an MPI/RT communication is completed and the number of timeouts can be correlated with execution times to measure performance. This is a measure of quality of service.

   Timeout Counts are confined to timeouts of MPI/RT communications, and are extensible to collective operations and communicator groups.

   Collective operations counts may include a count of the number of processes participating and the number of MPI/RT communications completed.

   Communicator group counts may include a count of the number of MPI/ RT communications completed, total number MPI/RT communication timeouts and the number of MPI/RT collective operations completed

3. Workload

   Workload is a measure of the message sizes and traffic load over time and may defined per communication group. To develop metrics for workload, the following information may be collected per communicator:

   - message sizes (large small)
   - frequency large number of large messages
   - frequency small number of small messages
   - monitoring frequency of recordings taken

   Where message size definition (large and small) is recommended by the implementor.

   Time is determined by the start and end time in the snapshot created by the MPI/RT start and stop monitoring bindings.

(*End of advice to implementors.*)

11.16 Timing Correctness

Timing correctness has these properties:

- A real-time program can proceed, in a meaningful way, only if all previous steps are time-correct.
- Most basic MPI (current) functions should have analogous functions for MPI/RT that guarantee message passing in a timely fashion.

For practicality, the work of this chapter should be to minimize the number of new functions introduced into MPI profiles to provide these features. For instance, we should not add new versions of each send and receive mode, increasing the number of calls multiplicitively, but rather seek solutions that cause additive additions in the number of calls.

11.17 Deadlock Avoidance/Recovery

It is important to be able to prevent, avoid or in the worst case detect and recover from deadlocks. The problem of deadlocks is difficult in distributed real-time systems and it is exacerbated with prioritization.

Two related issues are as follows:

- Livelock — it happens when two interacting tasks miss deadlines due to one doing some secondary activity when the other tries to rendezvous.
- Orphan processes — in the presence of spawn capability (a process creating other processes) and failures.

MPI/RT will utilize timeouts to help applications deal with deadlock in the face of possible remote faults or remote resource exhaustion. Both sender- and receiver-based timeouts are to be considered.

Specifically, we expect non-blocking, persistent communication to be used to recover from deadlock. The following allows one to asynchronously start a receive, and then wait immediately for its completion, with a defined timeout.

```
MPI_Status status;
MPI_Request request;
MPI_Receive_init(buf, count, datatype, src, tag, comm, &request);
...
for(;;)
{
   MPI_Start(request);
   MPI_Wait_timeout(request, &status);
}
```

Alternatively, a single call,

```
MPI_Start_and_wait(request, status, timeout);
```

could be considered, and might have value in that it is atomic. To be complete, this proposal should address "all" modes, as well as single persistent operations.

A similar sequence of calls would be used to achieve send with timeout. This approach makes only an additive addition to the number of calls in MPI.

11.18 Fault Tolerance

A fault tolerance policy may have three components: operability assessment, fault detection and fault recovery. Operability assessment is preventative, while fault detection and fault recovery are generally late response indicators.

Performance monitoring supports operability assessment by gathering metrics associated with MPI performance. The heuristics determined from these metrics are designed to reduce the effective number of timeouts, thereby increasing quality of service guarantees. Performance monitoring is described in the section on real-time instrumentation. Operability assessment for detecting hardware faults using predictable algorithms is currently outside the scope of MPI/RT .

In some systems, message failure or loss may be handled for all traffic with a general fault detection/recovery policy. For MPI communication, some quarantee of quality of service may be achieved by detecting MPI RT communication errors. An MPI/RT communication fault is detected when data is erroneous, data is lost or the latency period exceeds a tolerance limit for processing effectively. Fault recovery in MPI will not include a requirement for message recovery. Continuance of processing will be quaranteed with replication but data loss may occur.

Fault recovery in MPI is accomplished with communicator recreation. Both early warning and late response indicators may precipitate reconfiguration. Early warning indicators are designed to reduce data loss.

11.18.1 Replication Policies

A system may provide redundancy in the form of replicated processes or nodes. When failure occurs, duplicate process(es) or node(s) may already be executing to prevent data loss. Characteristically, the replicated node or process does not exercise output options, but instead dumps data until conscripted into service.

Recoverable Communicator

Seamless replacement without degradation is an optimal goal of fault tolerance policies. When a fault is detected, a new comm group may be created by the application using `MPI_COMM_CREATE`.

- A fault occurs within a communicator.
- Quality of service reservations cannot be met because of some catastrophe.
- The parent of the spawned group is notified of the fault or faults.
- The parent, at its option, kills the faulty process or processes of the communicator through an asynchronous mechanism (*e.g.*, `MPI_SIGNAL`).
- The parent signals the remaining processes of the communicator asynchronously, and they execute a signal handler as follows:

  ```
  MPI_COMM_RECREATE(IN oldcomm, IN newgroup, OUT newcomm, OUT status);
  ```

  where the `newgroup` has been passed as part of the fault recovery, through a mechanism to be defined. It may be necessary for a single node to execute several handlers, if it is involved in a number of communicators, and implementations should be able to do this efficiently.

The semantics of `MPI_COMM_RECREATE` are that it synchronizes over the `newgroup`.

11.18.2 MPI Communication Fault Tolerance

If undetected, communication faults can lead to system degradation, data loss and non-recoverable errors.

11.18.3 Timeouts for Traditional MPI Functionality

In this section, the proposal to add timeouts to regular MPI functions is a step towards an underlying fault tolerance prevention schema for MPI.

11.19 Communicator Semantics with Channels

One[2] of the most important features of MPI is its support of safe communication spaces. The introduction of persistent channels that refer back to communicators opens up issues of inheritance of such channels, during an MPI_COMM_DUP.

Because channels are not explicitly attached to collective operations, but rather quality of service, there is no specific issue here.

By default, a dup'd communicator would have no realtime properties, and the channels of the parent communicator would not be available for use by the child communicator. This is clearly suboptimal in certain circumstances.

Qualities of service come in two forms: intensive and extensive. Extensive quality of service includes bandwidths, rates, and so on, and would have to shared, or used exclusively

[2]Initial proposal by A. Skjellum, to be revised at September meeting.

when communicators are dup'd. Intensive properties, of which priority is the prime example, do not need to be subdivided but can be shared without loss of generality.

Thus, the following discussion is needed:

- To support passage of channels through a dup'd communicator
- To provide a means to duplicate a persistent collective operation, whether realtime or regular (Collective Chapter),
- To distinguish intensive and extensive quality of service, so that some may be shared, and others used duplicatively.

11.20 Errors and Error Codes

Discussion: Error codes will be discussed in this section.

11.21 APPENDIX: Quality of Service

Advice to implementors. Based on ideas in the Tenet project, the following discussion is offered. Communication in real-time systems has to be predictable, since unpredictable delays in the delivery of a message can adversely affect the execution of tasks dependent on these messages. Unfortunately, standard MPI point-to-point and collective communication operations have poor real-time characteristics. In MPI/RT, we provide real-time channels to allow predictable inter-process communication in systems with partially connected point-to-point interconnection networks. These channels allow MPI/RT to provide guarantees on the maximum delivery time for messages.

Discussion: The term *real-time channel* historically denoted a connection characterized by an upper bound on packet delays [4]. More recently, however, it has been applied to connections offering bounds on some other performance indices, such as bandwidth or loss rate. We use the term here with the latter, extended meaning.

In MPI/RT, real-time channels are a bi-directional connection between the source and destination. A real-time channel is specified by a traffic characterization of the potential client of the channel, and quality of service parameters which describe the performance requirements of the source-destination communication. Once such a channel is established, MPI/RT guarantees that these performance requirements will be met.

In this type of network, there is only one source node for a network link, so the real-time performance issues are not influenced by access to the communications medium, but are highly dependent on message scheduling in the network nodes. Although it is possible that messages which traverse multiple links to reach the destination may suffer from higher latencies, these latencies can be made more predictable with explicit message scheduling policies.

Discussion: This implies that there needs to be a uniform message scheduling policy across the entire MPI/RT system. If a single node in a multi-hop real-time channel path does not properly schedule the forwarding of a message, no quality of service guarantees can be made.

Therefore in MPI/RT, real-time channels are connection-oriented, and reservation based. This means that before a real-time channel can be used by its requester, it must be established by means of an admission control test, so that the requested quality of service can be supported in addition to existing real-time channels.

It is important to note that in MPI/RT, network resources are not reserved for the channel's exclusive use. Clients of the channel resulting from the admission control negotiation have a higher priority for using the resources, but they can also be used by non-real-time traffic when it would not interfere with real-time traffic on the channel.

Real-time Admission Control

Admission control is a necessary component of a real-time communication service [4]. Without a limitation on the number of real-time channels admitted to simultaneously share the resources of a packet-switched network, the quality of the received message streams cannot be guaranteed in the presence of network contention.

Many methods can be used to determine whether a new real-time channel can be added to those already existing in the network [10, 13]. The principle behind these schemes is based on verifying, at the time a request for a new connection is made, whether the resources available along the path of the new connection are sufficient, even in the worst case, to

- Provide the new channel with the performance guarantees it requested
- Allows the guarantees offered to all existing real-time channels to continue to be satisfied

Admission Tests for Point-to-Point Links

In MPI/RT, the admission control test begins when a source node requests the creation of a real-time channel by specifying quality of service parameters and traffic characterizations. A request message is issued by the source of the requested real-time channel. Terms for the requested channel's admission are tested by each node along the path of the channel, and is allowed to proceed only if all the admission tests at each node are satisfied. MPI/RT requires that the admission or rejection decision should be completed in one round-trip, so the final admission tests take place at the destination of the connection.

Discussion: The MPI/RT channel admission test assumes static routing. Since each node along the route from the source of the real-time channel to the destination must agree to reserve resources for the channel, dynamic routing must not be allowed.

If any of the admission tests fails at any of the nodes, the channel cannot be established, and a failure message is sent back to the source. Each node removes the tentative resource reservations made by the initial admission test message.

Discussion: It is conceivable that the real-time user would prefer "best-effort" behavior in this instance, where MPI/RT would return as much of the network resources as are currently available. This would involve the node which has rejected the channel admission test to reduce the requested quality of service to the point at which it is acceptable to that node, and continue with the admission test. However, there will undoubtably be a point at which the quality of service represented by this incremental reduction in network resource reservation will render the channel useless.

Discussion: The process of establishing a real-time channel with distributed admission tests is potentially a time-intensive operation. It is strongly suggested that real-time channels be negotiated in advance of sending low-latency message traffic.

Collective Admission Tests

The `MPIRT_CHANNEL_INIT()` call currently operates as a collective operation across a group of processes/nodes. This makes the channel admission test a collective operation, and strays into an area which has not dealt with by the real-time research community.

Here is an initial collective admission test proposal:

In MPI/RT, channel establishment is a collective operation, with simultaneous participation from all the members of the channel's communicator. The admission test for a collective real-time channel is decomposed into several (many) separate admission tests which operate on the point-to-point channels which comprise the collective channel. For a point-to-point intercommunication network, the `MPIRT_CHANNEL_INIT()` call will result in the creation of $N(N-1)/2$ point-to-point channels. An admission test message is issued by the source node of each of these channels. This message contains an identical traffic descriptor (`MPI_Traffic`) and quality of service request object (`MPI_QOS`). The messages visit each node along the route of their respective point-to-point channels, and causes several admission tests and computations to be performed at each node. The collective admission test message contains a channel identifier which is unique across the `MPI_COMM_WORLD` continuum. If the new channel passes all the tests in a node, the message is forwarded, along with information about the state of the admission test, to the next node along the route. If the node performing the admission test has already processed an admission test for this channel (an intermediate node may be revisited many times by several communicator members), then no additional resource reservations are made. The final tests are performed at each destination of the respective point-to-point channels. If they are successful, a channel establishment message is sent by the destination along the reverse route. When each node is re-visited, the node updates the tentative resource reservations made by the node for the initial admission test message. If any of the nodes fail the admission test, failure messages must be recursively generated across all the point-to-point links which comprise the collective real-time channel, releasing all the tentative network resource reservations made by the initial admission test messages.

Collective Real-time Channel Operation

The net result of the collective admission test is a real-time channel which has reserved an aggregate quality of service. Note that this does *not* imply that the sum total of all communications on the real-time channel must not exceed the requested quality of service. Instead, the traffic across any *node* that handles traffic for the real-time channel must not exceed the allocated quality of service.

Discussion: This means that the performance of a collective real-time channel is highly dependent on the network topology. If the point-to-point channels which comprise the collective real-time channel are well distributed (as in a mesh topology), the net resource utilization will far exceed the negotiated quality of service. However, if a single node handles all the traffic for a collective real-time channel (as in a fat-tree), then that node will be a bottleneck for all real-time traffic on that collective channel.

Traffic Parameters and Quality of Service Parameters

In classic real-time channel admission tests, the requester specifies both performance bounds and traffic bounds for the channel. In MPI/RT, the traffic bounds are implementation specific and must be clearly delineated for the potential real-time user. These traffic bounds include:

- Xmin : the minimum inter-packet interval,
- Xave : the minimum of the average inter-packet intervals,
- I : the averaging interval used in the definition of Xave,
- Smax : the maximum packet size

Discussion: These parameters, which are often described in the real-time research community, are more descriptive than a single bandwidth parameter.

The performance bounds include:

- Dmax : the end-to-end delay bound,
- Zmin : the minimum probability that packet delay is not greater than Dmax,
- Jmax : the end-to-end delay jitter bound,
- Wmin : the minimum probability that a packet is lost due to a buffer overflow.

(*End of advice to implementors.*)

11.22 APPENDIX: Deadlock Avoidance/Recovery

It is important to be able to prevent, avoid or in the worst case detect and recover from deadlocks. The problem of deadlocks is difficult in distributed real-time systems and it is exacerbated with prioritization.

Two related issues are as follows:

- Livelock — it happens when two interacting tasks miss deadlines due to one doing some secondary activity when the other tries to rendezvous.

- Orphan processes — in the presence of spawn capability (a process creating other processes) and failures.

MPI/RT will utilize timeouts to help applications deal with deadlock in the face of possible remote faults or remote resource exhaustion. Both sender- and receiver-based timeouts are to be considered.

Specifically, we expect non-blocking, persistent communication to be used to recover from deadlock. The following allows one to asynchronously start a receive, and then wait immediately for its completion, with a defined timeout.

```
MPI_Status status;
MPI_Request request;
MPI_Receive_init(buf, count, datatype, src, tag, comm, &request);
...
for(;;)
{
   MPI_Start(request);
   MPI_Wait_timeout(request, &status);
}
```

Alternatively, a single call,

```
MPI_Start_and_wait(request, status, timeout);
```

could be considered, and might have value in that it is atomic. To be complete, this proposal should address “all” modes, as well as single persistent operations.

A similar sequence of calls would be used to achieve send with timeout. This approach makes only an additive addition to the number of calls in MPI.

Chapter 12

Spawning Independent Processes

These sections of the Dynamic Process chapter received through consideration and even received one vote of approval. They failed to receive a second vote, however. Those wishing to extend MPI to add this capability might consider these functions, which have been designed to be consistent with the other dynamic process functions, which spawn processes with which the spawning processes will communicate.

12.1 Starting Independent Processes

The previous sections described how an MPI application could spawn new MPI processes and establish communication with them. This section describes how to spawn *independent* processes, that is, processes that do not communicate with their parent. These may be MPI processes (i.e., processes that call MPI_INIT) or not.

MPI_SPAWN_INDEPENDENT(command, argv, maxprocs, info, flag, group, array_of_errcodes)

IN	command	name of program to be started (string)
IN	argv	arguments to command (array of strings)
IN	maxprocs	maximum number of processes to start (integer)
IN	info	info object telling the runtime system where and how to start the processes (handle)
IN	flag	says whether processes are MPI processes or not (integer)
OUT	group	group of spawned processes (handle)
OUT	array_of_errcodes	one code per process (array of integer)

```
int MPI_Spawn_independent(char command[], char* argv[], int maxprocs,
             MPI_Info info, int flag, MPI_Group* group,
             int array_of_errcodes[])
```

```
MPI_SPAWN_INDEPENDENT(COMMAND, ARGV, MAXPROCS, INFO, FLAG, GROUP,
             ARRAY_OF_ERRCODES, IERR)
```

```
CHARACTER*(*) COMMAND, ARGV(*)
INTEGER INFO, MAXPROCS, FLAG, GROUP, ARRAY_OF_ERRCODES(*), IERR
```

This routine spawns a set of processes that do not establish communication with the parent processes. It therefore returns a group instead of an intercommunicator. It is not collective, and is called on a single process. The flag argument specifies whether or not the children are MPI processes, that is, whether or not they call MPI_INIT. In other respects, this routine is similar to MPI_SPAWN.

12.1.1 MPI processes

If flag has the value MPI_MPI then MPI_SPAWN_INDEPENDENT is identical to MPI_SPAWN except that it does not establish communication with the children, returning an MPI group rather than an intercommunicator, and it is not collective. Successful completion does not indicate that the processes have called MPI_INIT successfully, only that the processes were started successfully.

The child processes are *required* to call MPI_INIT when flag is MPI_MPI. All children spawned in a single call to MPI_SPAWN_INDEPENDENT have the same MPI_COMM_WORLD, which is separate from that of the parent. In the children, MPI_COMM_PARENT has an empty remote group.

12.1.2 Non-MPI processes

If flag has the value MPI_NONMPI then the child processes are assumed to be non-MPI processes. MPI_SPAWN_INDEPENDENT is not required to do any special setup to ensure that a call to MPI_INIT in the children will be able to establish communication among the children. The effect of calling MPI_INIT in the children is undefined (but see advice to implementors in section 4.5.3.

12.2 Starting multiple independent processes

MPI_SPAWN_MULTIPLE_INDEPENDENT(count, array_of_commands, array_of_argv, array_of_maxprocs, array_of_info, flag, group, array_of_errcodes)

IN	count	Number of commands (integer, size of each of the following arrays)
IN	array_of_commands	programs to be executed (array of strings)
IN	array_of_argv	arguments for commands (array of array of strings)
IN	array_of_maxprocs	maximum number of processes for each command line to start (array of integers)
IN	array_of_info	info objects telling the runtime system where and how to start processes (array of handles)
IN	flag	says whether processes are MPI processes or not (integer)
OUT	group	group of spawned processes (handle)
OUT	array_of_errcodes	one error code per process (array of integer)

```
int MPI_Spawn_multiple_independent(int count, char* array_of_commands[],
             char** array_of_argv[], int array_of_maxprocs[],
             MPI_Info array_of_info[], int flag, MPI_Group* group,
             int array_of_errcodes[])
```

```
MPI_SPAWN_MULTIPLE_INDEPENDENT(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
             ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, FLAG, GROUP,
             ARRAY_OF_ERRCODES, IERR)
    INTEGER COUNT, ARRAY_OF_INFO, ARRAY_OF_MAXPROCS, FLAG, GROUP,
    ARRAY_OF_ERRCODES(*), IERR
    CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)
```

MPI_SPAWN_MULTIPLE_INDEPENDENT is identical to MPI_SPAWN_INDEPENDENT except that there are multiple executable specifications. The first argument, count, gives the number of specifications. Each of the next four arguments are simply arrays of the corresponding arguments in MPI_SPAWN_INDEPENDENT. See the description of MPI_SPAWN_MULTIPLE (section 4.3.3) for an explanation of array_of_argv in Fortran.

12.3 Nonblocking operations

MPI_ISPAWN_INDEPENDENT(command, argv, maxprocs, info, flag, group, array_of_errcodes, request)

IN	command	name of program to be started (string)
IN	argv	arguments to command (array of strings)
IN	maxprocs	maximum number of processes to start (integer)
IN	info	info object telling the runtime system where and how to start the processes (handle)
IN	flag	says whether processes are MPI processes or not (integer)
OUT	group	group of spawned processes (handle)
OUT	array_of_errcodes	one code per process (array of integer)
OUT	request	request object (handle)

```
int MPI_Ispawn_independent(char command[], char* argv[], int maxprocs,
             MPI_Info info, int flag, MPI_Group* group,
             int array_of_errcodes[], MPI_Request* request)
```

```
MPI_SPAWN_INDEPENDENT(COMMAND, ARGV, MAXPROCS, INFO, FLAG, GROUP,
             ARRAY_OF_ERRCODES, REQUEST, IERR)
    CHARACTER*(*) COMMAND, ARGV(*)
    INTEGER INFO, MAXPROCS, FLAG, GROUP, ARRAY_OF_ERRCODES(*), REQUEST,
    IERR
```

MPI_ISPAWN_MULTIPLE_INDEPENDENT(count, array_of_commands, array_of_argv, array_of_maxprocs, array_of_info, flag, group, array_of_errcodes, request)

IN	count	Number of commands (integer, size of each of the following arrays)
IN	array_of_commands	programs to be executed (array of strings)
IN	array_of_argv	arguments for commands (array of array of strings)
IN	array_of_maxprocs	maximum number of processes for each command line to start (array of integers)
IN	array_of_info	info objects telling the runtime system where and how to start processes (array of handles)
IN	flag	says whether processes are MPI processes or not (integer)
OUT	group	group of spawned processes (handle)
OUT	array_of_errcodes	one error code per process (array of integer)
OUT	request	request object (handle)

```
int MPI_Ispawn_multiple_independent(int count, char* array_of_commands[],
             char** array_of_argv[], int array_of_maxprocs[],
             MPI_Info array_of_info[], int flag, MPI_Group* group,
             int array_of_errcodes[], MPI_Request* request)
```

```
MPI_ISPAWN_MULTIPLE_INDEPENDENT(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
             ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, FLAG, GROUP,
             ARRAY_OF_ERRCODES, REQUEST, IERR)
    INTEGER COUNT, ARRAY_OF_INFO, ARRAY_OF_MAXPROCS, FLAG,GROUP,
    ARRAY_OF_ERRCODES(*), REQUEST, IERR
    CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)
```

12.4 Signalling Processes

It is not possible to send messages to a process represented only by a (group, rank) pair. For instance, it is not possible to send a message to an MPI process spawned by MPI_SPAWN_INDEPENDENT, since no communicator is available.

However, MPI provides simple mechanisms to manage such processes. MPI_SIGNAL sends a signal to a process, and MPI_PROCESS_MONITOR provides a mechanism to detect when a process dies or changes state.

MPI_SIGNAL(group, rank, signal)

IN	group	group containing process to signal (handle)
IN	rank	rank of process to signal (integer)
IN	signal	signal type (integer)

```
int MPI_Signal(MPI_Group group, int rank, int signal)
```

```
MPI_SIGNAL(GROUP, RANK, SIGNAL, IERR)
    INTEGER GROUP, RANK, SIGNAL, IERR
```

MPI_SIGNAL sends a signal to a process represented by a (group, rank) pair. If rank is MPI_ANY_SOURCE, the signal is sent to every process in the group.

Where POSIX signals are supported, signal is a signal defined by POSIX. It is the responsibility of an implementation to translate between signals; in other words, a SIGINT that has value 3 on system A must be delivered as a SIGINT on system B, even if system B uses the value 5 for SIGINT. If there is no corresponding signal, the operation is erroneous and the result is undefined.

An MPI implementation must also support the MPI-defined signal type MPI_SIGNAL_KILL. Sending this "signal" will reliably kill a process and attempt to do the necessary system-level cleanup. In order to allow appropriate cleanup to occur, an implementation could, for instance, send a catchable signal (e.g., SIGINT) followed by a noncatchable signal (SIGKILL) if the process had not exited after a short time interval. MPI requires only that the process be reliably killed. Applications should not rely on a specific implementation.

Signals are delivered asynchronously. That is, when the MPI_SIGNAL function returns, the signal may not yet have been delivered. If a signal is undeliverable (e.g., the process has already exited), the signal is silently dropped.

If an MPI implementation can determine, at the time MPI_SIGNAL is called, that the process is already dead, it may return the error MPI_ERR_NOPROCESS. If MPI_ANY_SOURCE had been specified and some of the processes are known to be dead while others are not, no error is returned.

> *Advice to users.* Signals do not provide any of the reliability or guarantees of regular MPI communication. A high quality implementation will deliver signals quickly and reliably, but applications should never depend on ordering. For example, a signal sent after a message may arrive before the message does. Since signals may not be queued, when multiple signals are sent, only one may get through. Some MPI implementations may use signals internally. If an application attempts to use these signals, unexpected behavior may result. Finally, it may not be possible at the implementation level to restrict the effects of a signal to a single MPI process, so that a side-effect of sending a signal to a process may be that other processes receive that signal or are killed. (*End of advice to users.*)

12.5 Notification of change in state of a process

In order to manage processes an MPI application must be able to be notified when a process exits.

MPI_PROCESS_MONITOR(group, rank, event, request)

IN	group	a group of processes (handle)
IN	rank	rank of process to be monitored (integer)
IN	event	a flag specifying what event to be notified about (integer)
OUT	request	a request object (handle)

```
int MPI_Process_monitor(MPI_Group group, int rank, int event,
             MPI_Request* request)
```

```
MPI_PROCESS_MONITOR(GROUP, RANK, EVENT, REQUEST, IERR)
    INTEGER GROUP, RANK, EVENT, REQUEST(*), IERR
```

This function provides a general method for an MPI process to detect a change in state of another process. The request completes when the process changes state as indicated by event.

event is an integer specifying a change of state that the calling process wishes to detect. The most important state-change and the only one MPI currently defines for event is MPI_PROCESS_DIED. For this case, request completes when the process terminates. If a process has already terminated, the request completes immediately. Process termination is defined by the implementation, and may conceivably include cases where a process has become unreachable or whose status cannot be determined. If an application wants to be notified of several different events, it must call MPI_MONITOR once for each event. If multiple requests are active for a single type of event, all requests will complete when the event occurs.

A request generated by MPI_PROCESS_MONITOR must progress regardless of whether any other MPI functions are called. MPI_REQUEST_FREE on a monitor request deletes the request.

> *Advice to users.* As in other cases in MPI, freeing the request marks it for deletion but does not necessarily complete the request. Therefore a handler associated with a MPI_PROCESS_MONITOR request may be called after the request is freed. To prevent MPI from continuing to perform action on behalf of the request, it is necessary to call MPI_CANCEL. (*End of advice to users.*)

Future versions of the MPI standard may specify new types of events, and implementations may define new ones as well. If an application wants to be notified of several different events, it must call MPI_PROCSSS_MONITOR once for each event.

For a request generated by MPI_PROCESS_MONITOR, MPI_WAIT returns a status that contains no useful information (i.e., the status is undefined).

> *Rationale.* Another possibility would be to get the exit code from the status. However, the process may not have exited - it may have become unreachable. It may also have been killed by a signal and what we'd want to know is what signal killed it. In the end, we might have to reimplement Unix wait(). (*End of rationale.*)

MPI_GROUP_MONITOR(group, event, array_of_requests)

IN	group	a group of processes (handle)
IN	event	a flag specifying what event to be notified about (integer)
OUT	array_of_requests	array of request objects (array of handles)

```
int MPI_Group_monitor(MPI_Group group, int event,
              MPI_Request array_of_requests[])
```

```
MPI_GROUP_MONITOR(GROUP, EVENT, ARRAY_OF_REQUESTS, IERR)
    INTEGER GROUP, EVENT, ARRAY_OF_REQUESTS(*), IERR
```

This convenience function is equivalent to calling MPI_PROCESS_MONITOR once for each process in group. The ith element (numbered from 0) of array_of_requests is associated with rank i in the group. array_of_requests must have a number of elements greater than or equal to the size of group.

Chapter 13

Threads and MPI

This chapter contains a proposed definition of what mutexes and conditional variables would look like if integrated into a threaded MPI library. It was not voted on.

13.1 Global thread functions

It is useful to extend thread coordination functions to work across an MPI group of processes, including processes not under the same kernel. Such extensions can be implemented portably using MPI communication (with handlers) and proxies at each node to support the functions globally (of course, much more efficient implementations will be possible on many systems).

These functions can also be used to synchronize threads within one process, using the communicator MPI_COMM_SELF.

We outline below some possible choices.

13.1.1 Mutexes

MPI_MUTEX_CREATE(comm, mutex_attr, mutex)

IN	comm	communicator for group sharing mutex
IN	mutex_attr	info object for mutex attributes
OUT	mutex	mutex object

```
int MPI_Mutex_create(MPI_Comm comm, MPI_Info mutex_attr, MPI_mutex mutex)
```

```
MPI_MUTEX_CREATE(COMM, MUTEX_ATTR, MUTEX, IERROR)
    INTEGER COMM, MUTEX_ATTR, MUTEX, IERROR
```

Missing: Need to specify predefined mutex attributes. No avoid problems with priority inversion, may need to require that all processes in the group of comm have the same priority ceiling. If left implementation dependant, could be a problem in strongly typed languages such as C++.

Creates a mutex object that is shared by all processes in the group of comm. (We could, instead, return a newcomm and further overload communicators.) This is the extension of pthread_mutex_init

MPI_MUTEX_FREE(mutex)

INOUT	mutex	mutex object

```
int MPI_Mutex_free(MPI_Mutex *mutex)
```

```
MPI_MUTEX_FREE(MUTEX, IERROR)
    INTEGER MUTEX, IERROR
```

Free the mutex object (should that be collective?) This is the extension to pthread_mutex_destroy (Posix says that the behavior of a pthread_mutex_destroy call on a locked mutex is undefined. We can leave it this way, or have our usual, more lenient implementation of free, that waits until objects are not busy. The later is consistent both with Posix and with MPI, but may not be consistent with current implementations of Posix threads.)

MPI_MUTEX_LOCK(mutex)

IN	mutex	mutex to be locked

```
int MPI_Mutex_lock(MPI_Mutex mutex)
```

```
MPI_MUTEX_LOCK(MUTEX, IERROR)
    INTEGER MUTEX, IERROR
```

Discussion:
Note that the C++ binding is `const`. Is the `mutex` argument really IN? This implies that the OS must maintain the status of the mutex.

MPI_MUTEX_TRYLOCK(mutex, flag)

IN	mutex	mutex to be locked
OUT	flag	success flag (boolean)

```
int MPI_Mutex_trylock(MPI_Mutex mutex, int *flag)
```

```
MPI_MUTEX_TRYLOCK(MUTEX, FLAG, IERROR)
    INTEGER MUTEX, FLAG, IERROR
```

Discussion:
Note that the C++ binding is `const`. Is the `mutex` argument really IN? This implies that the OS must maintain the status of the mutex.

MPI_MUTEX_UNLOCK(mutex)

IN	mutex	mutex to be unlocked

```
int MPI_Mutex_unlock(MPI_Mutex mutex)
```

```
MPI_MUTEX_UNLOCK(MUTEX, IERROR)
    INTEGER MUTEX, IERROR
```

Discussion:
Note that the C++ binding is `const`. Is the `mutex` argument really IN? This implies that the OS must maintain the status of the mutex.

These are the extensions of the Posix calls pthread_mutex_lock, pthread_mutex_unlock and pthread_mutex_unlock. The lock call is blocking (like a wait), whereas the trylock function in nonblocking, like a test.

13.1.2 Condition variables

MPI_COND_CREATE(comm, cond_attr, cond)

IN	comm	communicator for group sharing condition variable
IN	cond_attr	info object for condition attributes
OUT	cond	condition object

```
int MPI_Cond_create(MPI_Comm comm, MPI_Info cond_attr, MPI_Cond *cond)
```

```
MPI_COND_CREATE(COMM, COND_ATTR, COND, IERROR)
    INTEGER COMM, COND_ATTR, COND, IERROR
```

Discussion:
Note that the C++ binding is `const`. Is the `cond` argument really IN? This implies that the OS must maintain the status of the condition variable.

Creates a condition variable that is shared by all processes in the group of comm. (We could, here, too, overload communicators, rahter than having a new opaque object.)

MPI_COND_FREE(cond)

INOUT	cond	condition object

```
int MPI_Cond_free(MPI_Cond *cond)
```

```
MPI_COND_FREE(COND, IERROR)
    INTEGER COND, IERROR
```

Free the condition object.
These are the MPI extensions to pthread_cond_init and pthread_cond_destroy.

MPI_COND_SIGNAL(cond)

IN	cond	condition to be signaled

```
int MPI_Cond_signal(MPI_Cond cond)
```

```
MPI_COND_SIGNAL(COND, IERROR)
    INTEGER COND, IERROR
```

Discussion:
Note that the C++ binding is `const`. Is the `cond` argument really IN? This implies that the OS must maintain the status of the condition variable.

MPI_COND_BROADCAST(cond)

IN	cond	condition to be signaled

```
int MPI_Cond_broadcast(MPI_Cond cond)
```

```
MPI_COND_BROADCAST(COND, IERROR)
    INTEGER COND, IERROR
```

Discussion:
Note that the C++ binding is `const`. Is the `cond` argument really IN? This implies that the OS must maintain the status of the condition variable.

MPI_COND_WAIT(cond)

IN	cond	condition to wait on

```
int MPI_Cond_wait(MPI_Cond cond)
```

```
MPI_COND_WAIT(COND, IERROR)
    INTEGER COND, IERROR
```

Discussion:
Note that the C++ binding is `const`. Is the `cond` argument really IN? This implies that the OS must maintain the status of the condition variable.

These are the MPI extensions of the Posix calls pthread_cond_signal, pthread_cond_broadcast and pthread_cond_wait. The cond_wait call is blocking. The cond_signal call unblocks at least one of the threads waiting on the condition (if any) and the cond_broadcast call unblocks all threads that are waiting on the condition.

Discussion: I have left the extension of pthread_cond_timedwait to the real-time people.

Chapter 14

Caching on MPI Handles

This chapter contains a proposal to describes the extension of caching on MPI objects to handles other than communicators. The Forum decided not to include in the Standard.

14.1 Introduction

The MPI standard provides a caching facility that allows an application to attach arbitrary pieces of information, called attributes, to communicators. Providing caching for other MPI handles provides a way for attaching information to these handles as well.

An example of the utility of caching is for a profiling library to intercept every call that creates a persistent request in MPI. The profiling library needs to "remember" the information about a persistent request since it is not available when MPI_START is called. The profiling library could cache the persistent requests characteristics on the persistent request. Then, whenever this persistent request is used in a profiled call, the needed information can be extracted. The advantage of caching the information is that the risk of using stale information from reuse of requests while the profiling library was disabled is eliminated.

14.2 Functions

MPI-2 provides general caching functions that allow attributes to be cached on the following types MPI handles:

- `MPI_COMM`
- `MPI_DATATYPE`
- `MPI_FILE`
- `MPI_GROUP`
- `MPI_REQUEST` (Generalized and Persistent only)
- `MPI_WIN`

Missing:

This is still written in the style of an MPI proposal. That is likely to change if it remains in the JOD. Whether we have a list of handles to cache on and what are the exact members of this list is up in the air.

> *Rationale.* It was decided not to allow caching on all handles since it could have a performance impact. For example, (a nonpersistent) MPI_Request is created and destroyed in time critical sections of code. However, profilers would like to be able to cache on persistent requests. Since they are longer lived, it was decided to allow caching on them. Also, a datatype might be created in critical sections of F90 code, e.g., for array sections. However, it was decided to allow caching on datatypes even though this may happen. (*End of rationale.*)

The other main difference between these functions and the ones in MPI-1 are that these are designed to work on systems where the size of an integer and an address are different. The same mechanism discussed in section 9.7.6 is used here. For the attribute caching functions this effects the extra_state and attribute_val which are passed by address in C and C++ but as integers in Fortran.

> *Rationale.* The difficulty with using the old functions is that the function does not know the type of handle being passed and cannot deal with the address size issue. To overcome these problems new functions are required. (*End of rationale.*)

With the new caching mechanism, a `keyval` created by the MPI function MPI_KEYVAL_CREATE may be used for caching on any of the above MPI handles. The new function is:

MPI_HANDLE_KEYVAL_CREATE(copy_fn, delete_fn, keyval, extra_state)

IN	copy_fn	Copy callback function for keyval
IN	delete_fn	Delete callback function for keyval
OUT	keyval	Key value for future access
IN	extra_state	Extra state for callback functions

```
int MPI_Handle_keyval_create(MPI_Copy_function *copy_fn, MPI_delete_function
              *delete_fn, int *keyval, void *extra_state
```

```
MPI_HANDLE_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA STATE, IERROR)
    EXTERNAL COPY_FN, DELETE_FN
    INTEGER KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

The copy and delete functions callback functions are modified accordingly:

```
typedef int MPI_Handle_copy_function(MPI_Handle *handle, MPI_Handle_type
              handle_kind, int keyval, void *extra_state, void
              *attribute_val_in, void *attribute_val_out, int *flag);

typedef int MPI_Handle_delete_function(MPI_Handle *handle, MPI_Handle_type
              handle_kind, int keyval, void *attribute_val, void
              *extra_state);
```

```
SUBROUTINE HANDLE_COPY_FUNCTION(HANDLE, HANDLE_KIND, KEYVAL, EXTRA_STATE,
             ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR)
    INTEGER HANDLE, HANDLE_KIND, KEYVAL, IERR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
    ATTRIBUTE_VAL_OUT
```

```
SUBROUTINE HANDLE_DELETE_FUNCTION(HANDLE, HANDLE_KIND, KEYVAL,
             ATTRIBUTE_VAL, EXTRA_STATE, IERR)
    INTEGER HANDLE, HANDLE_KIND, KEYVAL, IERR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

The predefined functions MPI_HANDLE_DELETE_FN, MPI_HANDLE_NULL_COPY_FN, and MPI_HANDLE_DUP_FN are defined in MPI-2. They are identical to MPI_DELETE_FN, MPI_NULL_COPY_FN, and MPI_DUP_FN in MPI-1 but with the new calling sequence.

The functions for putting, getting, and deleting attributes are also modified.

MPI_HANDLE_ATTR_PUT(handle, handle_kind, keyval, attribute_val)

INOUT	handle	handle to which attribute will be attached (handle)
IN	handle_kind	type of handle (integer)
IN	keyval	key value, as returned by MPI_KEYVAL_CREATE (integer)
IN	attribute_val	attribute value (choice)

```
int MPI_Handle_attr_put(MPI_Handle handle, MPI_Handle_type handle_kind, int
             keyval, void *attribute_val)
```

```
MPI_HANDLE_ATTR_PUT(HANDLE, HANDLE_KIND, KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER HANDLE, HANDLE_KIND, KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
```

MPI_HANDLE_ATTR_GET(handle, handle_kind, keyval, attribute_val, flag)

IN	handle	handle to which attribute is attached (handle)
IN	handle_kind	type of handle (integer)
IN	keyval	key value (integer)
OUT	attribute_val	attribute value, unless flag = false (choice)
OUT	flag	true if an attribute value was extracted; false if no attribute is associated with the key (logical)

```
int MPI_Handle_attr_get(MPI_Handle handle, MPI_Handle_type handle_kind, int
             keyval, void *attribute_val, int *flag)
```

```
MPI_HANDLE_ATTR_GET(HANDLE, HANDLE_KIND, KEYVAL, ATTRIBUTE_VAL, FLAG,
             IERROR)
    INTEGER HANDLE, HANDLE_KIND, KEYVAL, FLAG, IERROR
```

```
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
```

MPI_HANDLE_ATTR_DELETE(handle, handle_kind, keyval)

INOUT	handle	handle to which attribute is attached (handle)
IN	handle_kind	type of handle (integer)
IN	keyval	The key value of the deleted attribute (integer)

```
int MPI_Handle_attr_delete(MPI_Handle handle, MPI_Handle_type handle_kind,
             int keyval)

MPI_HANDLE_ATTR_DELETE(HANDLE, HANDLE_KIND, KEYVAL, IERROR)
    INTEGER HANDLE, HANDLE_KIND, KEYVAL, IERROR
```

The argument handle_kind must be MPI_HANDLE_COMM, MPI_HANDLE_DATATYPE, MPI_HANDLE_GROUP, MPI_HANDLE_PREQUEST, or MPI_HANDLE_FILE if the handle is a communicator, datatype, group, persistent request, or file, respectively.

Advice to users. The caching capability on communicators in MPI-1 has the same functionality as the capability presented here. One difference here is that handle is of type MPI_Handle and handle_kind is the second argument. A second difference is that the Fortran binding for arguments that are addresses in the C and C++ case are modified to help when integers and addresses are different lengths. Old calls can be converted to the new calls. This is accomplished by adding adding the second argument which is handle_kind. In addition in Fortran the type of the address arguments must be modified. Users are encouraged the use the new calls in the future to avoid the problems encountered with the older style calls. (*End of advice to users.*)

Adding an attribute to a handle does not change the results of any comparison functions in MPI. For example, MPI_GROUP_COMPARE is defined to be a comparison of group members. Thus, adding an attribute does not change the result.

In MPI-1, the caching capability was limited to communicators. The attributes could only be propagated when the communicator was explicitly duplicated with MPI_COMM_DUP. The exact outcome is determined by the callback function. For the new handles where caching is allowed, it is not possible to explicitly duplicate the object. As a result, the attributes do not propagate to new handles which may be derived from a handle with cached values. For example, performing a MPI_GROUP_INCL creates a new group without any of the attributes of the input group. As a result, the callback function for duplication may never be called for keyvals on the new handles. In such a case, the exact function provided is not of importance.

Advice to users. The predefined callback function of MPI_NULL_COPY_FUN is fine to use in these cases. (*End of advice to users.*)

Rationale. Not propagating attributes on other handles allows for a consistent and easy design. Some of the MPI functions that create new handles from old take two input groups, e.g., MPI_GROUP_DIFFERENCE. If attributes propagated it would be difficult to decide which ones from each group would be attached to the new handle.

This is especially true if the same keyval is used to cache information on both of the input handles. (*End of rationale.*)

In MPI-1, it says (p. 139, line 19), "MPI_COMM_GROUP returns in group a handle to the group of comm." It also says (p. 139, lines 4-5), "Each group constructor behaves as if it returned a new group object." In MPI-2, the second statement is important. If one caches on the handle returned by MPI_COMM_GROUP, the original group on the communicator is unaffected. Thus, issuing a second MPI_COMM_GROUP on the same communicator will return a group without the cached information.

Advice to implementors. In MPI-1, it is noted that it is not necessary to make a copy of a group when the identical group is used. However, the group creation functions "behaves as if it returned a new group object." If a reference-count mechanism is used, then it must allow for the caching on each group separately. Thus, greater care must be used with such a mechanism in MPI-2. (*End of advice to implementors.*)

14.3 Example

An example of how one might use these caching functions to cache the decoded datatype information on the datatype.

```
int datatype_keyval;

/* library initialization function.  MUST be called before the library
is used */
lib_init(int *datatype_keyval)
{
   MPI_Handle_keyval_create(MPI_HANDLE_NULL_COPY_FN, MPI_HANDLE_NULL_DEL_FN,
      datatype_keyval, NULL);
}


/* the library */
lib(MPI_DATATYPE datatype, int datatype_keyval)
{
   char datatype_char[1024];
   int flag;

   MPI_Handle_attr_get(datatype, MPI_DATATYPE_HANDLE, datatype_keyval, datatype_char, flag)
   if (!flag) {
      /* get datatype and cache for later use */
      MPI_Handle_get_char_data_type(datatype, datatype_char);
      MPI_Handle_attr_put(datatype, MPI_DATATYPE_HANDLE, datatype_keyval, datatype_char)
   }

   /* use datatype_char */

}
```

Chapter 15

Miscellany

This chapter contains the MPI Forum's work on a number of proposals. Two new datatype constructors were designed, but concern over the details of how they might be implemented prevented them from being voted in. In addition, there are proposals that take advantage of single-copy possibilities for message-passing on shared-memory machines. These were not discussed in detail and were not voted on.

15.1 New Datatype Constructors

15.1.1 Simple Struct Types

MPI_TYPE_SIMPLE_STRUCT(count, array_of_blocklengths, array_of_types, newtype)

IN	count	number of blocks (integer)
IN	array_of_blocklengths	number of elements in each block (array of integer)
IN	array_of_types	type of elements in each block (array of handle)
OUT	newtype	new datatype (handle)

This datatype constructor is present to allow portable struct datatypes to be constructed. A portable datatype is a basic datatype or a datatype constructed from portable datatypes using one of the constructors MPI_TYPE_CONTIGUOUS, MPI_TYPE_VECTOR, MPI_TYPE_INDEXED, or MPI_TYPE_SIMPLE_STRUCT.

This type constructor is similar to MPI_TYPE_STRUCT, except that the user does not supply an array of displacements. Instead, the successive blocks are assumed to be contiguous. Padding spaces are added, according to the default alignment rules for structure components. This facilitates the construction of datatypes for structures, as the user need not compute displacements.

This can probably be more elegantly expressed in terms of extents.

Example 15.1 The following code declares a structure type and creates a datatype for that structure type.

```
struct record {
   char name;
```

```
   double position[3];
   float  mass;
   }

MPI_Datatype record_type;
MPI_datatype types[3] = {MPI_CHAR, MPI_DOUBLE, MPI_FLOAT};
int          lengths[3] = {1, 3, 1};

MPI_Type_simple_struct( 3, lengths, types, record_type);
```

Rationale. The code above is much simpler and more natural than the current approach, that uses MPI_TYPE_STRUCT, and requires to compute displacements.

It also provides a definition of the record type that is architecture independent (assuming that compilers don't fiddle with the order of fields, which is true in C) – this facilitates transfer of records using RMA, an fits well with various proposals for third party transfers. (*End of rationale.*)

Advice to users. Datatypes constructed using MPI_TYPE_SIMPLE_STRUCT should be used only for structures that use the default alignment rules. They should not be used if alignment rules have been changed, using compiler options or compilation pragmas. (*End of advice to users.*)

Advice to implementors. The datatype constructor MPI_TYPE_SIMPLE_STRUCT is most easily implemented on systems that use *natural* data alignment rules. On such systems, each basic datatype has an extent and an alignment; the alignment of a compound type is the strictest alignment of a component. Padding is added so that each component is aligned at its natural alignment. We can then derive obvious definitions for the extent and alignment of portable MPI datatypes:
The extent and alignment of a basic datatype is the extent and alignment of the corresponding language type.
The alignment of a datatype built with one of the constructors MPI_TYPE_CONTIGUOUS, MPI_TYPE_VECTOR, and PI_TYPE_INDEXED is the alignment of the component datatype; the extent of the compound datatype is as defined in MPI1.
The alignment of a datatype built by a call to MPI_TYPE_SIMPLE_STRUCT(n, lens, types, newtype) is $\max_i$ `alignment(types[i])`; the extent is computed by laying out the components with minimal padding added so that each component starts at its alignment.

Some compilers treat the first component of a structure differently. E.g., the IBM compiler aligns doubles at word boundaries, but align a structure that starts with a double component at doubleword boundaries. The definition of the alignment of a datatype built with MPI_TYPE_SIMPLE_STRUCT is changed, accordingly. (*End of advice to implementors.*)

15.1.2 Contiguous Struct Types

Current Status: Failed 6-7-10 at September meeting.

Data structures may contain padding holes, added for alignment. The definition of an MPI datatype, done using the MPI_TYPE_STRUCT constructor, does not distinguish between holes that contain no significant data, and are there only for alignment purposes, and between holes that may contain significant data and were purposefully left out by a user that wishes not to overwrite some of the information in a structure. As a result, MPI cannot overwrite these holes, this prevents the obvious optimization of shipping a data structure as a contiguous block of data, padding holes included. The type constructor MPI_TYPE_CONTIGUOUS_STRUCT marks any such holes as "dont-care" locations, allowing the implementation to overwrite them.

MPI_TYPE_CONTIGUOUS_STRUCT(count, array_of_blocklengths, array_of_displacements, array_of_types, newtype)

IN	count	number of blocks (integer)
IN	array_of_blocklengths	number of elements in each block (array of integer)
IN	array_of_displacements	byte displacement of each block (array of integer)
IN	array_of_types	type of elements in each block (array of handle)
OUT	newtype	new datatype (handle)

The semantics of MPI_TYPE_CONTIGUOUS_STRUCT are identical to MPI_TYPE_STRUCT, except that, if a receive call uses a datatype constructed with this constructor, then the internal holes in the structure can be overwritten. The same rule (namely, that holes between components can be overwritten) applies also to datatypes constructed using MPI_TYPE_SIMPLE_STRUCT.

Discussion: It is reasonable to assume that holes in layouts defined using MPI_TYPE_SIMPLE_STRUCT contain no significant information, since this datatype constructor is expected to be used for specifying the layout of structures. Users will be required to use MPI_TYPE_STRUCT when holes hold significant information.

Advice to users. Users should arrange the fields in a C struct type in descending order of their size. This arrangement has two advantages: firstly it may reduce the size of the struct by eliminating padding between elements with different alignments (this is generally true for C programs). Secondly, it may increase the number of cases for which the implementation can perform block copies by ensuring that the first element of the structure has the most stringent alignment requirements. (*End of advice to users.*)

Advice to implementors. An implementation may ignore the constructor MPI_TYPE_CONTIGUOUS_STRUCT, and handle it exactly as MPI_TYPE_STRUCT. Thus, there is (almost) no overhead in supporting this constructor, without taking advantage of it. One possible way of taking advantage of it, is to adopt a canonical "wire" representation for structures. Suppose that the compiler uses, by default, natural alignment rules for structures: each basic component is aligned at a natural boundary. Suppose that one adopts natural alignment rules for messages on the wire, in a homogeneous environment: messages are padded so that each basic element is aligned at a natural boundary. Then

(1) A receiver can always decode an incoming message and retrieve correctly the basic components. This because the basic datatype of the element indicates at what boundary it is aligned.
(2) A sender can block copy a structure onto the "wire" (most likely, onto a message buffer), when the structure is naturally aligned, *relative to the start of the sending buffer*. This optimization does not require that the sending datatype be built using MPI_TYPE_CONTIGUOUS_STRUCT: The MPI library can scan datatypes, when they are committed, and recognize when a datatype (or a datatype fragment) is naturally aligned. Block copying can be used even if the holes in the datatype contain significant information.
(3) A receiver can block copy an incoming message onto the receive buffer when the receiving structure is naturally aligned *relative to the start of the receiving buffer* and holes can be ignored. I.e., when the receiving datatype is built using MPI_TYPE_CONTIGUOUS_STRUCT or MPI_TYPE_SIMPLE_STRUCT and is naturally aligned, relative to its first position.

The same wire representation may be used in a heterogenous environment, in cases where data conversion is done at the destination; the destination has to know, not only the basic datatypes used by the source, but also the alignment rules used by the source.

Example: Consider a C structure `struct {char a[2]; float b[2]; double c}`; assume that floats are 4 bytes and doubles are 8 bytes, and structures are naturally aligned. The memory layout is `[c][c]xx[ffff][ffff]xxxx[dddddddd]`, and the structure is aligned to an 8 byte boundary.

Case 1: The entire structure is communicated. Sender constructs a datatype for structure using MPI_TYPE_STRUCT, MPI_TYPE_SIMPLE_STRUCT or MPI_TYPE_CONTIGUOUS_STRUCT; receiver constructs a datatype for this structure, using MPI_CONTIGUOUS_STRUCT or MPI_TYPE_SIMPLE_STRUCT. Then the layouts in the sender memory, on the wire, and in the receiver memory are identical, and data can be simply copied.

Case 2: communication involves only the second of the two characters, and the remainder of the structure. Receiver constructs the datatype using MPI_CONTIGUOUS_STRUCT. The memory layout is `[c]xx[ffff][ffff]xxxx[dddddddd]`, and the wire layout is `[c]xxx[ffff][ffff]xxxx[dddddddd]`. The datatype is not naturally aligned, so that block copying cannot be used at the sender or the receiver. However, a smart implementation may realize that the fragment starting from the float is naturally aligned, relative to the start of the buffer, so that block copying can be used for this fragment.

Case 3: communication involves only the two floats and the double. The memory layout is `[ffff][ffff]xxxx[dddddddd]`. The layout on the wire is `[ffff][ffff][dddddddd]`. The datatype is not naturally aligned, relative to its start, so that block copying cannot be used. (*End of advice to implementors.*)

Alternatives:

1. No new function MPI_TYPE_CONTIGUOUS_STRUCT: instead, holes can be overwritten only for datatypes built with MPI_TYPE_SIMPLE_STRUCT. The argument is that, at least in the scenario I presented, hole overwriting is taken advantage of only for structures with a natural alignment. But, then, their layout can be specified using MPI_TYPE_SIMPLE_STRUCT.

2. Both sender and receiver are required to use MPI_TYPE_CONTIGUOUS_STRUCT, and the

datatypes on both ends have to be structurally equivalent, i.e., defined by the same sequence of definitions. There are several possible variants for such proposal

(a) This construct can be only used in a homogeneous environment. But, then, we lose portability to heterogeneous environment, and we achieve little that could not be achieved by using MPI_BYTE.

(b) This construct can be used portably, with enhanced performance obtained in a homogeneous environment. E.g., a homogeneous implementation will use block copying, while a heterogeneous implementation will move element by element (at least, for heterogeneous communicators). This assumes that, in a homogeneous environment, different processes are compiled with the same data alignment options. Also, this runs counter the datatype matching rules in MPI, where only signature matching is required.

(c) MPI_TYPE_CONTIGUOUS_STRUCT can be matched with other datatypes, as long as signatures match; improved performance is achieved when both ends use MPI_TYPE_CONTIGUOUS_STRUCT. But, then, the wire protocol must be so that individual elements can be extracted, in case the receiver does not use MPI_TYPE_CONTIGUOUS_STRUCT. In this case, asking the sender to use MPI_TYPE_CONTIGUOUS_STRUCT does not provide additional function over the main proposal. The difference is only whether the use asserts that the sending datatype has a natural layout (using MPI_TYPE_CONTIGUOUS_STRUCT), or whether MPI discovers that fact by itself.

15.2 Two Process "Shared" Buffers

It is not uncommon for a pair of processes to engage in frequent pre-determined communications. On example of such a code is a pipeline algorithm in which process A creates data that is subsequently used by process B. Another example is a finite difference approximation model when communicating boundary values (a.k.a. ghost points). On a machine with shared memory, it is possible to share a common buffer to use for exchanging such data. Ownership of the buffer must be established in order for a process to safely fill or empty the buffer. This can be expressed as a produce/comsume relationship. This functionality can be expressed in such a way that it should be no less efficient when using MPI Send/Recv on a distributed memory machine and is simply a pointer pass on shared memory machines.

Discussion: This proposal is prompted by Tony Skjellum's proposal titled: "Infinite asymptotic bandwidth for thread communication"

We have a similiar protocol inside our system when we work on shared memory machines. Important to us is not only the bandwidth and latency, but also lightweight flow-control. The proposal in its current form solves our problem in particular, but we feel it may prompt discussion that can shape this proposal and Tony's proposal into something that will provide a portable high bandwidth/low-latency mechanism on true SMPs as well as provide high performance in distributed memory machines.

Questions for discussion:

1) Is this really going to be faster on a shared memory machine with a "high-quality" implementation.

2) Does the user visible flow-control provide significant added value?

3) For both shared and distributed memory machines, does predefining the communication pattern provide options for optimization?

`Example:`

```
Process 0:                            Process 1:

! Make a shared buffer with process 0 as the source, process 1 as the dest
! sbh is the shared buffer handle and buff is some memory allocated
! by MAKE_SHARED_BUFFER
MAKE_SHARED_BUFFER (...,              MPI_MAKE_SHARED_BUFFER (...,
      0, 1, buff, sbh)                          0, 1, buff, sbh)
! Process 0 owns the buffer           ! Process 1 waits for buffer
 ...Fill buff with useful stuff...    ACQUIRE_SHARED_BUFFER (sbh)

! Process 1 releases the buffer
! communicating that it was
! modified (PRODUCEd)
RELEASE_SHARED_BUFFER (sbh, PRODUCE)
                                      ! At this point process 1 unblocks
! Wait for process 0 to use the       ! and has full access to buff
! Information produced before         ...Read buff and do something...
! creating more data
ACQUIRE_SHARED_BUFFER (sbh)           ! Process 1 releases the buffer, but
                                      ! has not modified it (CONSUMEd)
                                      RELEASE_SHARED_BUFFER (sbh, CONSUME)
! At this point process 0 unblocks
! and is free to modify buff
          ...                                  ...
FREE_SHARED_BUFF(sbh)                 FREE_SHARED_BUFF(sbh)
```

MPI_MAKE_SHARED_BUFFER (dataType, count, comm, source, dest, buffer, sharedBuffHandle)

IN	dataType	MPI Data Type (contiguous)
IN	count	Number of dataTypes
IN	comm	Communicator
IN	source	First process in pair (initially owns buffer)
IN	dest	Second process in pair
IN	tag	Tag for use in communications and for uniqueness
OUT	buffer	Buffer creaated by make_shared_buffer
OUT	sharedBuffHandle	Handle to describe shared object

```
int MPI_Make_shared_buffer (MPI_Datatype dataType, int, count,
            MPI_Comm comm, int source, int dest,
            void *buffer, MPI_Shared_buff *sharedBuffHandle)

MPI_MAKE_SHARED_BUFFER (DATA_TYPE, COUNT, COMM, SOURCE, DEST,
                       BUFF_REF, BUFFER_IX, SHARED_BUFF_HANDLE, IERROR)
```

```
    INTEGER DATA_TYPE
    INTEGER COUNT
    INTEGER COMM
    INTEGER SOURCE
    INTEGER DEST
    INTEGER BUFF_REFF(*)
    INTEGER BUFFER_IX
    INTEGER SHARED_BUFF_HANDLE
    INTEGER IERROR
```

This routine creates a buffer shared between two processes. The process identified by source initially "owns" the buffer. Source may write to the buffer and manipulate the buffer in any way that makes sense with standard language access (assignment statements, functions calls, etc.). When source is done with the buffer, it must release the buffer for dest to acquire it. Dest (and only dest) can acquire the buffer once released by source. Dest then has the same priviledgs as source previously had. To access a released buffer is considered to be erroneous. This function needs to be viewed as "bi-collective".

Discussion: Do we want to have a version that allows the user to supply a buffer. In that event on a distributed memory machine this becomes a "safe" interface to ready send, on a shared memory machine it may introduce extra copies.

MPI_ACQUIRE_SHARED_BUFFER (sharedBuffHandle)

INOUT	sharedBuffHandle	A shared buffer handle

```
int MPI_Acquire_shared_buffer (MPI_Shared_buff *sharedBuffHandle)

MPI_ACQUIRE_SHARED_BUFFER (SHARED_BUFF_HANDLE, IERROR)
    INTEGER SHARED_BUFF_HANDLE
    INTEGER IERROR
```

This routine waits for the shared buffer to be released by the partner. It is a blocking call. When it returns, the buffer associated with the shared buffer handle can be fully accessed. A subsequent acquire without an intervening release is erroneous (this includes an acquire by the source immediately after the make_shared_buffer). An acquire will correctly return only when the partner process performs a release.

MPI_RELEASE_SHARED_BUFFER (sharedBuffHandle, releaseAction)

INOUT	sharedBuffHandle	A shared buffer handle
IN	releaseAction	Indicate whether a produce (MPI_PRODUCE) or consume (MPI_CONSUME) action is associated with the release

```
int MPI_Release_shared_buffer (MPI_Shared_buff *sharedBuffHandle,
          int releaseAction)

MPI_RELEASE_SHARED_BUFFER (SHARED_BUFF_HANDLE, RELEASE_ACTION, IERROR)
   INTEGER SHARED_BUFF_HANDLE
   INTEGER RELEASE_ACTION
   INTEGER IERROR
```

This routine releases a shared buffer so that it can be acquired by the partner process. If the process has only read the buffer associated with the shared buffer handler, it may use the releaseAction of MPI_CONSUME. If the process has written to the buffer associated with the shared buffer handler, it MUST use the releaseAction of MPI_PRODUCE to guarantee that the partner sees the changes.

MPI_FREE_SHARED_BUFFER (sharedBuffHandle)

INOUT	sharedBuffHandle	A shared buffer handle

```
int MPI_Free_shared_buffer (MPI_Shared_buff *sharedBuffHandle)

MPI_FREE_SHARED_BUFFER (SHARED_BUFF_HANDLE, IERROR)
   INTEGER SHARED_BUFF_HANDLE
   INTEGER IERROR
```

This function frees the buffer associated with the shared buffer handler, the handle itself, and satisfies any outstanding communications. This function needs to be viewed as "bi-collective".

```
Possible Send/Recv Implementation:

int MPI_Make_shared_buffer (MPI_Datatype dataType, int, count,
            MPI_Comm comm, int source, int dest,
            void *buffer, MPI_Shared_buff *sharedBuffHandle)
  {
  /* Ok, ok this isn't really C, but you get the idea) */
  sharedBuffHandle = malloc (sizeof(sharedBuffHandle));
  *buffer = malloc (sizeof(dataType) * count);
  sharedBuffHandle->buffer = buffer;
  sharedBuffHandle->count = count;
  sharedBuffHandle->dataType = dataType; /* Dup here? */
  sharedBuffHandle->tag = getSharedBuffTag(); /* Some magic? */
  /* At this point, a robust application might check with it's partner
     to see if the parameters are correct and the malloc worked. But it
     would take too much space (besides if programming text books
     can omit error checking, why can't I? :-) */
  if (myProc == src)
    {
```

```
    Recv (comm, sharedBuffHandle->tag, dest, (count=0, buff=NULL,...));
    sharedBuffHandle->partner = dest;
    }
  else
    {
    sharedBuffHandle->waitHandle = IRECV (comm, sharedBuffHandle->tag,
                                          source, buffer, count,...)
    Send (comm, sharedBuffHandle->tag, source, (count=0, buff=NULL,...));
    sharedBuffHandle->partner = source;
    }
  return;
  }

int MPI_Acquire_shared_buffer (MPI_Shared_buff *sharedBuffHandle)
  {
  wait (sharedBuffHandle->waitHandle);
  }

int MPI_Release_shared_buffer (MPI_Shared_buff *sharedBuffHandle,
          int releaseAction)
  {
  if (releaseAction == MPI_CONSUME)
    count = 0;
  else
    count = sharedBuffHandle->count;
  sharedBuffHandle->waitHandle = IRECV (comm, sharedBuffHandle->tag,
                                        sharedBuffHandle->partner,
                                        sharedBuffHandle->buffer, count,...);
  /* Note that a non-buffering send can be used here */
  /* Note that a "ready" protocol can also be used since we guarantee
     the receive is always posted before the send */
  localHandle = ISSEND (comm, sharedBuffHandle->tag, sharedBuffHandle->partner,
                        sharedBuffHandle->buffer, count,...);
  freeHandle (localHandle);
  }

int MPI_Free_shared_buffer (MPI_Shared_buff *sharedBuffHandle)
  {
  /*
    Either cancel the outstanding receive or (if you don't beleive
    cancel works) send to complete the receive.
  */
  /*
    Free all structures
  */
  }
```

Possible Shared Memory Implementation:

```
int MPI_Make_shared_buffer (MPI_Datatype dataType, int, count,
            MPI_Comm comm, int source, int dest,
            void *buffer, MPI_Shared_buff *sharedBuffHandle)
  {
  /* Ok, ok this isn't really C, but you get the idea) */
  sharedBuffHandle = malloc (sizeof(sharedBuffHandle));
  if (myProc == source)
    {
    sharedBuffHandle->semaphore =
                shared_malloc (sizeof(dataType) * count + cacheLineSize);
    }
  sharedBuffHandle->buffer = buffer;
  sharedBuffHandle->count = count;
  sharedBuffHandle->dataType = dataType; /* Dup here? */
  if (myProc == source)
    {
    i = myProc;
    memcpy (sharedBuffHandle->semaphore, &i, sizeof(int));
    sharedBuffHandle->partner = dest;
    Send (comm, sharedTag, dest, sharedBuffHandle->semaphore, MPI_Pointer, ...);
    }
  else
    {
    sharedBuffHandle->partner = source;
    Recv (comm, sharedTag, source, sharedBuffHandle->semaphore,
                MPI_Pointer, ...);
    }
  *buffer = ((char *) sharedBuffHandle->semaphore) + cacheLineSize;
  }

int MPI_Acquire_shared_buffer (MPI_Shared_buff *sharedBuffHandle)
  {
  i = myProc - 1;
  while (i != myProc)
    {
    memcpy (&i, sharedBuffHandle->semaphore, sizeof(int));
    /* A smart implementation would release it's timeslice
       after some number of times through this loop */
    }
  }

int MPI_Release_shared_buffer (MPI_Shared_buff *sharedBuffHandle,
          int releaseAction)
  {
  i = sharedBuffHandle->partner;
  memcpy (sharedBuffHandle->semaphore, &i, sizeof(int));
  }
```

```
int MPI_Free_shared_buffer (MPI_Shared_buff *sharedBuffHandle)
  {
  /*
    Free all structures
  */
  }
```

15.3 Yet Another Send Mode

In the 1-sided chapter, we have introduced the notion of "good" memory vs. "bad" memory for gets and puts. The idea is that if your system supports shared memory, then you can call MPI_RMA_MALLOC to request a shared buffer and as a reward for doing this your gets and puts will be faster.

The thought occurs that 2-sided communication could benefit from this as well. In MPI-1, for example, many shared memory implementations still use a two-copy algorithm to pass messages around; the sender copies the data into a shared buffer, and the receiver copies it out again. While some systems are able to do better than this in some cases, many are not.

So, given that we already are going to be adding MPI_RMA_MALLOC to MPI-2, why not also add a new send mode to take advantage of it? Something like this:

```
MPI_Xsend
MPI_Ixsend
MPI_Xsend_init   (We can debate the proper value of 'X' later...)
```

In this mode, the sender asserts that the buffer being passed is one which was returned by a call to MPI_RMA_MALLOC. It otherwise has the same semantics as standard mode. With this mode, processes calling MPI_RECV on shared memory machines would be able to copy the data directly out of the send buffer. There could also be advantages on NOWs, because (for example) high performance implementations might opt to pin down the send buffers in advance to prepare for subsequent DMAs. Machines not capable of optimizing this case could simply use MPI_SEND to implement the new calls, much as often happens today with MPI_RSEND.

Adding additional smarts to the already-existing send modes is probably not a practical solution because of the added latency that would be involved in building internal lists of shared buffers and comparing against them with every sent message.

Doing this would probably involve making some minor changes to MPI_RMA_MALLOC. For example, it would probably have to become a collective function, and we would perhaps want to tweak the syntax slightly.

Does anyone else think that this idea has merit?

Chapter 16

Towards a Full Fortran-90 Interface

This chapter contains ideas for elements of a "Full" Fortran 90 interface that tries to address the problems of the F77-based interface. Some of these ideas are compatible, and some are mutually exclusive.

16.1 A different way of handling buffer arguments

1. Define a constant MPI_AINT_KIND. An integer of this kind is enough to hold an address. This integer is conceptually similar to MPI_Aint and will be referred to in the rest of this message as an MPI_Aint.
2. The F90 version of F77 functions with choice argument take an MPI_Aint instead of the choice argument.
3. The MPI_ADDRESS function in Fortran 90 now returns an AINT.
4. Add the intrinsic function MPI_F90_DESCRIBE_OBJECT as described below, or one or more variants.
5. For backward compatibility with current MPI bindings, the following would work, but would be subject to all the caveats in the current F90 discussion:

```
real a(100) ! or real a(:)
MPI_ADDRESS(a, a_address)
MPI_ISEND(a_address, 100, MPI_REAL, ...)
```

6. For convenience, we add a new set of functions:

```
MPI_SEND_F90
MPI_ISEND_F90
MPI_BSEND_F90
etc.
```

16.1.1 MPI_F90_DESCRIBE_OBJECT and relatives

```
MPI_F90_DESCRIBE_OBJECT(IN object, OUT MPI_Datatype)
```

This is a generic function that can be used with any Fortran 90 object, including array sections, derived types, etc. It creates a committed MPI datatype that describes the Fortran 90 object. The datatype must be used in conjunction with a buf argument of MPI_BOTTOM.

Discussion: Need to define the type signature of the new datatype, for matching rules.

It could be used in the following way

```
MPI_F90_DESCRIBE_OBJECT(a(10:1:-2), newtype)
MPI_ISEND(MPI_BOTTOM, 1, newtype, ...)
```

MPI_F90_DESCRIBE_OBJECT is an intrinsic function. Moreover, it does not have copy-in/copy-out semantics. The address and type information recorded in the output datatype describes the data in the original object, not the data in a temporary copy. It is not possible to implement this routine portably in Fortran 90. It must be closely integrated with the compiler. It is expected that this routine and other MPI routines would be part of an intrinsic module.

Rationale. In many applications, you send the same data many times. Creating the datatype once means you only pay the overhead once, instead of at each send call. (*End of rationale.*)

We could also consider a routine for which the created datatype is relative:

```
MPI_F90_DESCRIBE_OBJECT(a(10:1:-2), newtype)
MPI_ADDRESS(a, a_address)
MPI_ISEND(a_address, 1, newtype, ...)
```

[I don't propose this because I'm not sure we can say clearly when/if this works for a second array b, or even if you were to try:

```
MPI_ADDRESS(a(5), a5_address) ...
```

] In order for this to be guaranteed to work, it is probably necessary to make MPI_ADDRESS be an intrinsic function as well. It might be possible to allow both approaches through two calls:

```
MPI_F90_DESCRIBE_OBJECT(...)
MPI_F90_DESCRIBE_OBJECT_RELATIVE(...)
```

The following approach suggested in subcommittee discussion might provide a cleaner separation between MPI and the compiler.

```
MPI_F90_GET_DOPEVECTOR(a(10:2:-2), OUT dope)
MPI_IRECV(dope, 1, MPI_REAL)
```

This needs to be fleshed out.

16.2 Using a Fortran 90 - ANSI C interface

Some work is currently being done on an interface between Fortran 90 and ANSI C. MPI may be able to use this.

16.3 Typed functions

Here are some possibilities for using typed functions:

1. Require overloaded interfaces for all functions with choice arguments.
2. Add typed functions for "the most important" basic types, for instance MPI_SEND_REAL.
3. Add a "typed version" of all functions with choice arguments, e.g. MPI_SEND_T.
4. Have two versions of the MODULE MPI – one "light" version and one with full generic interfaces. Same name of the module, but mpi.mod files stored in different MODPATHs. Users can switch between safe+debug and fast mode by changing only one Makefile line: the search path for module files (library may contain code for both versions in one ar file...).

16.4 The possibility of optional arguments

For any routines with an explicit interface, it would be possible to have optional arguments. Clear candidates are STATUS and IERR arguments. Datatype arguments could also be omitted and the data derived from the type of buf.

This proposal could work in conjunction with one of the typed-function proposals above (but not with implicit F77-style interfaces, for which optional arguments are not allowed).

16.5 Derived types for MPI handles

Type-checking enabled by explicit interfaces might not be too useful because almost all of the arguments to MPI routines are integers in Fortran. It would be possible (losing all compatibility with Fortran 77) to introduce derived types for MPI objects, substantially improving the usefulness of type checking (though it is unclear that there would be any other benefits).

Bibliography

[1] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, 1993.

[2] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[3] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison Wesley, 1990.

[4] D. Ferrari. A new admission control method for real-time communication in an Internetwork. In D. Son, editor, *Advances in Real-Time Systems.* Prentice Hall, 1994. Chapter 5.

[5] C++ Forum. Working paper for draft proposed international standard for information systems – programming language c++. Technical report, American National Standards Institute, 1995.

[6] Message Passing Interface Forum. Document for a standard message-passing interface. Technical Report Technical Report No. CS-93-214 (revised), University of Tennessee, April 1994. Available on **netlib**.

[7] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *The International Journal of Supercomputer Applicatons and High Performance Computing*, 8, 1994.

[8] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical Report Computer Science Department Technical Report CS-94-230, University of Tennessee, Knoxville, TN, May 5 1994. To appear in the International Journal of Supercomputing Applications, Volume 8, Number 3/4, 1994.

[9] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Bob Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Network Parallel Computing.* MIT Press, 1994.

[10] S. Jamin, S. Shenker, L. Zhang, and D. D. Clark. An admission control algorithm for predictive real-time service. In *Proc. Third International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 349—356. Springer-Verlag, 1992. Heidelberg, Germany.

[11] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook.* MIT Press, 1993.

[12] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.

[13] A. A. Lazar and G. Pacifici. Control of resources in broadband networks with quality of service guarantees. *IEEE Communications*, 29(10):66—73, October 1991.

[14] Bill Nitzberg. Performance of the iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, December 1992.

[15] William J. Nitzberg. *Collective Parallel I/O.* PhD thesis, Department of Computer and Information Science, University of Oregon, December 1995.

[16] Perry Partow and Dennis Cottel. Scalable Programming Environment. Technical Report 1672, Naval Command Control and Ocean Surveillance Center (NRAD), September 1994.

[17] David L. Ripps. *An Implementation Guide to Real-time Programming.* Yourdon Press, Englewood Cliffs, N.J., 1990.

[18] 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.

[19] Anthony Skjellum, Nathan E. Doss, and Kishore Viswanathan. Inter-communicator extensions to MPI in the MPIX (MPI eXtension) Library. Technical report, Mississippi State University — Dept. of Computer Science, April 1994. Draft version.

[20] Rajeev Thakur and Alok Choudhary. An extended two-phase method for accessing sections of out-of-core arrays. Technical Report CACR-103, Scalable I/O Initiative, Center for Advanced Computing Research, Caltech, June 1995. Revised November 1995.

Annex A

Language Binding

A.1 Introduction

This appendix summarizes the specific bindings for both Fortran and C. First the constants, error codes, and info keys are presented. Second, the MPI-1.2 bindings are given. Third, the MPI-2 binding are given.

A.2 Defined Constants, Error Codes, and Info Keys

A.2.1 Constants

MPI_ADDRESS_KIND
MPI_ADDRESS_SIZE
MPI_APPNUM
MPI_ARGV_NULL
MPI_BYTE
MPI_COMM_CANONICAL
MPI_COMM_HANDLE
MPI_COMM_INTERNAL
MPI_COMM_PARENT
MPI_CONTIG
MPI_CREATE
MPI_C_ORDER
MPI_DATATYPE_HANDLE
MPI_DATA_CANONICAL
MPI_DATA_INTERNAL
MPI_DATA_NATIVE
MPI_DELETE_ON_CLOSE
MPI_DISTRIBUTE_BLOCK
MPI_DISTRIBUTE_CYCLIC
MPI_DISTRIBUTE_DFLT_ARG
MPI_DISTRIBUTE_NONE
MPI_ERRCODES_IGNORE
MPI_EXCL
MPI_FILE_HANDLE

MPI_FILE_NULL
MPI_FORTRAN_ORDER
MPI_FUNDAMENTAL
MPI_GROUP_HANDLE
MPI_HINDEXED
MPI_HVECTOR
MPI_IN-PLACE
MPI_INDEXED
MPI_INFO_MAX_KEYLEN
MPI_INFO_MAX_VALUELEN
MPI_INFO_NULL
MPI_INTEGER_KIND
MPI_IO
MPI_LASTUSERCODE
MPI_LOCK_EXCLUSIVE
MPI_LOCK_SHARED
MPI_MAX_NAME_LEN
MPI_MAX_NAME_STRING
MPI_NOCHECK
MPI_NOPUT
MPI_NOSTORE
MPI_NULL_PUSH_FN
MPI_RDONLY
MPI_RDWR
MPI_REPLACE
MPI_REQUEST_GENERALIZED
MPI_REQUEST_HANDLE
MPI_REQUEST_OTHER
MPI_REQUEST_PERSISTENT
MPI_REQUEST_PT2PT
MPI_ROOT
MPI_SEEK_CUR
MPI_SEEK_END
MPI_SEEK_SET
MPI_STRUCT
MPI_THREADS
MPI_UNIVERSE_SIZE
MPI_VECTOR
MPI_WIN_HANDLE
MPI_WIN_NULL
MPI_WRONLY
PTHREAD_CREATE_DETACHED
PTHREAD_CREATE_UNDETACHED
PTHREAD_EXPLICIT_SCHED
PTHREAD_INHERIT_SCHED
PTHREAD_SCOPE_PROCESS
PTHREAD_SCOPE_SYSTEM
SCHED_FIFO

SCHED_OTHER
SCHED_RR

A.2.2 Error Codes

MPI_ERR_ACCESS
MPI_ERR_BAD_FILE
MPI_ERR_EOF
MPI_ERR_FILE_EXISTS
MPI_ERR_FILE_NOT_DELETED
MPI_ERR_FILE
MPI_ERR_GR
MPI_ERR_INFO_NOKEY
MPI_ERR_INFO_VALUE
MPI_ERR_IO
MPI_ERR_NAME
MPI_ERR_NOPUBLISHING
MPI_ERR_NOSTRING
MPI_ERR_NOT_SAME
MPI_ERR_NO_SPACE
MPI_ERR_NO_SUCH_FILE
MPI_ERR_PORT
MPI_ERR_QUOTA
MPI_ERR_READ_ONLY
MPI_ERR_REQUEST
MPI_ERR_SERVICE
MPI_ERR_SPAWN

A.2.3 Info Keys

C_order
FORTRAN_order
access_style
appnum
arch
cb_block_size
cb_buffer_size
cb_nodes
chunked_primary
chunked_secondary
chunked_size
chunked
collective_buffering
column_major
false
file_perm

filename
file
host
io_node_list
ip_address
ip_port
nb_proc
num_io_nodes
path
pthread_contentionscope
pthread_detachstate
pthread_inheritsched
pthread_priority
pthread_schedpolicy
pthread_stackaddr
pthread_stacksize
random
read_mostly
read_once
reverse_sequential
row_major
sequential
soft
striping_factor
striping_unit
true
wdir
write_mostly
write_once

A.3 MPI-1.2 C bindings

```
int MPI_Get_version( int *version, int *subversion )
```

A.4 MPI-1.2 Fortran bindings

```
MPI_GET_VERSION( VERSION, SUBVERSION, IERROR )
    INTEGER VERSION, SUBVERSION, IERROR
```

A.5 MPI-2 C bindings

A.5.1 Process Creation and Management

```
int MPI_Accept(char port_name[], MPI_Info info, int root, MPI_Comm comm,
              MPI_Comm* newcomm)
```

```
int MPI_Comm_disconnect(MPI_Comm* comm)

int MPI_Connect(char port_name[], MPI_Info::info, int root, MPI_Comm comm,
             MPI_Comm* newcomm)

int MPI_Iaccept(char port_name[], MPI_Info info, int root, MPI_Comm comm,
             MPI_Comm* newcomm, MPI_Request* request)

int MPI_Iconnect(char port_name[], MPI_Info info, int root, MPI_Comm comm,
             MPI_Comm* newcomm, MPI_Request* request)

int MPI_Info_create(MPI_Info *info)

int MPI_Info_delete(MPI_Info info, char key[])

int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo)

int MPI_Info_free(MPI_Info* info)

int MPI_Info_get(MPI_Info info, char  key[], int valuelen, char value[],
             int* flag)

int MPI_Info_get_nkeys(MPI_Info info, int* nkeys)

int MPI_Info_get_nthkey(MPI_Info info, int n, char key[])

int MPI_Info_get_valuelen(MPI_Info info,char key[],int* valuelen,int* flag)

int MPI_Info_set(MPI_Info info, char key[], char value[])

int MPI_Ispawn(char command[], char* argv[], int maxprocs, MPI_Info info,
             int root, MPI_Comm comm, MPI_Comm* intercomm,
             int array_of_errcodes[], MPI_Request* request)

int MPI_Ispawn_multiple(int count, char* array_of_commands[],
             char** array_of_argv[], int array_of_maxprocs[],
             MPI_Info array_of_info[], int root, MPI_Comm comm,
             MPI_Comm* intercomm, int array_of_errcodes[],
             MPI_Request *request)

int MPI_Name_get(char service_name[], MPI_Info info, char port_name[])

int MPI_Name_publish(char port_name[], MPI_Info info, char service_name[])

int MPI_Name_unpublish(char port_name[], char service_name[])

int MPI_Port_close(char port_name[])

int MPI_Port_open(MPI_Info info, char port_name[])

int MPI_Spawn(char command[], char* argv[], int maxprocs, MPI_Info info,
             int root, MPI_Comm comm, MPI_Comm* intercomm,
             int array_of_errcodes[])

int MPI_Spawn_multiple(int count, char* array_of_commands[],
             char** array_of_argv[], int array_of_maxprocs[],
             MPI_Info array_of_info[], int root, MPI_Comm comm,
```

```
            MPI_Comm* intercomm, int array_of_errcodes[])
```

A.5.2 One-Sided Communications

```
int MPI_Accumulate(void* origin_addr, int origin_count,
            MPI_Datatype origin_datatype, int target_rank, int target_disp,
            int target_count, MPI_Datatype target_datatype, MPI_Op op,
            MPI_Win win)

int MPI_Mem_alloc(int size, MPI_Comm comm, void *base)

int MPI_Mem_free(void *base)

int MPI_Put( void *origin_addr, int origin_count, MPI_Datatype
            origin_datatype, int target_rank, int target_disp, int
            target_count, MPI_Datatype target_datatype, MPI_Win win)

int MPI_Win_Lock(int lock_type, int rank, int assert, MPI_Win win)

int MPI_Win_barrier( int assert, MPI_Win win)

int MPI_Win_complete(MPI_Win win)

int MPI_Win_free(MPI_Win *win)

int MPI_Win_init(void* base, int size, int disp_unit, MPI_Info info,
            MPI_Comm comm, MPI_Win *win)

int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)

int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)

int MPI_Win_test(MPI_Win win, int *flag)

int MPI_Win_unlock(int rank, MPI_Win win)

int MPI_Win_wait(MPI_Win win)

int MPI_get( void *origin_addr, int origin_count, MPI_Datatype
            origin_datatype, int target, int target_disp, int
            target_count, MPI_Datatype target_datatype, MPI_Win win)
```

A.5.3 Extended Collective Operations

```
int MPI_Alltoallw(void* sendbuf, int* sendcounts,
            int* sdispls,MPI_Datatype* sendtypes, void* recvbuf,
            int* recvcounts, int* rdispls,MPI_Datatype* recvtypes,
            MPI_Comm comm)

int MPI_Barrier(MPI_Comm comm)

int MPI_Barrier_init(MPI_Comm comm, MPI_Request *request)

int MPI_Bcast_init(void *buffer, int count, MPI_Datatype datatype, int root,
            MPI_Comm comm, MPI_Request *request)
```

```
int MPI_Bind(MPI_Request *request)

int MPI_Exscan(void *sendbuf, void *recvbuf, int count,
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

A.5.4 External Interfaces

```
int MPI_Add_error_class(int *errorclass)

int MPI_Add_error_codes(int errorclass, int *array_errorcodes, int count)

int MPI_Add_error_string(int errorcode, char *string, int len)

int MPI_Do(MPI_Request *request, MPI_Status *status)

int MPI_Get_type_contents(MPI_Datatype datatype, int start, int count,
              int *array_blocklens, int *array_disps, MPI_Aint *array_hdisps,
              MPI_Datatype *array_types)

int MPI_Get_type_envelope(MPI_Datatype datatype, int* count, int* combiner)

int MPI_Meta_request_create(MPI_Start_function *start_fn, MPI_Push_function
              *push_fn, MPI_Complete_function *complete_fn,
              MPI_Cancel_function *cancel_fn, void *extra_state, MPI_Request
              *meta_req)

int MPI_Meta_request_free(MPI_Request *meta_req)

int MPI_Meta_request_init(MPI_Request meta_req, MPI_Request *request)

int MPI_Name_get(MPI_Comm comm, char **name)

int MPI_Name_put(MPI_Comm comm, char *name)

int MPI_Post_handler(MPI_Request request, MPI_Info info,
              MPI_Request_handler handler, void *extra_state, int
              handler_type)

int MPI_Request_mark_complete(MPI_Request *request)

int MPI_Request_type(MPI_Request request, int *type_req)

int MPI_Set_elements_status(int count, MPI_Datatype datatype, MPI_Status
              *status)

int MPI_Thread_finalize()

int MPI_Thread_init()

int MPI_Thread_initialized(int *flag)

int MPI_Type_compare(MPI_Datatype datatype1, MPI_Datatype datatype2, int
              *flag)

int MPI_Type_decode(void *buffer, int buffsize, int *len, MPI_Datatype
              *datatype, MPI_Comm comm)
```

```
int MPI_Type_encode(MPI_Datatype datatype, void *buffer, int buffsize, int
              *len, MPI_Comm comm)

int MPI_Type_encode_size(MPI_Datatype datatype, int *maxlen, MPI_Comm comm)
```

A.5.5 Miscellany

```
int MPI_Address_x(void* location, MPI_Aint *address)

int MPI_Attr_get_x(MPI_Comm comm, int keyval, void *attribute_val)

int MPI_Attr_put_x(MPI_Comm comm, int keyval, void* attribute_val)

int MPI_Err_is_continuable( int error, int *flag )

int MPI_Finalized(int *flag)

int MPI_Get_extent_x(MPI_Datatype datatype, MPI_Aint *lb, MPI_Aint *extent)

int MPI_Get_status(MPI_Request request, int *flag, MPI_Status *status)

int MPI_Keyval_create_x(MPI_Copy_function *copy_fn, MPI_delete_function
              *delete_fn, int *keyval, void *extra_state

int MPI_True_extent_x(MPI_Datatype datatype, MPI_Aint *true_lb, MPI_Aint
              *true_extent)

int MPI_Type_Indexed_Block( int count, int blocklength,
              int* array_of_displacements, MPI_Datatype oldtype,
              MPI_Datatype* newtype )

int MPI_Type_hindexed_x(int count, int *array_of_blocklengths,
              MPI_Aint *array_of_displacements, MPI_Datatype oldtype,
              MPI_Datatype *newtype)

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

int MPI_Type_resized_x(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint extent,
              MPI_Datatype *newtype)

int MPI_Type_struct_x(int count, int *array_of_blocklengths,
              MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types,
              MPI_Datatype *newtype)

MPI::<CLASS>(const MPI_<CLASS>* data)

MPI::<CLASS>(const MPI_<CLASS>& data)

MPI::<CLASS>::operator MPI_<CLASS>() const

MPI::<CLASS>::operator MPI_<CLASS>*()

MPI::<CLASS>& MPI::<CLASS>::operator=(const MPI_<CLASS>& data)

MPI_C2F_Status( MPI_Status *cstatus, MPI_Fint *f_status)
```

```
MPI_F2C_status( MPI_Fint *f_status, MPI_Status *cstatus)

MPI_Fint MPI_C2F_handle( MPI_Handle c_handle, int handle_kind)

MPI_Handle MPI_F2C_handle( MPI_Fint f_handle, int handle_kind)
```

A.5.6 I/O

```
int MPI_Close(MPI_File *fh)

int MPI_Delete(char *filename)

int MPI_File_get_amode(MPI_File fh, int *amode)

int MPI_File_get_atomicity(MPI_File fh, int *flag)

int MPI_File_get_comm(MPI_File fh, MPI_Comm *comm)

int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)

int MPI_File_get_position(MPI_File fh, MPI_Offset *offset,
             MPI_Offset *byte_offset)

int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset,
             MPI_Offset *byte_offset)

int MPI_File_get_size(MPI_File fh, MPI_Offset *size)

int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype,
             MPI_Aint *extent)

int MPI_File_get_type_size(MPI_File fh, MPI_Datatype datatype,
             MPI_Aint *size)

int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype,
             MPI_Datatype *filetype)

int MPI_File_set_atomicity(MPI_File fh, int flag)

int MPI_File_set_info(MPI_File fh, MPI_Info info)

int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
             MPI_Datatype filetype, MPI_Info info)

int MPI_File_sync(MPI_File fh)

int MPI_Iclose(MPI_File *fh, MPI_Request *request)

int MPI_Iopen(MPI_Comm comm, char *filename, int amode, MPI_Info info,
             MPI_File *fh, MPI_Request *request)

int MPI_Iread(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
             MPI_Request *request)

int MPI_Iread_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
             MPI_Request *request)
```

```
int MPI_Iread_explicit(MPI_File fh, MPI_Offset offset, void *buf, int count,
            MPI_Datatype datatype, MPI_Request *request)

int MPI_Iread_explicit_all(MPI_File fh, MPI_Offset offset, void *buf,
            int count, MPI_Datatype datatype, MPI_Request *request)

int MPI_Iread_shared(MPI_File fh, void *buf, int count,
            MPI_Datatype datatype, MPI_Request *request)

int MPI_Iread_shared_ordered(MPI_File fh, void *buf, int count,
            MPI_Datatype datatype, MPI_Request *request)

int MPI_Iwrite(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
            MPI_Request *request)

int MPI_Iwrite_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
            MPI_Request *request)

int MPI_Iwrite_explicit(MPI_File fh, MPI_Offset offset, void *buf,
            int count, MPI_Datatype datatype, MPI_Request *request)

int MPI_Iwrite_explicit_all(MPI_File fh, MPI_Offset offset, void *buf,
            int count, MPI_Datatype datatype, MPI_Request *request)

int MPI_Iwrite_shared(MPI_File fh, void *buf, int count,
            MPI_Datatype datatype, MPI_Request *request)

int MPI_Iwrite_shared_ordered(MPI_File fh, void *buf, int count,
            MPI_Datatype datatype, MPI_Request *request)

int MPI_Open(MPI_Comm comm, char *filename, int amode, MPI_Info info,
            MPI_File *fh)

int MPI_Read(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
            MPI_Status *status)

int MPI_Read_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
            MPI_Status *status)

int MPI_Read_explicit(MPI_File fh, MPI_Offset offset, void *buf, int count,
            MPI_Datatype datatype, MPI_Status *status)

int MPI_Read_explicit_all(MPI_File fh, MPI_Offset offset, void *buf,
            int count, MPI_Datatype datatype, MPI_Status *status)

int MPI_Read_shared(MPI_File fh, void *buf, int count,
            MPI_Datatype datatype, MPI_Status *status)

int MPI_Read_shared_ordered(MPI_File fh, void *buf, int count,
            MPI_Datatype datatype, MPI_Status *status)

int MPI_Resize(MPI_File fh, MPI_Offset size)

int MPI_Seek(MPI_File fh, MPI_Offset offset, int whence)

int MPI_Seek_shared(MPI_File fh, MPI_Offset offset, int whence)
```

```
int MPI_Type_darray(int size, int rank, int ndims, int *dsizes, int
             *distribs, int *dargs, int pdims, int *psizes, int order,
             MPI_Datatype oldtype, MPI_Datatype *newtype)

int MPI_Type_darray_block(int size, int rank, int dsize, int darg,
             MPI_Datatype oldtype, MPI_Datatype *newtype)

int MPI_Type_darray_cyclic(int size, int rank, int dsize, int darg,
             MPI_Datatype oldtype, MPI_Datatype *newtype)

int MPI_Type_scatter_gather(int size, int rank, MPI_Datatype oldtype,
             MPI_Datatype *newtype)

int MPI_Type_scatterv_gatherv(int size, int rank, int count,
             MPI_Datatype oldtype, MPI_Datatype *newtype)

int MPI_Type_subarray(int ndims, int *sizes, int *subsizes, int *starts,
             int order, MPI_Datatype oldtype, MPI_Datatype *newtype)

int MPI_Write(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
             MPI_Status *status)

int MPI_Write_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
             MPI_Status *status)

int MPI_Write_explicit(MPI_File fh, MPI_Offset offset, void *buf, int count,
             MPI_Datatype datatype, MPI_Status *status)

int MPI_Write_explicit_all(MPI_File fh, MPI_Offset offset, void *buf,
             int count, MPI_Datatype datatype, MPI_Status *status)

int MPI_Write_shared(MPI_File fh, void *buf, int count,
             MPI_Datatype datatype, MPI_Status *status)

int MPI_Write_shared_ordered(MPI_File fh, void *buf, int count,
             MPI_Datatype datatype, MPI_Status *status)
```

A.6 MPI-2 C functions

```
typedef int MPI_Push_fn(void *extra_state, MPI_Request request, int *done);

typedef int MPI_Request_cancel_fn(void *extra_state, MPI_Request request);

typedef int MPI_Request_complete_fn(void *extra_state, MPI_Request request,
             MPI_Status *status);

typedef int MPI_Request_handler(MPI_Request *request, void *extra_state);

typedef int MPI_Start_fn(void *extra_state, MPI_Request request);
```

A.7 MPI-2 Fortran bindings

A.7.1 Process Creation and Management

```
MPI_ACCEPT(PORT_NAME, ROOT, COMM, NEWCOMM, IERR)
    CHARACTER*(*) PORT_NAME
    INTEGER COMM, ROOT, NEWCOMM, IERR

MPI_COMM_DISCONNECT(COMM, IERR)
    INTEGER COMM, IERR

MPI_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERR)
    CHARACTER*(*) PORT_NAME, INFO
    INTEGER ROOT, COMM, NEWCOMM, IERR

MPI_IACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, REQUEST, IERR)
    CHARACTER*(*) PORT_NAME
    INTEGER ROOT, INFO, COMM, NEWCOMM, REQUEST, IERR

MPI_ICONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, REQUEST, IERR)
    CHARACTER*(*) PORT_NAME,
    INTEGER ROOT, INFO, COMM, NEWCOMM, REQUEST, IERR

MPI_INFO_CREATE(INFO, IERR)
    INTEGER INFO, IERR

MPI_INFO_DELETE(INFO, KEY, IERR)
    INTEGER INFO, IERR
    CHARACTER*(*) KEY

MPI_INFO_DUP(INFO, NEWINFO, IERR)
    INTEGER INFO, NEWINFO, IERR

MPI_INFO_FREE(INFO, IERR)
    INTEGER INFO, IERR

MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERR)
    INTEGER INFO, VALUELEN, IERR
    CHARACTER*(*) KEY, VALUE
    LOGICAL FLAG

MPI_INFO_GET_NKEYS(INFO, NKEYS, IERR)
    INTEGER INFO, NKEYS, IERR

MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERR)
    INTEGER INFO, N, IERR
    CHARACTER*(*) KEY

MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERR)
    INTEGER INFO, VALUELEN, IERR
    LOGICAL FLAG
    CHARACTER*(*) KEY

MPI_INFO_SET(INFO, KEY, VALUE, IERR)
```

```
    INTEGER INFO, IERR
    CHARACTER*(*) KEY, VALUE

MPI_ISPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,
             ARRAY_OF_ERRCODES, REQUEST, IERR)
    CHARACTER*(*) COMMAND, ARGV(*)
    INTEGER INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*),
    REQUEST, IERR

MPI_ISPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
             ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,
             ARRAY_OF_ERRCODES, REQUEST, IERR)
    INTEGER COUNT, ARRAY_OF_INFO(*), ARRAY_OF_MAXPROCS(*), ROOT, COMM,
    INTERCOMM, ARRAY_OF_ERRCODES(*), REQUEST, IERR
    CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)

MPI_NAME_GET(SERVICE_NAME, INFO, PORT_NAME, IERR)
    CHARACTER*(*) SERVICE_NAME
    CHARACTER*(*) PORT_NAME
    INTEGER INFO, IERR

MPI_NAME_PUBLISH(PORT_NAME, INFO, SERVICE_NAME, IERR)
    INTEGER INFO, IERR
    CHARACTER*(*) PORT_NAME, SERVICE_NAME

MPI_NAME_UNPUBLISH(PORT_NAME, SERVICE_NAME, IERR)
    INTEGER IERR
    CHARACTER*(*) PORT_NAME, SERVICE_NAME

MPI_PORT_CLOSE(PORT_NAME, IERR)
    CHARACTER*(*) PORT_NAME
    INTEGER IERR

MPI_PORT_OPEN(INFO, PORT_NAME, IERR)
    CHARACTER*(*) PORT_NAME
    INTEGER INFO, IERR

MPI_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,
             ARRAY_OF_ERRCODES, IERR)
    CHARACTER*(*) COMMAND, ARGV(*)
    INTEGER INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*),
    IERR

MPI_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
             ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,
             ARRAY_OF_ERRCODES, IERR)
    INTEGER COUNT, ARRAY_OF_INFO(*), ARRAY_OF_MAXPROCS(*), ROOT, COMM,
    INTERCOMM, ARRAY_OF_ERRCODES(*), IERR
    CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)
```

A.7.2 One-Sided Communications

```
MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
             TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP,
    TARGET_DATATYPE, OP, WIN, IERROR

MPI_GET( ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET, TARGET_DISP,
             TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET, TARGET_DISP,
    TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR

MPI_MEM_ALLOC( SIZE, COMM, BASE, IERROR)
    INTEGER SIZE, COMM, IERROR
    INTEGER(KIND=MPI_ADDRESS_SIZE) BASE

MPI_MEM_FREE( BASE, IERROR)
    INTEGER(KIND=MPI_ADDRESS_SIZE) BASE

MPI_PUT( ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
             TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP,
    TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR

MPI_WIN_BARRIER(ASSERT, WIN, IERROR)
    INTEGER ASSERT, WIN, IERROR

MPI_WIN_COMPLETE(WIN, IERROR)
    INTEGER WIN, IERROR

MPI_WIN_FREE(WIN, IERROR)
    INTEGER WIN, IERROR

MPI_WIN_INIT(BASE, SIZE, DISP_UNIT, INFO, COMM, WIN, IERROR)
    <type> BASE(*)
    INTEGER SIZE, DISP_UNIT, INFO, COMM, WIN, IERROR

MPI_WIN_LOCK( LOCK_TYPE, RANK, ASSERT, WIN, IERROR)
    INTEGER RANK, LOCK_TYPE, RANK, ASSERT, WIN, IERROR

MPI_WIN_POST(GROUP, ASSERT, IERROR)
    INTEGER GROUP, ASSERT, WIN, IERROR

MPI_WIN_START(GROUP, ASSERT, WIN, IERROR)
    INTEGER GROUP, ASSERT, WIN, IERROR

MPI_WIN_TEST(WIN, FLAG, IERROR)
    INTEGER WIN, IERROR
    LOGICAL FLAG

MPI_WIN_UNLOCK(RANK, WIN, IERROR)
```

```
    INTEGER RANK, WIN, IERROR

MPI_WIN_WAIT(WIN, IERROR)
    INTEGER WIN, IERROR
```

A.7.3 Extended Collective Operations

```
MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS,
             RDISPLS, RECVTYPES, COMM)
    INTEGER SENDBUF, SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVBUF,
    RECVBUFS(*), RECVCOUNTS(*), RDISPLS(*), RECVTYPES(*), COMM, IERR

MPI_BARRIER_INIT(COMM, REQUEST, IERR)
    INTEGER COMM, REQUEST, IERR

MPI_BCAST_INIT(BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERR)
    INTEGER BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERR

MPI_BIND(REQUEST, IERR)
    INTEGER REQUEST, IERR

MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, IERR)
    INTEGER SENDBUF, RECVBUF, COUNT, DATATYPE, OP, IERR
```

A.7.4 External Interfaces

```
MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR)
    INTEGER ERRORCLASS, IERROR

MPI_ADD_ERROR_CODES(ERRORCLASS, ARRAY_ERRORCODES, COUNT, IERROR)
    INTEGER ERRORCLASS, ARRAY_ERRORCODES(*), COUNT, IERROR

MPI_ADD_ERROR_STRING(ERRORCODE, STRING, LEN, IERROR)
    INTEGER ERRORCODE, LEN, IERROR
    CHARACTER*(*) STRING

MPI_DO(REQUEST, STATUS, IERROR)
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

MPI_GET_TYPE_CONTENTS(DATATYPE, START, COUNT, ARRAY_BLOCKLENS, ARRAY_DISPS,
             ARRAY_HDISPS, ARRAY_TYPES, IERROR)
    INTEGER DATATYPE, START, COUNT, ARRAY_BLOCKLENS(*), ARRAY_DISPS(*),
    ARRAY_HDISPS(*), ARRAY_TYPES(*), IERROR

MPI_GET_TYPE_ENVELOPE(DATATYPE, COUNT, COMBINER, IERROR)
    INTEGER DATATYPE, COUNT, COMBINER, IERROR

MPI_META_REQUEST_CREATE(START_FN, PUSH_FN, COMPLETE_FN, CANCEL_FN,
             EXTRA_STATE, META_REQ, IERROR)
    EXTERNAL START_FN, PUSH_FN, COMPLETE_FN, CANCEL_FN
    INTEGER EXTRA_STATE, META_REQ, IERROR

MPI_META_REQUEST_FREE(META_REQ, IERROR)
```

```
    INTEGER META_REQ, IERROR

MPI_META_REQUEST_INIT(META_REQ, REQUEST, IERROR)
    INTEGER META_REQ, REQUEST, IERROR

MPI_NAME_GET(COMM, NAME, IERROR)
    INTEGER COMM
    CHARACTER*(*) NAME
    INTEGER IERROR

MPI_NAME_PUT(COMM, NAME, IERROR)
    INTEGER COMM
    CHARACTER*(*) NAME
    INTEGER IERROR

MPI_POST_HANDLER(REQUEST, INFO, HANDLER, EXTRA_STATE, HANDLER_TYPE, IERROR)
    INTEGER REQUEST, INFO, EXTRA_STATE, HANDLER_TYPE, IERROR
    PROCEDURE HANDLER

MPI_REQUEST_MARK_COMPLETE(REQUEST, IERROR)
    INTEGER REQUEST, IERROR

MPI_REQUEST_TYPE(REQUEST, TYPE_REQ, IERROR)
    INTEGER REQUEST, TYPE_REQ, IERROR

MPI_SET_ELEMENTS_STATUS(COUNT, DATATYPE, STATUS, IERROR)
    int COUNT, DATATYPE, STATUS, IERROR

MPI_THREAD_FINALIZE(IERROR)
    INTEGER IERROR

MPI_THREAD_INIT(IERROR)
    INTEGER IERROR

MPI_THREAD_INITIALIZED(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR

MPI_TYPE_COMPARE(DATATYPE1, DATATYPE2, FLAG)
    integer DATATYPE1, DATATYPE2
    LOGICAL FLAG

MPI_TYPE_DECODE(BUFFER, BUFFSIZE, LEN, DATATYPE, COMM, IERROR)
    <type> BUFFER(*)
    INTEGER BUFFSIZE, LEN, DATATYPE, COMM, IERROR

MPI_TYPE_ENCODE(DATATYPE, BUFFER, BUFFSIZE, LEN, COMM, IERROR)
    <type> BUFFER(*)
    INTEGER DATATYPE, BUFFSIZE, LEN, COMM, IERROR

MPI_TYPE_ENCODE_SIZE(DATATYPE, MAXLEN, COMM, IERROR)
    INTEGER DATATYPE, MAXLEN, COMM, IERROR
```

A.7.5 Miscellany

```
MPI_ADDRESS_X(LOCATION, ADDRESS, IERROR)
    <type> LOCATION(*)
    INTEGER IERROR
    INTEGER(KIND=MPI_ADDRESS_SIZE) ADDRESS

MPI_ATTR_GET_X(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER COMM, KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
    LOGICAL FLAG

MPI_ATTR_PUT_X(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER COMM, KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL

MPI_ERR_IS_CONTINUABLE( ERROR, FLAG, IERROR )
    INTEGER ERROR, IERROR
    LOGICAL FLAG

MPI_FINALIZED(flag, IERROR))
    FLAG, IERROR

MPI_GET_EXTENT_X(DATATYPE, LB, EXTENT,IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND = MPI_ADDRESS_SIZE) LB, EXTENT

MPI_GET_STATUS( REQUEST, FLAG, STATUS, IERROR)
    INTEGER REQUEST, FLAG, STATUS(MPI_STATUS_SIZE), IERROR

MPI_KEYVAL_CREATE_X(COPY_FN, DELETE_FN, KEYVAL, EXTRA STATE, IERROR)
    EXTERNAL COPY_FN, DELETE_FN
    INTEGER KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

MPI_TRUE_EXTENT_X(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
    INTEGER DATATYPE, IERROR
    INTGER(KIND = MPI_ADDRESS_SIZE) TRUE_LB, TRUE_EXTENT

MPI_TYPE_HINDEXED_X(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
             OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), OLDTYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_SIZE) ARRAY_OF_DISPLACEMENTS(*)

MPI_TYPE_HVECTOR_X(COUNT, BLOCKLENGTH, STIDE, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_SIZE) STRIDE

MPI_TYPE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS, OLDTYPE,
             NEWTYPE, IERROR)
    INTEGER COUNT, INTEGER BLOCKLENGTH, ARRAY_OF_DISPLACEMENT(*), OLDTYPE,
    NEWTYPE, IERROR

MPI_TYPE_RESIZED_X(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)
```

```
    INTEGER DATATYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_SIZE) LB, EXTENT

MPI_TYPE_STRUCT_X(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
             ARRAY_OF_TYPES, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_TYPES(*), NEWTYPE,
    IERROR
    INTEGER(KIND=MPI_ADDRESS_SIZE) ARRAY_OF_DISPLACEMENTS(*),
```

A.7.6 I/O

```
MPI_CLOSE(FH, IERROR)
    INTEGER FH, IERROR

MPI_DELETE(FILENAME, IERROR)
    CHARACTER FILENAME(*)
    INTEGER IERROR

MPI_FILE_GET_AMODE(FH, AMODE, IERROR)
    INTEGER FH, AMODE, IERROR

MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR)
    INTEGER FH, FLAG, IERROR

MPI_FILE_GET_COMM(FH, COMM, IERROR)
    INTEGER FH, COMM, IERROR

MPI_FILE_GET_INFO(FH, INFO_USED, IERROR)
    INTEGER FH, INFO_USED, IERROR

MPI_FILE_GET_POSITION(FH, OFFSET, BYTE_OFFSET, IERROR)
    INTEGER FH, IERROR
    INTEGER*8 OFFSET, BYTE_OFFSET

MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, BYTE_OFFSET, IERROR)
    INTEGER FH, IERROR
    INTEGER*8 OFFSET, BYTE_OFFSET

MPI_FILE_GET_SIZE(FH, SIZE, IERROR)
    INTEGER FH, IERROR
    INTEGER*8 SIZE

MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERROR)
    INTEGER FH, DATATYPE, IERROR
    INTEGER(KIND = MPI_ADDRESS_SIZE) EXTENT

MPI_FILE_GET_TYPE_SIZE(FH, DATATYPE, SIZE, IERROR)
    INTEGER FH, DATATYPE, IERROR
    INTEGER(KIND = MPI_ADDRESS_SIZE) SIZE

MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, IERROR)
    INTEGER FH, ETYPE, FILETYPE, IERROR
    INTEGER*8 DISP
```

```
MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR)
    INTEGER FH, FLAG, IERROR

MPI_FILE_SET_INFO(FH, INFO, IERROR)
    INTEGER FH, INFO, IERROR

MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, INFO, IERROR)
    INTEGER FH, ETYPE, FILETYPE, INFO, IERROR
    INTEGER*8 DISP

MPI_FILE_SYNC(FH, IERROR)
    INTEGER FH, IERROR

MPI_ICLOSE(FH, REQUEST, IERROR)
    INTEGER FH, REQUEST, IERROR

MPI_IOPEN(COMM, FILENAME, AMODE, INFO, FH, REQUEST, IERROR)
    CHARACTER FILENAME(*)
    INTEGER COMM, AMODE, INFO, FH, REQUEST, IERROR

MPI_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR

MPI_IREAD_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR

MPI_IREAD_EXPLICIT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
    INTEGER*8 OFFSET

MPI_IREAD_EXPLICIT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
    INTEGER*8 OFFSET

MPI_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR

MPI_IREAD_SHARED_ORDERED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR

MPI_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR

MPI_IWRITE_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
```

```
MPI_IWRITE_EXPLICIT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
    INTEGER*8 OFFSET

MPI_IWRITE_EXPLICIT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
    INTEGER*8 OFFSET

MPI_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR

MPI_IWRITE_SHARED_ORDERED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR

MPI_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR)
    CHARACTER FILENAME(*)
    INTEGER COMM, AMODE
    INTEGER INFO, FH, IERROR

MPI_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_READ_EXPLICIT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER*8 OFFSET

MPI_READ_EXPLICIT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER*8 OFFSET

MPI_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_READ_SHARED_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_RESIZE(FH, SIZE, IERROR)
    INTEGER FH, IERROR
    INTEGER*8 SIZE
```

```
MPI_SEEK(FH, OFFSET, WHENCE, IERROR)
    INTEGER FH, WHENCE, IERROR
    INTEGER*8 OFFSET

MPI_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR)
    INTEGER FH, WHENCE, IERROR
    INTEGER*8 OFFSET

MPI_TYPE_DARRAY(SIZE, RANK, NDIMS, DSIZES, DISTRIBS, DARGS, PDIMS, PSIZES,
             ORDER, OLDTYPE, NEWTYPE, IERROR)
    INTEGER SIZE, RANK, NDIMS, DSIZES(*), DISTRIBS(*), DARGS(*), PDIMS,
    PSIZES(*)
    INTEGER ORDER, OLDTYPE, NEWTYPE, IERROR

MPI_TYPE_DARRAY_BLOCK(SIZE, RANK, DSIZE, DARG, OLDTYPE, NEWTYPE, IERROR)
    INTEGER SIZE, RANK, DSIZE, DARG, OLDTYPE, NEWTYPE, IERROR

MPI_TYPE_DARRAY_CYCLIC(SIZE, RANK, DSIZE, DARG, OLDTYPE, NEWTYPE, IERROR)
    INTEGER SIZE, RANK, DSIZE, DARG, OLDTYPE, NEWTYPE, IERROR)

MPI_TYPE_SCATTERV_GATHERV(SIZE, RANK, COUNT, OLDTYPE, NEWTYPE, IERROR)
    INTEGER SIZE, RANK, COUNT, OLDTYPE, NEWTYPE, IERROR

MPI_TYPE_SCATTER_GATHER(SIZE, RANK, OLDTYPE, NEWTYPE, IERROR)
    INTEGER SIZE, RANK, OLDTYPE, NEWTYPE, IERROR

MPI_TYPE_SUBARRAY(NDIMS, SIZES, SUBSIZES, STARTS, ORDER, OLDTYPE, NEWTYPE,
             IERROR)
    INTEGER NDIMS, SIZES(*), SUBSIZES(*), STARTS(*), ORDER, OLDTYPE,
    NEWTYPE, IERROR

MPI_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_WRITE_EXPLICIT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER*8 OFFSET

MPI_WRITE_EXPLICIT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER*8 OFFSET

MPI_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_WRITE_SHARED_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
```

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

A.8 MPI-2 Fortran Subroutines

```
SUBROUTINE PUSH_FN(EXTRA_STATE, REQUEST, DONE, IERR)
    INTEGER EXTRA_STATE, REQUEST, IERR//LOGICAL DONE

SUBROUTINE REQUEST_CANCEL_FN(EXTRA_STATE, REQUEST, IERR)
    INTEGER EXTRA_STATE, REQUEST, IERR

SUBROUTINE REQUEST_COMPLETE_FN(EXTRA_STATE, REQUEST, STATUS, IERR)
    INTEGER EXTRA_STATE, META_REQ, STATUS(MPI_STATUS_SIZE

SUBROUTINE REQUEST_HANDLER( REQUEST, EXTRA_STATE, IERR)
    INTEGER REQUEST, EXTRA_STATE, IERR

SUBROUTINE START_FN(EXTRA_STATE, REQUEST, IERR)
    INTEGER EXTRA_STATE, REQUEST, IERR
```

Annex B

MPI-1 C++ Language Binding - Proposal 1

Current Status: No votes

Discussion:
This is the "old" C++ bindings proposal.

B.1 C++ Classes

The following are the classes provided with the C++ MPI-1 language bindings:

```
namespace MPI {
  class Comm {...};
  class Datatype {...};
  class Errhandler {...};
  class Group {...};
  class Op {...};
  class Request {...};
  class Status {...};
}
```

B.2 Defined Constants

These are required constants, defined in the file `mpi++.h`. For brevity, the types of the constants are defined below are defined in the comments.

```
// return codes
// Type: const int (or enum)
MPI::SUCCESS
MPI::ERR_BUFFER
MPI::ERR_COUNT
MPI::ERR_TYPE
```

```
MPI::ERR_TAG
MPI::ERR_COMM
MPI::ERR_RANK
MPI::ERR_REQUEST
MPI::ERR_ROOT
MPI::ERR_GROUP
MPI::ERR_OP
MPI::ERR_TOPOLOGY
MPI::ERR_DIMS
MPI::ERR_ARG
MPI::ERR_UNKNOWN
MPI::ERR_TRUNCATE
MPI::ERR_OTHER
MPI::ERR_INTERN
MPI::ERR_PENDING
MPI::ERR_IN_STATUS
MPI::ERR_LASTCODE
// assorted constants
// Type: const MPI::Aint
MPI::BOTTOM
// Type: const int (or enum)
MPI::PROC_NULL
MPI::ANY_SOURCE
MPI::ANY_TAG
MPI::UNDEFINED
MPI::BSEND_OVERHEAD
MPI::KEYVAL_INVALID

// Error-handling specifiers
// Type: const int (or enum)
MPI::ERRORS_ARE_FATAL
MPI::ERRORS_RETURN

// Maximum sizes for strings
// Type: const int
MPI::MAX_PROCESSOR_NAME
MPI::MAX_ERROR_STRING

// elementary datatypes (C / C++)
// Type: const MPI::Datatype
MPI::CHAR
MPI::SHORT
MPI::INT
MPI::LONG
MPI::UNSIGNED_CHAR
MPI::UNSIGNED_SHORT
MPI::UNSIGNED
MPI::UNSIGNED_LONG
```

```
MPI::FLOAT
MPI::DOUBLE
MPI::LONG_DOUBLE
MPI::BYTE
MPI::PACKED

// elementary datatypes (Fortran 77)
// Type: const MPI::Datatype
MPI::INTEGER
MPI::REAL
MPI::DOUBLE_PRECISION
MPI::COMPLEX
MPI::LOGICAL
MPI::CHARACTER
MPI::BYTE

// datatypes for reduction functions (C / C++)
// Type: const MPI::Datatype
MPI::FLOAT_INT
MPI::DOUBLE_INT
MPI::LONG_INT
MPI::TWOINT
MPI::SHORT_INT
MPI::LONG_DOUBLE_INT

// datatype for reduction functions (Fortran 77)
// Type const MPI::Datatype
MPI::TWOREAL
MPI::TWODOUBLE_PRECISION
MPI::TWOINTEGER

// optional datatypes (Fortran 77)
// Type: const MPI::Datatype
MPI::INTEGER1
MPI::INTEGER2
MPI::INTEGER4
MPI::REAL2
MPI::REAL4
MPI::REAL8

// optional datatypes (C / C++)
// Type: const MPI::Datatype
MPI::LONG_LONG_INT

// special datatypes for construction derived datatypes
// Type: const MPI::Datatype
MPI::UB
MPI::LB
```

```
// C++ datatypes
// Type: const MPI::Datatype
MPI::BOOL
MPI::COMPLEX
MPI::DOUBLE_COMPLEX
MPI::LONG_DOUBLE_COMPLEX

// reserved communicators
// Type: const MPI::Comm
MPI::COMM_WORLD
MPI::COMM_SELF

// results of communicator and group comparisons
// Type: const int (or enum)
MPI::IDENT
MPI::CONGRUENT
MPI::SIMILAR
MPI::UNEQUAL

// environmental inquiry keys
// Type: const int (or enum)
MPI::TAG_UB
MPI::IO
MPI::HOST
MPI::WTIME_IS_GLOBAL

// collective operations
// Type: const MPI::Op
MPI::MAX
MPI::MIN
MPI::SUM
MPI::PROD
MPI::MAXLOC
MPI::MINLOC
MPI::BAND
MPI::BOR
MPI::BXOR
MPI::LAND
MPI::LOR
MPI::LXOR

// Null handles
// Type: const MPI::Group
MPI::GROUP_NULL
// Type: const MPI::Comm
MPI::COMM_NULL
// Type: const MPI::Datatype
```

```
MPI::DATATYPE_NULL
// Type: const MPI::Request
MPI::REQUEST_NULL
// Type: const MPI::Op
MPI::OP_NULL
// Type: const int (or enum)
MPI::ERRHANDLER_NULL

// Empty group
// Type: const MPI::Group
MPI::GROUP_EMPTY

// topologies
// Type: const int (or enum)
MPI::GRAPH
MPI::CART
```

B.3 Typedefs

The following are defined C++ types, also included in the file `mpi++.h`.

```
// opaque types
MPI::Aint
```

The rest of this appendix uses the namespace notation because all the functions listed below are prototypes. The namespace notation is not used previously because the lists of constants and types above are not actual declarations.

```
// prototypes for user-defined functions
namespace MPI {
  typedef int Copy_function(Comm& oldcomm, int keyval,
                               void* extra_state, void *attribute_val_in,
                               void *attribute_val_out, int *flag);

  typedef int Delete_function(Comm& comm, int& keyval,
                                 void* extra_state);

  typedef void Handler_function(Comm& comm, int& err, ...);

  typedef void User_function(void *invec, void *inoutvec, int& len,
                                Datatype& datatype);
}
```

B.4 C++ Bindings for Point-to-Point Communication

```
namespace MPI {
```

```
int Comm::Send(const void* buf, int count, const Datatype& datatype,
          int dest, int tag) const

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

int Status::Get_count(const Datatype& datatype, int& count) const

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

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

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

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

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

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

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

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

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

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

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

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

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

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

int Comm::Probe(int source, int tag, Status& status) const

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

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

int Status::Test_cancelled(bool& flag) const

int Request::Wait(Status& status)

int Request::Test(bool& flag, Status& status)

int Request::Free(void)

int Request::Cancel(void) const

int Request::Start(void)

static int Buffer_attach(void* buffer, int size)

static int Buffer_detach(void*& buffer, int& size)

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

static int Testany(int count, Request array_of_requests[], int& index,
          bool& flag, Status& mpic)

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

static int Testall(int count, Request array_of_requests[], bool& flag,
          Status array_of_statuses[])

static int Waitsome(int incount, Request array_of_requests[],
          int& outcount, int array_of_indices[],
          Status array_of_statuses[])

static int Testsome(int incount, Request array_of_requests[],
          int& outcount, int array_of_indices[],
          Status array_of_statuses[])

static int Startall(int count, Request array_of_requests[])

int Datatype::Contiguous(int count, const Datatype& type)

int Datatype::Vector(int count, int blocklength, int stride,
          const Datatype& type)

int Datatype::Hvector(int count, int blocklength, Aint stride,
          const Datatype& type)

int Datatype::Indexed(int count, const int array_of_blocklengths[],
          const int array_of_displacements[], const Datatype& type)

int Datatype::Hindexed(int count, const int array_of_blocklengths[],
          const Aint array_of_displacements[], const Datatype& type)
```

```
  int Datatype::Struct(int count, const int array_of_blocklengths[],
            const Aint array_of_displacements[],
            const Datatype array_of_types[])

  static int Address(const void* location, Aint& address)

  int Datatype::Extent(int& extent) const

  int Datatype::Size(int& size) const

  int Datatype::Lb(int& displacement) const

  int Datatype::Ub(int& displacement) const

  int Datatype::Commit(void)

  int Datatype::Free(void)

  int Status::Get_elements(const Status& status, int& count) const

  int Datatype::Pack(const void* inbuf, int incount, void *outbuf,
            int outsize, int& position, const Comm &comm) const

  int Datatype::Unpack(const void* inbuf, int insize, int& position,
            void *outbuf, int outcount, const Comm& comm) const

  int Datatype::Pack_size(int incount, const Comm& comm, int& size) const

}
```

B.5 C++ Bindings for Collective Communication

```
namespace MPI {

  int Comm::Barrier(void) const

  int Comm::Bcast(void* buffer, int count, const Datatype& datatype,
            int root) const

  int Comm::Gather(const void* sendbuf, int sendcount, const
            Datatype& sendtype, void* recvbuf, int recvcount,
            const Datatype& recvtype, int root) const

  int Comm::Gatherv(const void* sendbuf, int sendcount, const
            Datatype& sendtype, void* recvbuf, const int recvcounts[],
            const int displs[], const Datatype& recvtype, int root) const

  int Comm::Scatter(const void* sendbuf, int sendcount, const
            Datatype& sendtype, void* recvbuf, int recvcount,
            const Datatype& recvtype, int root) const

  int Comm::Scatterv(const void* sendbuf, const int sendcounts[],
            const int displs[], const Datatype& sendtype, void* recvbuf,
            int recvcount, const Datatype& recvtype, int root) const
```

```
  int Comm::Allgather(const void* sendbuf, int sendcount, const
            Datatype& sendtype, void* recvbuf, int recvcount,
            const Datatype& recvtype) const

  int Comm::Allgatherv(const void* sendbuf, int sendcount, const
            Datatype& sendtype, void* recvbuf, const int recvcounts[],
            const int displs[], const Datatype& recvtype) const

  int Comm::Alltoall(const void* sendbuf, int sendcount, const
            Datatype& sendtype, void* recvbuf, int recvcount,
            const Datatype& recvtype) const

  int Comm::Alltoallv(const void* sendbuf, const int sendcounts[],
            const int sdispls[], const Datatype& sendtype, void* recvbuf,
            const int recvcounts[], const int rdispls[],
            const Datatype& recvtype) const

  int Comm::Reduce(const void* sendbuf, void* recvbuf, int count,
            const Datatype& datatype, const Op& op, int root) const

  int Op::Create(const User_function function, bool commute)

  int Op::Free(void)

  int Comm::Allreduce(const void* sendbuf, void* recvbuf, int count, const
            Datatype& datatype, const Op& op) const

  int Comm::Reduce_scatter(const void* sendbuf, void* recvbuf,
            int recvcounts[], const Datatype& datatype, const Op& op)
            const

  int Comm::Scan(const void* sendbuf, void* recvbuf, int count, const
            Datatype& datatype, const Op& op) const

}
```

B.6 C++ Bindings for Groups, Contexts, and Communicators

```
namespace MPI {

  int Group::Size(int& size) const

  int Group::Rank(int& rank) const

  int Group::Translate_ranks (const Group& group, int n,
            const int ranks1[], int ranks2[]) const

  int Group::Compare(const Group& group, int& result) const

  int Comm::Group(Group& group) const

  int Group::Union(const Group& group1, const Group& group2)

  int Group::Intersection(const Group& group1, const Group& group2)
```

```
int Group::Difference(const Group& group1, const Group& group2)

int Group::Incl(const Group& group, int n, const int ranks[])

int Group::Excl(const Group& group, int n, const int ranks[])

int Group::Range_incl(const Group& group, int n, const int ranges[][3])

int Group::Range_excl(const Group& group, int n, const int ranges[][3])

int Group::Free(void)

int Comm::Size(int& size) const

int Comm::Rank(int& rank) const

int Comm::Compare(const Comm& comm, int& result) const

int Comm::Dup(const Comm& comm)

int Comm::Create(const Comm& comm, const Group& group)

int Comm::Split(const Comm& comm, int color, int key)

int Comm::Free(void)

int Comm::Test_inter(bool& flag) const

int Comm::Remote_size(int& size) const

int Comm::Remote_group(Group& group) const

int Comm::Intercomm_create(const Comm& local_comm, int local_leader,
          const Comm& peer_comm, int remote_leader, int tag)

int Comm::Intercomm_merge(const Comm& intercomm, bool high)

static int Keyval_create(const Copy_function& copy_fn,
          const Delete_function* delete_fn, int& keyval,
          void* extra_state)

static int Keyval_free(int& keyval)

int Comm::Attr_put(int keyval, const void* attribute_val) const

int Comm::Attr_get(int keyval, void* attribute_val, bool& flag) const

int Comm::Attr_delete(int keyval) const

}
```

B.7 C++ Bindings for Process Topologies

```
namespace MPI {

  int Comm::Cart_create(const Comm& comm_old, int ndims, const int dims[],
            const bool periods[], bool reorder)
```

```
  static int Dims_create(int nnodes, int ndims, int dims[]) const

  int Comm::Graph_create(const Comm& comm_old, int nnodes,
            const int index[], const int edges[], bool reorder)

  int Comm::Topo_test(int& status) const

  int Comm::Graphdims_get(int nnodes[], int nedges[]) const

  int Comm::Graph_get(int maxindex, int maxedges, int index[], int edges[])
            const

  int Comm::Cartdim_get(int& ndims) const

  int Comm::Cart_get(int maxdims, int dims[], bool periods[], int coords[])
            const

  int Comm::Cart_rank(const int coords[], int& rank) const

  int Comm::Cart_coords(int rank, int maxdims, int coords[]) const

  int Comm::Graph_neighbors_count(int rank, int& nneighbors) const

  int Comm::Graph_neighbors(int rank, int maxneighbors, int neighbors[])
            const

  int Comm::Cart_shift(int direction, int disp, int& rank_source,
            int& rank_dest) const

  int Comm::Cart_sub(const Comm& comm, const bool remain_dims[])

  int Comm::Cart_map(int ndims, const int dims[], const bool periods[],
            int& newrank) const

  int Comm::Graph_map(int nnodes, const int index[], const int edges[],
            int& newrank) const

}
```

B.8 C++ Bindings for Environmental Inquiry

```
namespace MPI {

  static int Get_processor_name(char*& name, int& resultlen)

  int Errhandler::Create(const Handler_function* function)

  int Comm::Errhandler_set(const Errhandler& errhandler) const

  int Comm::Errhandler_get(Errhandler& errhandler) const

  int Errhandler::Free(void)

  int Error_string(int errorcode, char*& string, int& resultlen)

  int Error_class(int errorcode, int& errorclass)
```

```
  virtual static double Wtime(void) const

  virtual static double Wtick(void) const

  static int Init(int& argc, char**& argv)

  static int Finalize(void)

  static int Initialized(bool& flag) const

  int Comm::Abort(int errorcode)

}
```

B.9 C++ Bindings for Profiling

```
namespace MPI {

  static int Pcontrol(const int level, ...)

}
```

B.10 C++ Bindings for Status Access

```
namespace MPI {

  int Status::Get_source(void) const

  int Status::Set_source(int source)

  int Status::Get_tag(void) const

  int Status::Set_tag(int tag)

  int Status::Get_error(void) const

  int Status::Set_error(int error)

}
```

B.11 C++ Bindings on all MPI Classes

The C++ language requires all classes to have four special functions: a default constructor, a copy constructor, a destructor, and an assignment operator. The bindings for these functions are listed below; their semantics are discussed in section 8.1.4. The bindings prototype functions using the type ⟨`CLASS`⟩ rather than listing each function for every MPI class; the token ⟨`CLASS`⟩ can be replaced with valid MPI-2 class names, such as `Comm`, `Datatype`, etc. In addition, bindings are provided for comparison and inter-language operability from sections 8.1.4 and 9.7.4.

B.11.1 Construction / Destruction

```
namespace MPI {

  ⟨CLASS⟩(void)

  ~⟨CLASS⟩(void)

}
```

B.11.2 Copy / Assignment

```
namespace MPI {

  ⟨CLASS⟩(const ⟨CLASS⟩& data)

  ⟨CLASS⟩& ⟨CLASS⟩::operator=(const ⟨CLASS⟩& data)

}
```

B.11.3 Comparison

```
namespace MPI {

  bool ⟨CLASS⟩::operator==(const ⟨CLASS⟩& data) const

  bool ⟨CLASS⟩::operator!=(const ⟨CLASS⟩& data) const

}
```

B.11.4 Inter-language Operability

```
namespace MPI {

  ⟨CLASS⟩& ⟨CLASS⟩::operator=(const MPI_⟨CLASS⟩& data)

  ⟨CLASS⟩& ⟨CLASS⟩::operator=(const MPI_⟨CLASS⟩ data)

  ⟨CLASS⟩(const MPI_⟨CLASS⟩& data)

  ⟨CLASS⟩(const MPI_⟨CLASS⟩* data)

  ⟨CLASS⟩::operator MPI_⟨CLASS⟩() const

  ⟨CLASS⟩::operator MPI_⟨CLASS⟩*()

}
```

Annex C

MPI-1 C++ Language Binding - Proposal 2

Current Status: No votes

> **Discussion:**
> This is the "new" C++ bindings proposal.

C.1 C++ Classes

The following are the classes provided with the C++ MPI-1 language bindings:

```
namespace MPI {
  class Comm {...};
  class Intra : public Comm {...};
  class Graph : public Intra {...};
  class Cart : public Intra {...};
  class Inter : public Comm {...};
  class Datatype {...};
  class Errhandler {...};
  class Group {...};
  class Op {...};
  class Request {...};
  class Persistent_request : public Request {...};
  class Status {...};
}
```

C.2 Defined Constants

These are required constants, defined in the file `mpi++.h`. For brevity, the types of the constants are defined below are defined in the comments.

```
// return codes
```

```
// Type: const int (or enum)
MPI::SUCCESS
MPI::ERR_BUFFER
MPI::ERR_COUNT
MPI::ERR_TYPE
MPI::ERR_TAG
MPI::ERR_COMM
MPI::ERR_RANK
MPI::ERR_REQUEST
MPI::ERR_ROOT
MPI::ERR_GROUP
MPI::ERR_OP
MPI::ERR_TOPOLOGY
MPI::ERR_DIMS
MPI::ERR_ARG
MPI::ERR_UNKNOWN
MPI::ERR_TRUNCATE
MPI::ERR_OTHER
MPI::ERR_INTERN
MPI::ERR_PENDING
MPI::ERR_IN_STATUS
MPI::ERR_LASTCODE
// assorted constants
// Type: const MPI::Aint
MPI::BOTTOM
// Type: const int (or enum)
MPI::PROC_NULL
MPI::ANY_SOURCE
MPI::ANY_TAG
MPI::UNDEFINED
MPI::BSEND_OVERHEAD
MPI::KEYVAL_INVALID

// Error-handling specifiers

// Type: MPI::Handler_function (see below)
MPI::ERRORS_ARE_FATAL
MPI::ERRORS_RETURN


// Maximum sizes for strings
// Type: const int
MPI::MAX_PROCESSOR_NAME
MPI::MAX_ERROR_STRING

// elementary datatypes (C / C++)
// Type: const MPI::Datatype
MPI::CHAR
```

```
MPI::SHORT
MPI::INT
MPI::LONG
MPI::UNSIGNED_CHAR
MPI::UNSIGNED_SHORT
MPI::UNSIGNED
MPI::UNSIGNED_LONG
MPI::FLOAT
MPI::DOUBLE
MPI::LONG_DOUBLE
MPI::BYTE
MPI::PACKED

// elementary datatypes (Fortran 77)
// Type: const MPI::Datatype
MPI::INTEGER
MPI::REAL
MPI::DOUBLE_PRECISION
MPI::COMPLEX
MPI::LOGICAL
MPI::CHARACTER
MPI::BYTE

// datatypes for reduction functions (C / C++)
// Type: const MPI::Datatype
MPI::FLOAT_INT
MPI::DOUBLE_INT
MPI::LONG_INT
MPI::TWOINT
MPI::SHORT_INT
MPI::LONG_DOUBLE_INT

// datatype for reduction functions (Fortran 77)
// Type const MPI::Datatype
MPI::TWOREAL
MPI::TWODOUBLE_PRECISION
MPI::TWOINTEGER

// optional datatypes (Fortran 77)
// Type: const MPI::Datatype
MPI::INTEGER1
MPI::INTEGER2
MPI::INTEGER4
MPI::REAL2
MPI::REAL4
MPI::REAL8

// optional datatypes (C / C++)
```

```
// Type: const MPI::Datatype
MPI::LONG_LONG_INT

// special datatypes for construction derived datatypes
// Type: const MPI::Datatype
MPI::UB
MPI::LB

// C++ datatypes
// Type: const MPI::Datatype
MPI::BOOL
MPI::COMPLEX
MPI::DOUBLE_COMPLEX
MPI::LONG_DOUBLE_COMPLEX

// reserved communicators
// Type: const MPI::Comm
MPI::COMM_WORLD
MPI::COMM_SELF

// results of communicator and group comparisons
// Type: const int (or enum)
MPI::IDENT
MPI::CONGRUENT
MPI::SIMILAR
MPI::UNEQUAL

// environmental inquiry keys
// Type: const int (or enum)
MPI::TAG_UB
MPI::IO
MPI::HOST
MPI::WTIME_IS_GLOBAL

// collective operations
// Type: const MPI::Op
MPI::MAX
MPI::MIN
MPI::SUM
MPI::PROD
MPI::MAXLOC
MPI::MINLOC
MPI::BAND
MPI::BOR
MPI::BXOR
MPI::LAND
MPI::LOR
MPI::LXOR
```

```
// Null handles
// Type: const MPI::Group
MPI::GROUP_NULL
// Type: const MPI::Comm
MPI::COMM_NULL
// Type: const MPI::Datatype
MPI::DATATYPE_NULL
// Type: const MPI::Request
MPI::REQUEST_NULL
// Type: const MPI::Op
MPI::OP_NULL
// Type: const int (or enum)
MPI::ERRHANDLER_NULL

// Empty group
// Type: const MPI::Group
MPI::GROUP_EMPTY

// topologies
// Type: const int (or enum)
MPI::GRAPH
MPI::CART
```

C.3 Typedefs

The following are defined C++ types, also included in the file `mpi++.h`.

```
// opaque types
MPI::Aint
```

The rest of this appendix uses the namespace notation because all the functions listed below are prototypes. The namespace notation is not used previously because the lists of constants and types above are not actual declarations.

```
// prototypes for user-defined functions
namespace MPI {
  typedef int Copy_function(Comm& oldcomm, int keyval,
                               void* extra_state, void *attribute_val_in,
                               void *attribute_val_out, int *flag);

  typedef int Delete_function(Comm& comm, int& keyval,
                                 void* extra_state);

  typedef void Handler_function(Comm& comm, int& err, ...);

  typedef void User_function(void *invec, void *inoutvec, int& len,
                                Datatype& datatype);
}
```

C.4 C++ Bindings for Point-to-Point Communication

```
namespace MPI {

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

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

   int Status::Get_count(const Datatype& datatype) const

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

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

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

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

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

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

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

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

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

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

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

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

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

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

   void Comm::Probe(int source, int tag, Status& status) const

   void Comm::Sendrecv(const void *sendbuf, int sendcount, const
             Datatype& sendtype, int dest, int sendtag, void *recvbuf,
```

```
          int recvcount, const Datatype& recvtype, int source,
          int recvtag, Status& status) const

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

bool Status::Test_cancelled(void) const

void Request::Wait(Status& status)

bool Request::Test(Status& status)

void Request::Free(void)

void Request::Cancel(void) const

void Request::Persistent::Start(void)

void Buffer_attach(void* buffer, int size)

int Buffer_detach(void*& buffer)

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

bool Testany(int count, Request array_of_requests[], int& index,
          Status& status)

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

bool Testall(int count, Request array_of_requests[],
          Status array_of_statuses[])

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

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

void Startall(int count, Request::Persistent array_of_requests[])

Datatype Datatype::Contiguous(int count) const

Datatype Datatype::Vector(int count, int blocklength, int stride) const

Datatype Datatype::Hvector(int count, int blocklength, const
          Aint& stride) const

Datatype Datatype::Indexed(int count, const int array_of_blocklengths[],
          const int array_of_displacements[]) const

Datatype Datatype::Hindexed(int count, const int array_of_blocklengths[],
          const Aint array_of_displacements[]) const

Datatype Struct(int count, const int array_of_blocklengths[],
          const Aint array_of_displacements[],
          const Datatype array_of_types[])
```

```
  Aint Address(const void* location)

  int Datatype::Extent(void) const

  int Datatype::Size(void) const

  int Datatype::Lb(void) const

  int Datatype::Ub(void) const

  void Datatype::Commit(void)

  void Datatype::Free(void)

  int Status::Get_elements(const Status& status) const

  int Datatype::Pack(const void* inbuf, int incount, void *outbuf,
            int outsize, const Comm &comm) const

  int Datatype::Unpack(const void* inbuf, int insize, void *outbuf,
            int outcount, const Comm& comm) const

  int Datatype::Pack_size(int incount, const Comm& comm) const


}
```

C.5 C++ Bindings for Collective Communication

```
namespace MPI {

  void Comm::Barrier(void) const

  void Comm::Bcast(void* buffer, int count, const Datatype& datatype,
            int root) const

  void Comm::Gather(const void* sendbuf, int sendcount, const
            Datatype& sendtype, void* recvbuf, int recvcount,
            const Datatype& recvtype, int root) const

  void Comm::Gatherv(const void* sendbuf, int sendcount, const
            Datatype& sendtype, void* recvbuf, const int recvcounts[],
            const int displs[], const Datatype& recvtype, int root) const

  void Comm::Scatter(const void* sendbuf, int sendcount, const
            Datatype& sendtype, void* recvbuf, int recvcount,
            const Datatype& recvtype, int root) const

  void Comm::Scatterv(const void* sendbuf, const int sendcounts[],
            const int displs[], const Datatype& sendtype, void* recvbuf,
            int recvcount, const Datatype& recvtype, int root) const

  void Comm::Allgather(const void* sendbuf, int sendcount, const
            Datatype& sendtype, void* recvbuf, int recvcount,
            const Datatype& recvtype) const
```

```
  void Comm::Allgatherv(const void* sendbuf, int sendcount, const
            Datatype& sendtype, void* recvbuf, const int recvcounts[],
            const int displs[], const Datatype& recvtype) const

  void Comm::Alltoall(const void* sendbuf, int sendcount, const
            Datatype& sendtype, void* recvbuf, int recvcount,
            const Datatype& recvtype) const

  void Comm::Alltoallv(const void* sendbuf, const int sendcounts[],
            const int sdispls[], const Datatype& sendtype, void* recvbuf,
            const int recvcounts[], const int rdispls[],
            const Datatype& recvtype) const

  void Comm::Reduce(const void* sendbuf, void* recvbuf, int count,
            const Datatype& datatype, const Op& op, int root) const

  void Op::Create(const User_function function, bool commute)

  void Op::Free(void)

  void Comm::Allreduce(const void* sendbuf, void* recvbuf, int count,
            const Datatype& datatype, const Op& op) const

  void Comm::Reduce_scatter(const void* sendbuf, void* recvbuf,
            int recvcounts[], const Datatype& datatype, const Op& op)
            const

  void Comm::Scan(const void* sendbuf, void* recvbuf, int count, const
            Datatype& datatype, const Op& op) const

}
```

C.6 C++ Bindings for Groups, Contexts, and Communicators

```
namespace MPI {

  int Group::Size(void) const

  int Group::Rank(void) const

  void Group::Translate_ranks (const Group& group, int n,
            const int ranks1[], int ranks2[]) const

  int Group::Compare(const Group& group) const

  Group Comm::Group(void) const

  Group Group::Union(const Group& group2) const

  Group Group::Intersection(const Group& group2) const

  Group Group::Difference(const Group& group2) const

  Group Group::Incl(int n, const int ranks[]) const
```

```
Group Group::Excl(int n, const int ranks[]) const

Group Group::Range_incl(int n, const int ranges[][3]) const

Group Group::Range_excl(int n, const int ranges[][3]) const

void Group::Free(void)

int Comm::Size(void) const

int Comm::Rank(void) const

int Comm::Compare(const Comm& comm) const

Comm Comm::Dup(void) const

Comm::Intra Comm::Intra::Create(const Group& group) const

Comm::Intra Comm::Intra::Split(int color, int key) const
```

Discussion:
MPI-2 may change the semantics for MPI_COMM_CREATE and MPI_COMM_SPLIT to include Intercommunicators. Such changes will change the C++ bindings, but this is OK, because: 1) there are no "real" C++ implementations yet, and 2) it will exapand the bindings, not restrict them.

```
void Comm::Free(void)

bool Comm::Test_inter(void) const

int Comm::Inter::Remote_size(void) const

Group Comm::Inter::Remote_group(void) const

Comm::Inter Comm::Intra::Intercomm_create(int local_leader, const
           Comm& peer_comm, int remote_leader, int tag) const

Comm::Intra Comm::Inter::Merge(bool high) const

int Keyval_create(const Copy_function& copy_fn,
           const Delete_function* delete_fn, void* extra_state)

void Keyval_free(int& keyval)

void Comm::Attr_put(int keyval, const void* attribute_val) const

bool Comm::Attr_get(int keyval, void* attribute_val) const

void Comm::Attr_delete(int keyval) const

}
```

C.7 C++ Bindings for Process Topologies

```
namespace MPI {
```

```
  Comm::Intra::Cart Comm::Intra::Cart_create(int ndims, const int dims[],
            const bool periods[], bool reorder) const

  void Dims_create(int nnodes, int ndims, int dims[]) const

  Comm::Intra::Graph Comm::Intra::Graph_create(int nnodes,
            const int index[], const int edges[], bool reorder) const

  int Comm::Topo_test(void) const

  void Comm::Graph::Dims_get(int nnodes[], int nedges[]) const

  void Comm::Graph::Get(int maxindex, int maxedges, int index[],
            int edges[]) const

  int Comm::Cart::Dim_get(void) const

  void Comm::Cart::Get(int maxdims, int dims[], bool periods[],
            int coords[]) const

  int Comm::Cart::Rank(const int coords[]) const

  void Comm::Cart::Coords(int rank, int maxdims, int coords[]) const

  int Comm::Graph::Neighbors_count(int rank) const

  void Comm::Graph::Neighbors(int rank, int maxneighbors, int neighbors[])
            const

  void Comm::Cart::Shift(int direction, int disp, int& rank_source,
            int& rank_dest) const

  Comm::Intra::Cart Comm::Cart::Sub(const bool remain_dims[]) const

  int Comm::Cart::Map(int ndims, const int dims[], const bool periods[])
            const

  int Comm::Graph::Map(int nnodes, const int index[], const int edges[])
            const

}
```

C.8 C++ Bindings for Environmental Inquiry

```
namespace MPI {

  void Get_processor_name(char name[], int& resultlen)

  void Errhandler::Create(const Handler_function* function)

  void Comm::Errhandler_set(const Errhandler& errhandler) const

  void Comm::Errhandler_get(Errhandler& errhandler) const

  void Errhandler::Free(void)

  void Error_string(int errorcode, char name[], int& resultlen)
```

```
  int Error_class(int errorcode)

  double Wtime(void) const

  double Wtick(void) const

  void Init(int& argc, char**& argv)

  void Finalize(void)

  bool Initialized(void) const

  void Comm::Abort(int errorcode)

}
```

C.9 C++ Bindings for Profiling

```
namespace MPI {

  void Pcontrol(const int level, ...)

}
```

C.10 C++ Bindings for Status Access

```
namespace MPI {

  int Status::Get_source(void) const

  void Status::Set_source(int source)

  int Status::Get_tag(void) const

  void Status::Set_tag(int tag)

  int Status::Get_error(void) const

  void Status::Set_error(int error)

}
```

C.11 C++ Bindings on all MPI Classes

The C++ language requires all classes to have four special functions: a default constructor, a copy constructor, a destructor, and an assignment operator. The bindings for these functions are listed below; their semantics are discussed in section 8.1.4. The bindings prototype functions using the type ⟨`CLASS`⟩ rather than listing each function for every MPI class; the token ⟨`CLASS`⟩ can be replaced with valid MPI-2 class names, such as `Comm`, `Datatype`, etc. In addition, bindings are provided for comparison and inter-language operability from sections 8.1.4 and 9.7.4.

C.11.1 Construction / Destruction

```
namespace MPI {

  ⟨CLASS⟩(void)

  ~⟨CLASS⟩(void)

}
```

C.11.2 Copy / Assignment

```
namespace MPI {

  ⟨CLASS⟩(const ⟨CLASS⟩& data)

  ⟨CLASS⟩& ⟨CLASS⟩::operator=(const ⟨CLASS⟩& data)

}
```

C.11.3 Comparison

```
namespace MPI {

  bool ⟨CLASS⟩::operator==(const ⟨CLASS⟩& data) const

  bool ⟨CLASS⟩::operator!=(const ⟨CLASS⟩& data) const

}
```

C.11.4 Inter-language Operability

```
namespace MPI {

  ⟨CLASS⟩& ⟨CLASS⟩::operator=(const MPI_⟨CLASS⟩& data)

  ⟨CLASS⟩& ⟨CLASS⟩::operator=(const MPI_⟨CLASS⟩ data)

  ⟨CLASS⟩(const MPI_⟨CLASS⟩& data)

  ⟨CLASS⟩::operator MPI_⟨CLASS⟩() const

  ⟨CLASS⟩(const MPI_⟨CLASS⟩* data)

  ⟨CLASS⟩::operator MPI_⟨CLASS⟩*()

}
```

MPI Function Index

MPI_ACCEPT, 55
MPI_ACCUMULATE, 82
MPI_ACQUIRE_SHARED_BUFFER, 329
MPI_ADD_ERROR_CLASS, 145
MPI_ADD_ERROR_CODES, 145
MPI_ADD_ERROR_STRING, 146
MPI_ADDRESS_X, 199
MPI_ALLGATHER, 117
MPI_ALLGATHERV, 118
MPI_ALLREDUCE, 121
MPI_ALLTOALL, 119
MPI_ALLTOALLV, 120
MPI_ALLTOALLW, 124
MPI_ATTR_GET_X, 203
MPI_ATTR_PUT_X, 203
MPI_BARRIER, 122
MPI_BARRIER_INIT, 107
MPI_BCAST, 114
MPI_BCAST_INIT, 106
MPI_BIND, 126
MPI_CLOSE, 210
MPI_COMM_CREATE, 108
MPI_COMM_DISCONNECT, 70
MPI_COMM_SPLIT, 109
MPI_COND_BROADCAST, 317
MPI_COND_CREATE, 316
MPI_COND_FREE, 316
MPI_COND_SIGNAL, 317
MPI_COND_WAIT, 317
MPI_CONNECT, 56
MPI_DELETE, 211
MPI_DO, 141
MPI_ERR_IS_CONTINUABLE, 204
MPI_ERRHANDLER_FREE, 30
MPI_EXSCAN, 125
MPI_FILE_GET_AMODE, 213
MPI_FILE_GET_ATOMICITY, 239
MPI_FILE_GET_COMM, 212
MPI_FILE_GET_INFO, 214
MPI_FILE_GET_POSITION, 227
MPI_FILE_GET_POSITION_SHARED, 234
MPI_FILE_GET_SIZE, 212
MPI_FILE_GET_TYPE_EXTENT, 237
MPI_FILE_GET_TYPE_SIZE, 237
MPI_FILE_GET_VIEW, 217
MPI_FILE_SET_ATOMICITY, 239
MPI_FILE_SET_INFO, 213
MPI_FILE_SET_VIEW, 216
MPI_FILE_SYNC, 239
MPI_FINALIZED, 186
MPI_FREE_SHARED_BUFFER, 330
MPI_GATHER, 115
MPI_GATHERV, 116
MPI_GET, 78
MPI_GET_EXTENT_X, 200
MPI_GET_STATUS, 185
MPI_GET_TYPE_CONTENTS, 148
MPI_GET_TYPE_ENVELOPE, 147
MPI_GET_VERSION, 21
MPI_GROUP_MONITOR, 313
MPI_HANDLE_ATTR_DELETE, 321
MPI_HANDLE_ATTR_GET, 320
MPI_HANDLE_ATTR_PUT, 320
MPI_HANDLE_KEYVAL_CREATE, 319
MPI_IACCEPT, 59
MPI_ICLOSE, 210
MPI_ICONNECT, 59
MPI_INFO_CREATE, 64
MPI_INFO_DELETE, 64
MPI_INFO_DUP, 66
MPI_INFO_FREE, 66
MPI_INFO_GET, 65
MPI_INFO_GET_NKEYS, 66
MPI_INFO_GET_NTHKEY, 66
MPI_INFO_GET_VALUELEN, 65
MPI_INFO_SET, 64
MPI_IOPEN, 209
MPI_IREAD, 225
MPI_IREAD_ALL, 226
MPI_IREAD_EXPLICIT, 222

MPI_IREAD_EXPLICIT_ALL, 222
MPI_IREAD_SHARED, 229
MPI_IREAD_SHARED_ORDERED, 232
MPI_ISPAWN, 47
MPI_ISPAWN_INDEPENDENT, 309
MPI_ISPAWN_MULTIPLE, 48
MPI_ISPAWN_MULTIPLE_INDEPENDENT, 310
MPI_IWRITE, 226
MPI_IWRITE_ALL, 226
MPI_IWRITE_EXPLICIT, 223
MPI_IWRITE_EXPLICIT_ALL, 223
MPI_IWRITE_SHARED, 230
MPI_IWRITE_SHARED_ORDERED, 233
MPI_KEYVAL_CREATE_X, 202
MPI_MAKE_SHARED_BUFFER, 328
MPI_MEM_ALLOC, 72
MPI_MEM_FREE, 72
MPI_META_REQUEST_CREATE, 130
MPI_META_REQUEST_FREE, 135
MPI_META_REQUEST_INIT, 131
MPI_MUTEX_CREATE, 314
MPI_MUTEX_FREE, 315
MPI_MUTEX_LOCK, 315
MPI_MUTEX_TRYLOCK, 315
MPI_MUTEX_UNLOCK, 316
MPI_NAME_GET, 58, 143
MPI_NAME_PUBLISH, 57
MPI_NAME_PUT, 143
MPI_NAME_UNPUBLISH, 58
MPI_OP_FREE, 29
MPI_OPEN, 208
MPI_PORT_CLOSE, 55
MPI_PORT_OPEN, 54
MPI_POST_HANDLER, 161
MPI_PROCESS_MONITOR, 312
MPI_PUT, 77
MPI_READ, 224
MPI_READ_ALL, 224
MPI_READ_EXPLICIT, 220
MPI_READ_EXPLICIT_ALL, 220
MPI_READ_SHARED, 228
MPI_READ_SHARED_ORDERED, 231
MPI_REDUCE, 120
MPI_REDUCE_SCATTER, 121
MPI_RELEASE_SHARED_BUFFER, 329
MPI_REQUEST_MARK_COMPLETE, 133
MPI_REQUEST_POST_HANDLER, 277
MPI_REQUEST_TYPE, 157
MPI_RESIZE, 211
MPI_SCAN, 122
MPI_SCATTER, 116
MPI_SCATTERV, 117
MPI_SEEK, 227
MPI_SEEK_SHARED, 233
MPI_SET_ELEMENTS_STATUS, 142
MPI_SIGNAL, 310
MPI_SPAWN, 41
MPI_SPAWN_INDEPENDENT, 306
MPI_SPAWN_MULTIPLE, 45
MPI_SPAWN_MULTIPLE_INDEPENDENT, 308
MPI_START_TIME, 274
MPI_THREAD_FINALIZE, 160
MPI_THREAD_INIT, 160
MPI_THREAD_INITIALIZED, 160
MPI_TRUE_EXTENT_X, 201
MPI_TYPE_COMPARE, 150
MPI_TYPE_CONTIGUOUS_STRUCT, 325
MPI_TYPE_DARRAY, 244
MPI_TYPE_DARRAY_BLOCK, 245
MPI_TYPE_DARRAY_CYCLIC, 246
MPI_TYPE_DECODE, 152
MPI_TYPE_ENCODE, 151
MPI_TYPE_ENCODE_SIZE, 152
MPI_TYPE_F90, 180
MPI_TYPE_HINDEXED_X, 198
MPI_TYPE_HVECTOR_X, 198
MPI_TYPE_INDEXED_BLOCK, 184
MPI_TYPE_RESIZED_X, 200
MPI_TYPE_SCATTER_GATHER, 241
MPI_TYPE_SCATTERV_GATHERV, 242
MPI_TYPE_SIMPLE_STRUCT, 323
MPI_TYPE_STRUCT_X, 199
MPI_TYPE_SUBARRAY, 242
MPI_WIN_BARRIER, 90
MPI_WIN_COMPLETE, 87
MPI_WIN_FREE, 75
MPI_WIN_INIT, 74
MPI_WIN_LOCK, 91
MPI_WIN_POST, 88
MPI_WIN_START, 87
MPI_WIN_TEST, 89
MPI_WIN_UNLOCK, 91
MPI_WIN_WAIT, 88
MPI_WRITE, 225

MPI_WRITE_ALL, 225
MPI_WRITE_EXPLICIT, 221
MPI_WRITE_EXPLICIT_ALL, 221
MPI_WRITE_SHARED, 229
MPI_WRITE_SHARED_ORDERED, 231
MPIRT_BUFFER_AVAIL, 261
MPIRT_BUFFER_GET, 261
MPIRT_BUFFER_POOL_CREATE, 260
MPIRT_BUFFER_POOL_HANDLE_FREE, 261
MPIRT_CHANNEL_BUFPOOL_MODIFY, 269
MPIRT_CHANNEL_COLL_DELETE, 270
MPIRT_CHANNEL_DELETE, 269
MPIRT_CHANNEL_MODIFY, 268
MPIRT_CHANNELS_BUFPOOL_INIT, 265, 285
MPIRT_CHANNELS_BUFPOOL_MODIFY, 267
MPIRT_CHANNELS_DELETE, 266
MPIRT_CHANNELS_INIT, 264
MPIRT_CHANNELS_MODIFY, 267
MPIRT_Deregister_Event_Names, 288
MPIRT_Generate_User_Event, 288
MPIRT_GUARDED_CHANNEL_USE, 287
MPIRT_GUARDED_FUNCTION, 287
MPIRT_MONITOR_CREATE, 296
MPIRT_MONITOR_DELETE, 296
MPIRT_MONITOR_RESULTS, 296
MPIRT_Register_Event_Names, 288
MPIRT_Register_Generated_Event_Names, 288