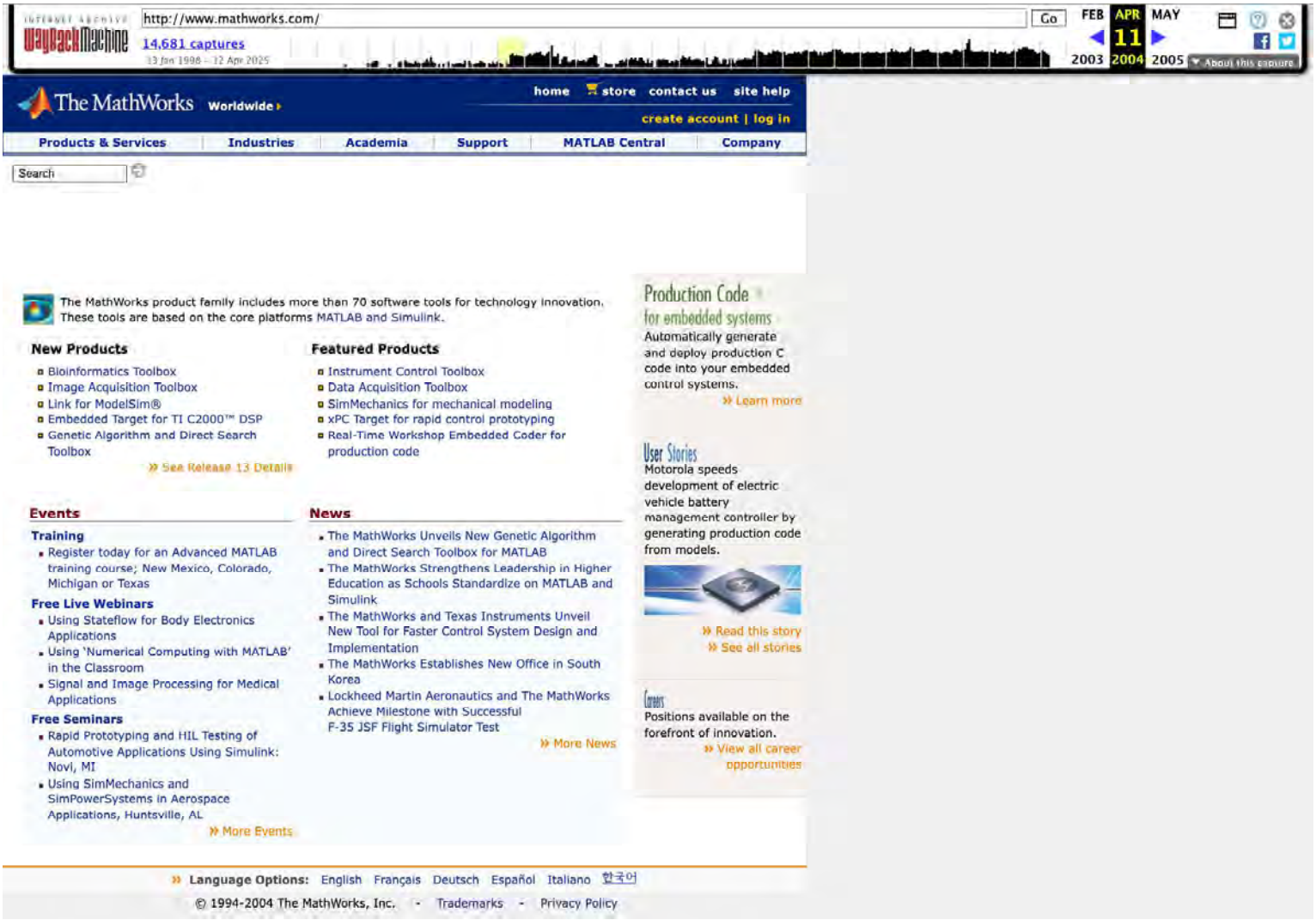
# EXHIBIT A

https://web.archive.org/web/20040411173035/http:/www.mathworks.com/


http://www.mathworks.com/
Go
14,681 captures
13 Jan 1998 - 12 Apr 2025
FEB APR MAY
11
2003 2004 2005
About this capture
The MathWorks Worldwide
home store contact us site help
create account | log in
Products & Services Industries Academia Support MATLAB Central Company
Search
The MathWorks product family includes more than 70 software tools for technology innovation. These tools are based on the core platforms MATLAB and Simulink.
New Products
Bioinformatics Toolbox
Image Acquisition Toolbox
Link for ModelSim®
Embedded Target for TI C2000™ DSP
Genetic Algorithm and Direct Search Toolbox
» See Release 13 Details
Featured Products
Instrument Control Toolbox
Data Acquisition Toolbox
SimMechanics for mechanical modeling
xPC Target for rapid control prototyping
Real-Time Workshop Embedded Coder for production code
Events
Training
Register today for an Advanced MATLAB training course; New Mexico, Colorado, Michigan or Texas
Free Live Webinars
Using Stateflow for Body Electronics Applications
Using 'Numerical Computing with MATLAB' in the Classroom
Signal and Image Processing for Medical Applications
Free Seminars
Rapid Prototyping and HIL Testing of Automotive Applications Using Simulink: Novi, MI
Using SimMechanics and SimPowerSystems in Aerospace Applications, Huntsville, AL
» More Events
News
The MathWorks Unveils New Genetic Algorithm and Direct Search Toolbox for MATLAB
The MathWorks Strengthens Leadership in Higher Education as Schools Standardize on MATLAB and Simulink
The MathWorks and Texas Instruments Unveil New Tool for Faster Control System Design and Implementation
The MathWorks Establishes New Office in South Korea
Lockheed Martin Aeronautics and The MathWorks Achieve Milestone with Successful F-35 JSF Flight Simulator Test
» More News
Production Code for embedded systems
Automatically generate and deploy production C code into your embedded control systems.
» Learn more
User Stories
Motorola speeds development of electric vehicle battery management controller by generating production code from models.
» Read this story
» See all stories
Careers
Positions available on the forefront of innovation.
» View all career opportunities
» Language Options: English Français Deutsch Español Italiano 한국어
© 1994-2004 The MathWorks, Inc. - Trademarks - Privacy Policy

https://web.archive.org/web/20040401104314/http:/www.mathworks.com/matlabcentral/

http://www.mathworks.com/matlabcentral/ Go FEB APR JUN 01 2003 2004 2005
2,919 captures
1 Nov 2003 - 10 Apr 2025

# MATLAB CENTRAL

File exchange and newsgroup access for the MATLAB & Simulink user community

The MathWorks

File Exchange | MATLAB Newsgroup | Programming Contest | MathWorks.com

## File Exchange

a user-contributed code library

The MATLAB Central File Exchange contains over a thousand files contributed by users and developers of MATLAB, Simulink and related products.

Recently Added Files...

- Particle Size Analysis from Image Processing Webinar - Robert Bemis
- AUTOSAVE - Steve Simon
- Snake! - Adam Wojciechowski

## MATLAB Newsgroup

usenet newsgroup access

comp.soft-sys.matlab is a public Usenet newsgroup for MATLAB, Simulink and related products. This is an independent, unmoderated forum. Find out what is comp.soft-sys.matlab or see the newsgroup FAQ for more information.

Recently added posts...

- Why do I get run-time errer 429 ? (1 message) - dj Lee
- modelling a guitar using waveguides (1 message) - gareth
- matlab combinatoric: grouping a set of n-elements (1 message) - Picander

**Public Submission Policy**

NOTICE: Any content you submit to MATLAB Central, including personal information, is not subject to the protections which may be afforded information collected under other sections of The MathWorks, Inc. Web site. You are entirely responsible for all content that you upload, post, e-mail, transmit or otherwise make available via MATLAB Central. The MathWorks does not control the content posted by visitors to MATLAB Central and, does not guarantee the accuracy, integrity, or quality of such content. Under no circumstances will The MathWorks be liable in any way for any content not authored by The MathWorks, or any loss or damage of any kind incurred as a result of the use of any content posted, e-mailed, transmitted or otherwise made available via MATLAB Central. Read the complete Disclaimer prior to use.

## Coming soon

**File Exchange Highlights**
Picked weekly by MathWorks engineers, Scott and Doug.




**Free Webinars**
Using Numerical Computing with MATLAB in the Classroom
April 27, 2004

**Link to Us**
Link Your Page to MATLAB Central

**Screensaver**
Download the new MATLAB Screensaver



**Available for:**
Windows 98/NT/2000/XP

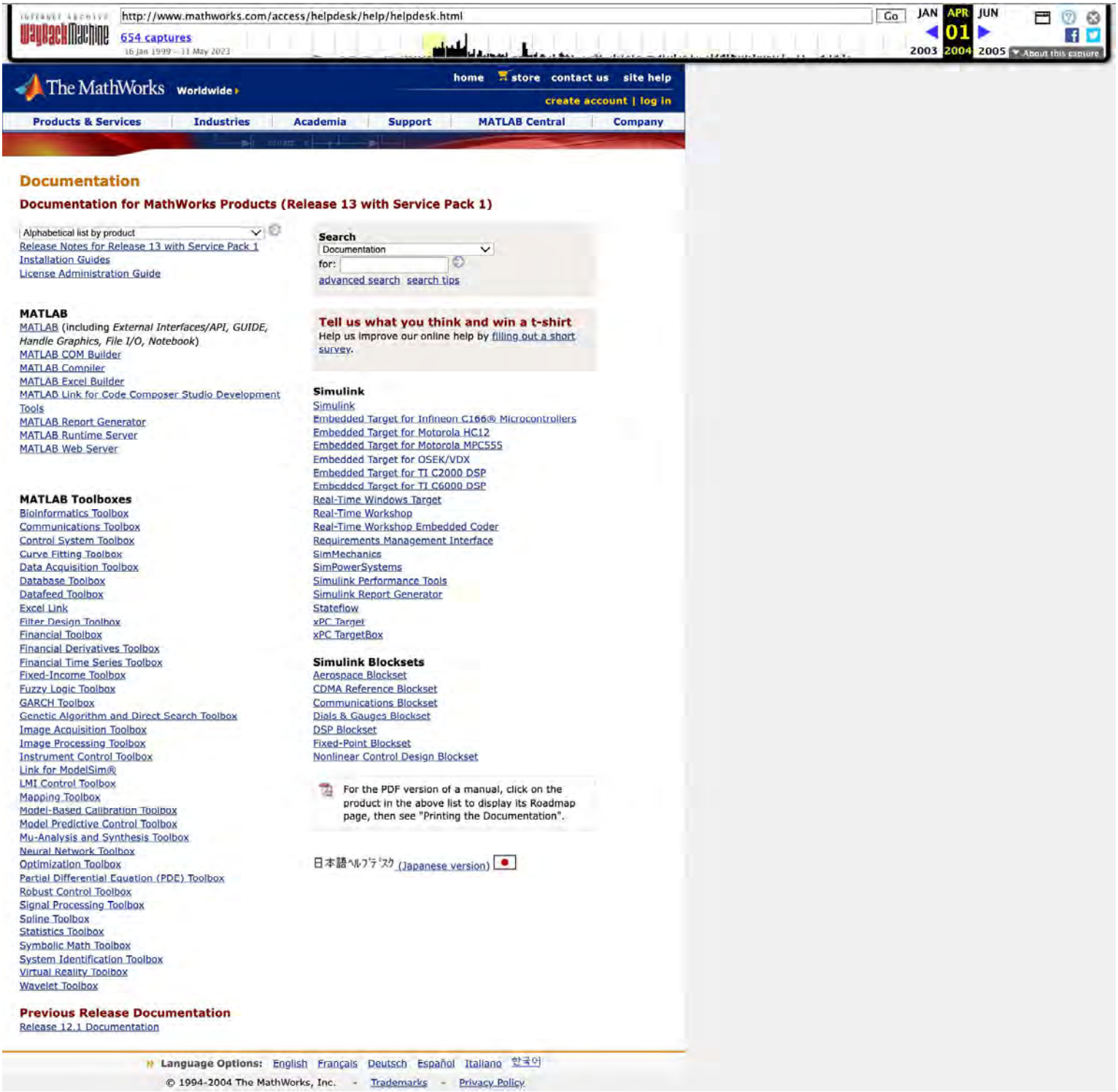
## Related Topics

New Products | Support | Documentation | Training | Webinars | Careers | Newsletters

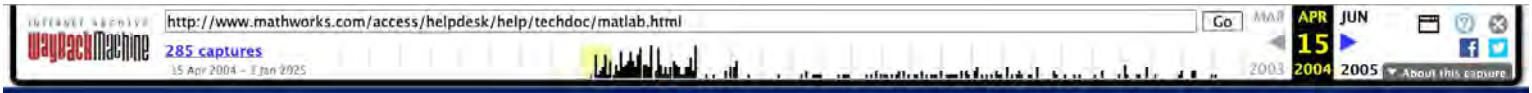
Problems? Suggestions? Contact us at files@mathworks.com

© 1994-2004 The MathWorks, Inc. Trademarks Privacy Policy

https://web.archive.org/web/20040401082318/http:/www.mathworks.com/access/helpdesk/help/helpdesk.html


http://www.mathworks.com/access/helpdesk/help/helpdesk.html
Go
JAN APR JUN
01
2003 2004 2005
654 captures
16 Jan 1999 - 11 May 2023
About this capture
The MathWorks Worldwide
home store contact us site help
create account | log in
Products & Services
Industries
Academia
Support
MATLAB Central
Company
Documentation
Documentation for MathWorks Products (Release 13 with Service Pack 1)
Alphabetical list by product
Release Notes for Release 13 with Service Pack 1
Installation Guides
License Administration Guide
Search
Documentation
for:
advanced search search tips
Tell us what you think and win a t-shirt
Help us improve our online help by filling out a short survey.
MATLAB
MATLAB (including External Interfaces/API, GUIDE, Handle Graphics, File I/O, Notebook)
MATLAB COM Builder
MATLAB Compiler
MATLAB Excel Builder
MATLAB Link for Code Composer Studio Development Tools
MATLAB Report Generator
MATLAB Runtime Server
MATLAB Web Server
MATLAB Toolboxes
Bioinformatics Toolbox
Communications Toolbox
Control System Toolbox
Curve Fitting Toolbox
Data Acquisition Toolbox
Database Toolbox
Datafeed Toolbox
Excel Link
Filter Design Toolbox
Financial Toolbox
Financial Derivatives Toolbox
Financial Time Series Toolbox
Fixed-Income Toolbox
Fuzzy Logic Toolbox
GARCH Toolbox
Genetic Algorithm and Direct Search Toolbox
Image Acquisition Toolbox
Image Processing Toolbox
Instrument Control Toolbox
Link for ModelSim®
LMI Control Toolbox
Mapping Toolbox
Model-Based Calibration Toolbox
Model Predictive Control Toolbox
Mu-Analysis and Synthesis Toolbox
Neural Network Toolbox
Optimization Toolbox
Partial Differential Equation (PDE) Toolbox
Robust Control Toolbox
Signal Processing Toolbox
Spline Toolbox
Statistics Toolbox
Symbolic Math Toolbox
System Identification Toolbox
Virtual Reality Toolbox
Wavelet Toolbox
Simulink
Simulink
Embedded Target for Infineon C166® Microcontrollers
Embedded Target for Motorola HC12
Embedded Target for Motorola MPC555
Embedded Target for OSEK/VDX
Embedded Target for TI C2000 DSP
Embedded Target for TI C6000 DSP
Real-Time Windows Target
Real-Time Workshop
Real-Time Workshop Embedded Coder
Requirements Management Interface
SimMechanics
SimPowerSystems
Simulink Performance Tools
Simulink Report Generator
Stateflow
xPC Target
xPC TargetBox
Simulink Blocksets
Aerospace Blockset
CDMA Reference Blockset
Communications Blockset
Dials & Gauges Blockset
DSP Blockset
Fixed-Point Blockset
Nonlinear Control Design Blockset
For the PDF version of a manual, click on the product in the above list to display its Roadmap page, then see "Printing the Documentation".
日本語ﾍﾙﾌﾟﾃﾞｽｸ (Japanese version)
Previous Release Documentation
Release 12.1 Documentation
Language Options: English Français Deutsch Español Italiano 한국어
© 1994-2004 The MathWorks, Inc. - Trademarks - Privacy Policy

https://web.archive.org/web/20040415214322/http:/www.mathworks.com/access/helpdesk/help/techdoc/matlab.html






Getting Started
Examples
Development Environment
Mathematics
Programming and Data Types
Graphics
3-D Visualization
Creating Graphical User Interfaces
Functions - By Category
Functions - Alphabetical List
Handle Graphics Property Browser
External Interfaces/API
External Interfaces/API Reference
Release Notes

Printable Documentation (PDF)

Roadmap

# MATLAB

## Learning MATLAB

- Getting Started - introduction to MATLAB.
- Using MATLAB - user guides for all of MATLAB.
- Programming Tips - tips on many aspects of programming with MATLAB.
- Examples - major examples in the MATLAB documentation.
- Release Notes - summary of new features, bug fixes, upgrade issues, etc.

## Finding Functions and Properties

- MATLAB Functions Listed by Category - browse MATLAB functions by category.
- MATLAB Functions Listed Alphabetically - find functions from an alphabetical list.

If you know the MATLAB function name, enter it below and click **Go**:

Go

- Handle Graphics Property Browser - view descriptions of all graphics object properties.

## Printing the Documentation

- Printable versions of the MATLAB documentation and related papers in PDF format.

© 1994-2004 The MathWorks, Inc. - Trademarks - Privacy Policy

# EXHIBIT B

https://web.archive.org/web/19970624201916/http://www.mcs.anl.gov/mpi/mpi-report.ps

# MPI: A Message-Passing Interface Standard

Message Passing Interface Forum

May 5, 1994

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

The Message Passing Interface Forum (MPIF), with participation from over 40 organizations, has been meeting since November 1992 to discuss and define a set of library interface standards for message passing. MPIF is not sanctioned or supported by any official standards organization.

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

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

Please send comments on MPI to `mpi-comments@cs.utk.edu`. Your comment will be forwarded to MPIF committee members who will attempt to respond.

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

# Contents


Acknowledgments vi

**1 Introduction to MPI** **1**
- 1.1 Overview and Goals . . . 1
- 1.2 Who Should Use This Standard? . . . 2
- 1.3 What Platforms Are Targets For Implementation? . . . 3
- 1.4 What Is Included In The Standard? . . . 3
- 1.5 What Is Not Included In The Standard? . . . 3
- 1.6 Organization of this Document . . . 4

**2 MPI Terms and Conventions** **6**
- 2.1 Document Notation . . . 6
- 2.2 Procedure Specification . . . 6
- 2.3 Semantic Terms . . . 7
- 2.4 Data Types . . . 8
  - 2.4.1 Opaque objects . . . 8
  - 2.4.2 Array arguments . . . 9
  - 2.4.3 State . . . 9
  - 2.4.4 Named constants . . . 9
  - 2.4.5 Choice . . . 9
  - 2.4.6 Addresses . . . 10
- 2.5 Language Binding . . . 10
  - 2.5.1 Fortran 77 Binding Issues . . . 10
  - 2.5.2 C Binding Issues . . . 11
- 2.6 Processes . . . 12
- 2.7 Error Handling . . . 12
- 2.8 Implementation issues . . . 13
  - 2.8.1 Independence of Basic Runtime Routines . . . 13
  - 2.8.2 Interaction with signals in POSIX . . . 14

**3 Point-to-Point Communication** **15**
- 3.1 Introduction . . . 15
- 3.2 Blocking Send and Receive Operations . . . 16
  - 3.2.1 Blocking send . . . 16
  - 3.2.2 Message data . . . 16
  - 3.2.3 Message envelope . . . 18
  - 3.2.4 Blocking receive . . . 19
  - 3.2.5 Return status . . . 20

3.3 Data type matching and data conversion . . . 21
3.3.1 Type matching rules . . . 21
3.3.2 Data conversion . . . 24
3.4 Communication Modes . . . 25
3.5 Semantics of point-to-point communication . . . 29
3.6 Buffer allocation and usage . . . 32
3.6.1 Model implementation of buffered mode . . . 34
3.7 Nonblocking communication . . . 34
3.7.1 Communication Objects . . . 36
3.7.2 Communication initiation . . . 36
3.7.3 Communication Completion . . . 39
3.7.4 Semantics of Nonblocking Communications . . . 42
3.7.5 Multiple Completions . . . 43
3.8 Probe and Cancel . . . 48
3.9 Persistent communication requests . . . 52
3.10 Send-receive . . . 56
3.11 Null processes . . . 58
3.12 Derived datatypes . . . 59
3.12.1 Datatype constructors . . . 60
3.12.2 Address and extent functions . . . 67
3.12.3 Lower-bound and upper-bound markers . . . 69
3.12.4 Commit and free . . . 70
3.12.5 Use of general datatypes in communication . . . 72
3.12.6 Correct use of addresses . . . 74
3.12.7 Examples . . . 75
3.13 Pack and unpack . . . 83

**4 Collective Communication** **90**
4.1 Introduction and Overview . . . 90
4.2 Communicator argument . . . 93
4.3 Barrier synchronization . . . 93
4.4 Broadcast . . . 93
4.4.1 Example using MPI_BCAST . . . 94
4.5 Gather . . . 94
4.5.1 Examples using MPI_GATHER, MPI_GATHERV . . . 96
4.6 Scatter . . . 103
4.6.1 Examples using MPI_SCATTER, MPI_SCATTERV . . . 105
4.7 Gather-to-all . . . 107
4.7.1 Examples using MPI_ALLGATHER, MPI_ALLGATHERV . . . 109
4.8 All-to-All Scatter/Gather . . . 109
4.9 Global Reduction Operations . . . 111
4.9.1 Reduce . . . 111
4.9.2 Predefined reduce operations . . . 112
4.9.3 MINLOC and MAXLOC . . . 114
4.9.4 User-Defined Operations . . . 118
4.9.5 All-Reduce . . . 122
4.10 Reduce-Scatter . . . 123
4.11 Scan . . . 124

4.11.1 Example using MPI_SCAN . . . . . 124
4.12 Correctness . . . . . 126

**5 Groups, Contexts, and Communicators 130**
5.1 Introduction . . . . . 130
5.1.1 Features Needed to Support Libraries . . . . . 130
5.1.2 MPI's Support for Libraries . . . . . 131
5.2 Basic Concepts . . . . . 133
5.2.1 Groups . . . . . 133
5.2.2 Contexts . . . . . 133
5.2.3 Intra-Communicators . . . . . 134
5.2.4 Predefined Intra-Communicators . . . . . 134
5.3 Group Management . . . . . 135
5.3.1 Group Accessors . . . . . 135
5.3.2 Group Constructors . . . . . 136
5.3.3 Group Destructors . . . . . 140
5.4 Communicator Management . . . . . 141
5.4.1 Communicator Accessors . . . . . 141
5.4.2 Communicator Constructors . . . . . 142
5.4.3 Communicator Destructors . . . . . 145
5.5 Motivating Examples . . . . . 146
5.5.1 Current Practice #1 . . . . . 146
5.5.2 Current Practice #2 . . . . . 147
5.5.3 (Approximate) Current Practice #3 . . . . . 147
5.5.4 Example #4 . . . . . 148
5.5.5 Library Example #1 . . . . . 149
5.5.6 Library Example #2 . . . . . 151
5.6 Inter-Communication . . . . . 153
5.6.1 Inter-communicator Accessors . . . . . 154
5.6.2 Inter-communicator Operations . . . . . 156
5.6.3 Inter-Communication Examples . . . . . 158
5.7 Caching . . . . . 165
5.7.1 Functionality . . . . . 165
5.7.2 Attributes Example . . . . . 169
5.8 Formalizing the Loosely Synchronous Model . . . . . 171
5.8.1 Basic Statements . . . . . 171
5.8.2 Models of Execution . . . . . 172

**6 Process Topologies 174**
6.1 Introduction . . . . . 174
6.2 Virtual Topologies . . . . . 175
6.3 Embedding in MPI . . . . . 175
6.4 Overview of the Functions . . . . . 176
6.5 Topology Constructors . . . . . 177
6.5.1 Cartesian Constructor . . . . . 177
6.5.2 Cartesian Convenience Function: MPI_DIMS_CREATE . . . . . 177
6.5.3 General (Graph) Constructor . . . . . 178
6.5.4 Topology inquiry functions . . . . . 180

6.5.5 Cartesian Shift Coordinates . . . . . 184
6.5.6 Partitioning of Cartesian structures . . . . . 185
6.5.7 Low-level topology functions . . . . . 185
6.6 An Application Example . . . . . 187

**7 MPI Environmental Management** **189**
7.1 Implementation information . . . . . 189
7.1.1 Environmental Inquiries . . . . . 189
7.2 Error handling . . . . . 191
7.3 Error codes and classes . . . . . 194
7.4 Timers . . . . . 195
7.5 Startup . . . . . 196

**8 Profiling Interface** **198**
8.1 Requirements . . . . . 198
8.2 Discussion . . . . . 198
8.3 Logic of the design . . . . . 199
8.3.1 Miscellaneous control of profiling . . . . . 199
8.4 Examples . . . . . 200
8.4.1 Profiler implementation . . . . . 200
8.4.2 MPI library implementation . . . . . 201
8.4.3 Complications . . . . . 202
8.5 Multiple levels of interception . . . . . 203

**Bibliography** **204**

**A Language Binding** **207**
A.1 Introduction . . . . . 207
A.2 Defined Constants for C and Fortran . . . . . 207
A.3 C bindings for Point-to-Point Communication . . . . . 211
A.4 C Bindings for Collective Communication . . . . . 214
A.5 C Bindings for Groups, Contexts, and Communicators . . . . . 215
A.6 C Bindings for Process Topologies . . . . . 216
A.7 C bindings for Environmental Inquiry . . . . . 217
A.8 C Bindings for Profiling . . . . . 217
A.9 Fortran Bindings for Point-to-Point Communication . . . . . 217
A.10 Fortran Bindings for Collective Communication . . . . . 221
A.11 Fortran Bindings for Groups, Contexts, etc. . . . . 223
A.12 Fortran Bindings for Process Topologies . . . . . 224
A.13 Fortran Bindings for Environmental Inquiry . . . . . 225
A.14 Fortran Bindings for Profiling . . . . . 226

**MPI Function Index** **227**

# Acknowledgments

The technical development was carried out by subgroups, whose work was reviewed by the full committee. During the period of development of the Message Passing Interface (MPI), many people served in positions of responsibility and are listed below.

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

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

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

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

MPI operated on a very tight budget (in reality, it had no budget when the first meeting was announced). ARPA and NSF have supported research at various institutions that have

made a contribution towards travel for the U.S. academics. Support for several European participants was provided by ESPRIT.

# Chapter 1

# Introduction to MPI

## 1.1 Overview and Goals

Message passing is a paradigm used widely on certain classes of parallel machines, especially those with distributed memory. Although there are many variations, the basic concept of processes communicating through messages is well understood. Over the last ten years, substantial progress has been made in casting significant applications in this paradigm. Each vendor has implemented its own variant. More recently, several systems have demonstrated that a message passing system can be efficiently and portably implemented. It is thus an appropriate time to try to define both the syntax and semantics of a core of library routines that will be useful to a wide range of users and efficiently implementable on a wide range of computers.

In designing MPI we have sought to make use of the most attractive features of a number of existing message passing systems, rather than selecting one of them and adopting it as the standard. Thus, MPI has been strongly influenced by work at the IBM T. J. Watson Research Center [1, 2], Intel's NX/2 [23], Express [22], nCUBE's Vertex [21], p4 [7, 6], and PARMACS [5, 8]. Other important contributions have come from Zipcode [24, 25], Chimp [14, 15], PVM [4, 11], Chameleon [19], and PICL [18].

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

A preliminary draft proposal, known as MPI1, was put forward by Dongarra, Hempel, Hey, and Walker in November 1992, and a revised version was completed in February 1993 [12]. MPI1 embodied the main features that were identified at the Williamsburg workshop as being necessary in a message passing standard. Since MPI1 was primarily intended to promote discussion and "get the ball rolling," it focused mainly on point-to-point communications. MPI1 brought to the forefront a number of important standardization issues, but did not include any collective communication routines and was not thread-safe.

In November 1992, a meeting of the MPI working group was held in Minneapolis, at which it was decided to place the standardization process on a more formal footing, and to

generally adopt the procedures and organization of the High Performance Fortran Forum. Subcommittees were formed for the major component areas of the standard, and an email discussion service established for each. In addition, the goal of producing a draft MPI standard by the Fall of 1993 was set. To achieve this goal the MPI working group met every 6 weeks for two days throughout the first 9 months of 1993, and presented the draft MPI standard at the Supercomputing 93 conference in November 1993. These meetings and the email discussion together constituted the MPI Forum, membership of which has been open to all members of the high performance computing community.

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

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

A complete list of goals follows.

- Design an application programming interface (not necessarily for compilers or a system implementation library).
- Allow efficient communication: Avoid memory-to-memory copying and allow overlap of computation and communication and offload to communication co-processor, where available.
- Allow for implementations that can be used in a heterogeneous environment.
- Allow convenient C and Fortran 77 bindings for the interface.
- Assume a reliable communication interface: the user need not cope with communication failures. Such failures are dealt with by the underlying communication subsystem.
- Define an interface that is not too different from current practice, such as PVM, NX, Express, p4, etc., and provides extensions that allow greater flexibility.
- Define an interface that can be implemented on many vendor's platforms, with no significant changes in the underlying communication and system software.
- Semantics of the interface should be language independent.
- The interface should be designed to allow for thread-safety.

## 1.2 Who Should Use This Standard?

This standard is intended for use by all those who want to write portable message-passing programs in Fortran 77 and C. This includes individual application programmers, developers of software designed to run on parallel machines, and creators of environments and tools. In order to be attractive to this wide audience, the standard must provide a simple, easy-to-use interface for the basic user while not semantically precluding the high-performance message-passing operations available on advanced machines.

## 1.3 What Platforms Are Targets For Implementation?

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

The interface is suitable for use by fully general MIMD programs, as well as those written in the more restricted style of SPMD. Although no explicit support for threads is provided, the interface has been designed so as not to prejudice their use. With this version of MPI no support is provided for dynamic spawning of tasks.

MPI provides many features intended to improve performance on scalable parallel computers with specialized interprocessor communication hardware. Thus, we expect that native, high-performance implementations of MPI will be provided on such machines. At the same time, implementations of MPI on top of standard Unix interprocessor communication protocols will provide portability to workstation clusters and heterogenous networks of workstations. Several proprietary, native implementations of MPI, and a public domain, portable implementation of MPI are in progress at the time of this writing [17, 13].

## 1.4 What Is Included In The Standard?

The standard includes:

- Point-to-point communication
- Collective operations
- Process groups
- Communication contexts
- Process topologies
- Bindings for Fortran 77 and C
- Environmental Management and inquiry
- Profiling interface

## 1.5 What Is Not Included In The Standard?

The standard does not specify:

- Explicit shared-memory operations
- Operations that require more operating system support than is currently standard; for example, interrupt-driven receives, remote execution, or active messages

- Program construction tools
- Debugging facilities
- Explicit support for threads
- Support for task management
- I/O functions

There are many features that have been considered and not included in this standard. This happened for a number of reasons, one of which is the time constraint that was self-imposed in finishing the standard. Features that are not included can always be offered as extensions by specific implementations. Perhaps future versions of MPI will address some of these issues.

## 1.6 Organization of this Document

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

- Chapter 2, MPI Terms and Conventions, explains notational terms and conventions used throughout the MPI document.
- Chapter 3, Point to Point Communication, defines the basic, pairwise communication subset of MPI. *send* and *receive* are found here, along with many associated functions designed to make basic communication powerful and efficient.
- Chapter 4, Collective Communications, defines process-group collective communication operations. Well known examples of this are barrier and broadcast over a group of processes (not necessarily all the processes).
- Chapter 5, Groups, Contexts, and Communicators, shows how groups of processes are formed and manipulated, how unique communication contexts are obtained, and how the two are bound together into a *communicator*.
- Chapter 6, Process Topologies, explains a set of utility functions meant to assist in the mapping of process groups (a linearly ordered set) to richer topological structures such as multi-dimensional grids.
- Chapter 7, MPI Environmental Management, explains how the programmer can manage and make inquiries of the current MPI environment. These functions are needed for the writing of correct, robust programs, and are especially important for the construction of highly-portable message-passing programs.
- Chapter 8, Profiling Interface, explains a simple name-shifting convention that any MPI implementation must support. One motivation for this is the ability to put performance profiling calls into MPI without the need for access to the MPI source code. The name shift is merely an interface, it says nothing about how the actual profiling should be done and in fact, the name shift can be useful for other purposes.

- Annex A, Language Bindings, gives specific syntax in Fortran 77 and C, for all MPI functions, constants, and types.
- The MPI Function Index is a simple index showing the location of the precise definition of each MPI function, together with both C and Fortran bindings.

# Chapter 2

# MPI Terms and Conventions

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

## 2.1 Document Notation

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

*Advice to users.* Throughout this document, material that speaks to 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 uses but does not update an argument marked IN,
- the call may update an argument marked OUT,
- the call both uses and updates an argument 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.

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.

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.

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 shown, and below this, a version of the same function in Fortran 77.

## 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, a different handle type is defined for each category of objects. These should be types that support assignment and equality operators.

In Fortran, the handle can be an index to a table of opaque objects in system table; in C it can be such index or a pointer to the object. More bizarre possibilities exist.

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 deallocate 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.

MPI provides certain predefined opaque objects and predefined, static handles to these objects. Such objects may not be destroyed.

> *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, objects in system space, allows space-reclaiming, 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 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 begining 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.

### 2.4.3 State

MPI procedures use at various places arguments with *state* types. The values of such 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 (Section 7).

### 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 mecha-

nism for providing such arguments will differ from language to language. For Fortran, the document uses <type> to represent a choice variable, for 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.

## 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 and C++ implementations use the Fortran 77 and ANSI C bindings, respectively. Although we consider it premature to define other bindings to Fortran 90 and C++, the current bindings are designed to encourage, rather than discourage, experimentation with better bindings that might be adopted later.

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

There are several important language binding issues not addressed by this standard. This standard does not discuss the interoperability of message passing between languages. It is fully expected that many implementations will have such features, and that such features are a sign of the quality of the implementation.

### 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 are functions, which do not have the return code argument. The return code value for successful completion is MPI_SUCCESS. Other error codes are implementation dependent; see Chapter 7.

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.
- MPI identifiers may contain underscores after the first character.

```
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.

- 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 (PARAMETERs) 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.
- 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 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.

## 2.6 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.

MPI does not provide mechanisms to specify the initial allocation of processes to an MPI computation and their binding to physical processors. It is expected that vendors will provide mechanisms to do so either at load time or at run time. Such mechanisms will allow the specification of the initial number of required processes, the code to be executed by each initial process, and the allocation of processes to processors. Also, the current proposal does not provide for dynamic creation or deletion of processes during program execution (the total number of processes is fixed), although it is intended to be consistent with such extensions. Finally, we always identify processes according to their relative rank in a group, that is, consecutive integers in the range `0..groupsize-1`.

## 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. The error handling facilities described in section 7.2 can be used to restrict the scope of an unrecoverable error, or design error recovery at the application level.

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 section 7.2.

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.

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 desireable 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

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 com-

plete 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 program is 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.

# Chapter 3

# Point-to-Point Communication

## 3.1 Introduction

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

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

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

the send operation associates an **envelope** with the message. This envelope specifies the message destination and contains distinguishing information that can be used by the **receive** operation to select a particular message. The last three parameters of the send operation specify the envelope for the message sent.

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

The next sections describe the blocking send and receive operations. We discuss send, receive, blocking communication semantics, type matching requirements, type conversion in heterogeneous environments, and more general communication modes. Nonblocking communication is addressed next, followed by channel-like constructs and send-receive operations. We then consider general datatypes that allow one to transfer efficiently heterogeneous and noncontiguous data. We conclude with the description of calls for explicit packing and unpacking of messages.

## 3.2 Blocking Send and Receive Operations

### 3.2.1 Blocking send

The syntax of the blocking send operation is given below.

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

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

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

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

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

### 3.2.2 Message data

The send buffer specified by the MPI_SEND operation consists of count successive entries of the type indicated by datatype, starting with the entry at address buf. Note that we specify

the message length in terms of number of *elements*, not number of *bytes*. The former is machine independent and closer to the application level.

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

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

Possible values for this argument for C and the corresponding C types are listed below.

| MPI datatype | C datatype |
|---|---|
| MPI_CHAR | signed char |
| MPI_SHORT | signed short int |
| MPI_INT | signed int |
| MPI_LONG | signed long int |
| MPI_UNSIGNED_CHAR | unsigned char |
| MPI_UNSIGNED_SHORT | unsigned short int |
| MPI_UNSIGNED | unsigned int |
| MPI_UNSIGNED_LONG | unsigned long int |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_LONG_DOUBLE | long double |
| MPI_BYTE | |
| MPI_PACKED | |

The datatypes MPI_BYTE and MPI_PACKED do not correspond to a Fortran or C datatype. A value of type MPI_BYTE consists of a byte (8 binary digits). A byte is uninterpreted and is different from a character. Different machines may have different representations for characters, or may use more than one byte to represent characters. On the other hand, a byte has the same binary value on all machines. The use of the type MPI_PACKED is explained in Section 3.13.

MPI requires support of the datatypes listed above, which match the basic datatypes of Fortran 77 and ANSI C. Additional MPI datatypes should be provided if the host language

has additional data types: MPI_LONG_LONG_INT, for (64 bit) C integers declared to be of type longlong int; MPI_DOUBLE_COMPLEX for double precision complex in Fortran declared to be of type DOUBLE PRECISION; MPI_REAL2, MPI_REAL4 and MPI_REAL8 for Fortran reals, declared to be of type REAL*2, REAL*4 and REAL*8, respectively; MPI_INTEGER1 MPI_INTEGER2 and MPI_INTEGER4 for Fortran integers, declared to be of type INTEGER*1, INTEGER*2 and INTEGER*4, respectively; etc.

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

### 3.2.3 Message envelope

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

source
destination
tag
communicator

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

The message destination is specified by the dest argument.

The integer-valued message tag is specified by the tag argument. This integer can be used by the program to distinguish different types of messages. The range of valid tag values is 0,...,UB, where the value of UB is implementation dependent. It can be found by querying the value of the attribute MPI_TAG_UB, as described in Chapter 7. MPI requires that UB be no less than 32767.

The comm argument specifies the **communicator** that is used for the send operation. Communicators are explained in Chapter 5; below is a brief summary of their usage.

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

The communicator also specifies the set of processes that share this communication context. This **process group** is ordered and processes are identified by their rank within this group. Thus, the range of valid values for dest is 0, ... , n-1, where n is the number of processes in the group. (If the communicator is an inter-communicator, then destinations are identified by their rank in the remote group. See Chapter 5.)

A predefined communicator MPI_COMM_WORLD is provided by MPI. It allows communication with all processes that are accessible after MPI initialization and processes are identified by their rank in the group of MPI_COMM_WORLD.

*Advice to users.* Users that are comfortable with the notion of a flat name space for processes, and a single communication context, as offered by most existing communication libraries, need only use the predefined variable MPI_COMM_WORLD as the

comm argument. This will allow communication with all the processes available at initialization time.

Users may define new communicators, as explained in Chapter 5. Communicators provide an important encapsulation mechanism for libraries and modules. They allow modules to have their own disjoint communication universe and their own process numbering scheme. (*End of advice to users.*)

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

### 3.2.4 Blocking receive

The syntax of the blocking receive operation is given below.

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

| | | |
|---|---|---|
| OUT | buf | initial address of receive buffer (choice) |
| IN | count | number of elements in receive buffer (integer) |
| IN | datatype | datatype of each receive buffer element (handle) |
| IN | source | rank of source (integer) |
| IN | tag | message tag (integer) |
| IN | comm | communicator (handle) |
| OUT | status | status object (Status) |

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

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

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

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

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

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

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

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

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

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

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

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

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

### 3.2.5 Return status

The source or tag of a received message may not be known if wildcard values were used in the receive operation. The information is returned by the status argument of MPI_RECV. The type of status is MPI-defined. Status variables need to be explicitly allocated by the user, that is, they are not system objects.

In C, status is a structure that contains two fields named MPI_SOURCE and MPI_TAG, and the structure may contain additional fields. Thus, status.MPI_SOURCE and status.MPI_TAG contain the source and tag, respectively, of the received message.

In Fortran, status is an array of INTEGERs of size MPI_STATUS_SIZE. The two constants MPI_SOURCE and MPI_TAG are the indices of the entries that store the source and tag fields. Thus status(MPI_SOURCE) and status(MPI_TAG) contain, respectively, the source and the tag of the received message.

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

MPI_GET_COUNT(status, datatype, count)

| | | |
|---|---|---|
| IN | status | return status of receive operation (Status) |
| IN | datatype | datatype of each receive buffer element (handle) |
| OUT | count | number of received elements (integer) |

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

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

Returns the number of elements received. (Again, we count *elements*, not *bytes*.) The datatype argument should match the argument provided by the receive call that set the status variable. (We shall later see, in Section 3.12.5, that MPI_GET_COUNT may return, in certain situations, the value MPI_UNDEFINED.)

> *Rationale.* Some message passing libraries use INOUT count, tag and source arguments, thus using them both to specify the selection criteria for incoming messages and return the actual envelope values of the received message. The use of a separate status argument prevents errors that are often attached with INOUT argument (e.g., using the MPI_ANY_TAG constant as the tag in a send). Some libraries use calls that refer implicitly to the "last message received." This is not thread safe.
>
> The datatype argument is passed to MPI_GET_COUNT so as to improve performance. A message might be received without counting the number of elements it contains, and the count value is often not needed. Also, this allows the same function to be used after a call to MPI_PROBE. (*End of rationale.*)

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

## 3.3 Data type matching and data conversion

### 3.3.1 Type matching rules

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

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

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

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

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

The types of a send and receive match (phase two) if both operations use identical names. That is, MPI_INTEGER matches MPI_INTEGER, MPI_REAL matches MPI_REAL, and so on. There is one exception to this rule, discussed in Sec. 3.13, the type MPI_PACKED can match any other type.

The type of a variable in a host program matches the type specified in the communication operation if the datatype name used by that operation corresponds to the basic type of the host program variable. For example, an entry with type name MPI_INTEGER matches a Fortran variable of type INTEGER. A table giving this correspondence for Fortran and C appears in Sec. 3.2.2. There are two exceptions to this last rule: an entry with type name MPI_BYTE or MPI_PACKED can be used to match any byte of storage (on a byte-addressable machine), irrespective of the datatype of the variable that contains this byte. The type MPI_PACKED is used to send data that has been explicitly packed, or receive data that will be explicitly unpacked, see Section 3.13. The type MPI_BYTE allows one to transfer the binary value of a byte in memory unchanged.

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

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

The following examples illustrate the first two cases.

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

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

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

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

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

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

**Example 3.3** Sender and receiver specify communication of untyped values.

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

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

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

### Type MPI_CHARACTER

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

**Example 3.4** Transfer of Fortran CHARACTERs.

```
CHARACTER*10 a
CHARACTER*10 b

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

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

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

A Fortran character variable is a constant length string, with no special termination symbol. There is no fixed convention on how to represent characters, and how to store their length. Some compilers pass a character argument to a routine as a pair of arguments, one holding the address of the string and the other holding the length of string. Consider the case of an MPI communication call that is passed a communication buffer with type defined by a derived datatype (Section 3.12). If this communicator buffer contains variables of type CHARACTER then the information on their length will not be passed to the MPI routine.

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

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

### 3.3.2 Data conversion

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

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

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

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

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

If a value sent in a message is untyped (i.e., of type MPI_BYTE), then the binary representation of the byte stored at the receiver is identical to the binary representation of the byte loaded at the sender. This holds true, whether sender and receiver run in the same or in distinct environments. No representation conversion is required. (Note that

representation conversion may occur when values of type MPI_CHARACTER or MPI_CHAR are transferred, for example, from an EBCDIC encoding to an ASCII encoding.)

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

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

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

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

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

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

MPI does not require support for inter-language communication. The behavior of a program is undefined if messages are sent by a C process and received by a Fortran process, or vice-versa.

> *Rationale.* MPI does not handle inter-language communication because there are no agreed standards for the correspondence between C types and Fortran types. Therefore, MPI programs that mix languages would not port. (*End of rationale.*)
>
> *Advice to implementors.* MPI implementors may want to support inter-language communication by allowing Fortran programs to use "C MPI types," such as MPI_INT, MPI_CHAR, etc., and allowing C programs to use Fortran types. (*End of advice to implementors.*)

## 3.4 Communication Modes

The send call described in Section 3.2.1 is **blocking**: it does not return until the message data and envelope have been safely stored away so that the sender is free to access and

overwrite the send buffer. The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer.

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

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

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

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

There are three additional communication modes.

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

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

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

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

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

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

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

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

Send in buffered mode.

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

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

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

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

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

Send in synchronous mode.

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

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

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

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

Send in ready mode.

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

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

*Rationale.* We prohibit read accesses to a send buffer while it is being used, even though the send operation is not supposed to alter the content of this buffer. This may seem more stringent than necessary, but the additional restriction causes little loss of functionality and allows better performance on some systems — consider the case where data transfer is done by a DMA engine that is not cache-coherent with the main processor. (*End of rationale.*)

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

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

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

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

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

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

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

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

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

A standard send can be implemented as a synchronous send. In such a case, no data buffering is needed. However, many (most?) users expect some buffering.

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

## 3.5 Semantics of point-to-point communication

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

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

If a process has a single thread of execution, then any two communications executed by this process are ordered. On the other hand, if the process is multi-threaded, then the semantics of thread execution may not define a relative order between two send operations executed by two distinct threads. The operations are logically concurrent, even if one physically precedes the other. In such a case, the two messages sent can be received in any order. Similarly, if two receive operations that are logically concurrent receive two successively sent messages, then the two messages can match the two receives in either order.

**Example 3.5** An example of non-overtaking messages.

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

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

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

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

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

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

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

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

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

available for buffering will be much smaller than program data memory on many systems. Then, it will be easy to write programs that overrun available buffer space.

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

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

In some situations, a lack of buffer space leads to deadlock situations. This is illustrated by the examples below.

**Example 3.7** An exchange of messages.

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

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

**Example 3.8** An attempt to exchange messages.

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

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

**Example 3.9** An exchange that relies on buffering.

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

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

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

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

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

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

## 3.6 Buffer allocation and usage

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

MPI_BUFFER_ATTACH( buffer, size)

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

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

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

Provides to MPI a buffer in the user's memory to be used for buffering outgoing messages. The buffer is used only by messages sent in buffered mode. Only one buffer can be attached to a process at a time.

MPI_BUFFER_DETACH( buffer, size)

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

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

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

Detach the buffer currently associated with MPI. This operation will block until all messages currently in the buffer have been transmitted. Upon return of this function, the user may reuse or deallocate the space taken by the buffer.

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

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

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

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

or controlling buffering that would be compatible with all these choices, yet provide meaningful information. (*End of rationale.*)

### 3.6.1 Model implementation of buffered mode

The model implementation uses the packing and unpacking functions described in Section 3.13 and the nonblocking communication functions described in Section 3.7.

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

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

- Traverse sequentially the PME queue from head towards the tail, deleting all entries for communications that have completed, up to the first entry with an uncompleted request; update queue head to point to that entry.
- Compute the number, **n**, of bytes needed to store entry for new message (length of packed message computed with MPI_PACK_SIZE plus space for request handle and pointer).
- Find the next contiguous empty space of **n** bytes in buffer (space following queue tail, or space at start of buffer if queue tail is too close to end of buffer). If space not found then raise buffer overflow error.
- Append to end of PME queue in contiguous space the new entry that contains request handle, next pointer and packed message data; MPI_PACK is used to pack data.
- Post nonblocking send (standard mode) for packed data.
- Return

## 3.7 Nonblocking communication

One can improve performance on many systems by overlapping communication and computation. This is especially true on systems where communication can be executed autonomously by an intelligent communication controller. Light-weight threads are one mechanism for achieving such overlap. An alternative mechanism that often leads to better performance is to use **nonblocking communication**. A nonblocking **send start** call initiates the send operation, but does not complete it. The send start call will return before the message was copied out of the send buffer. A separate **send complete** call is needed to complete the communication, i.e., to verify that the data has been copied out of the send buffer. With suitable hardware, the transfer of data out of the sender memory may proceed concurrently with computations done at the sender after the send was initiated and before it completed. Similarly, a nonblocking **receive start call** initiates the receive operation, but does not complete it. The call will return before a message is stored into the receive buffer. A separate **receive complete** call is needed to complete the receive operation and verify that the data has been received into the receive buffer. With suitable hardware, the transfer of data into the receiver memory may proceed concurrently with computations done after

the receive was initiated and before it completed. The use of nonblocking receives may also avoid system buffering and memory-to-memory copying, as information is provided early on the location of the receive buffer.

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

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

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

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

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

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

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

Nonblocking sends in the buffered and ready modes have a more limited impact. A nonblocking send will return as soon as possible, whereas a blocking send will return after the data has been copied out of the sender memory. The use of nonblocking sends is advantageous in these cases only if data copying can be concurrent with computation.

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

### 3.7.1 Communication Objects

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

### 3.7.2 Communication initiation

We use the same naming conventions as for blocking communication: a prefix of B, S, or R is used for buffered, synchronous or ready mode. In addition a prefix of I (for immediate) indicates that the call is nonblocking.

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

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

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

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

Start a standard mode, nonblocking send.

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

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

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

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

Start a buffered mode, nonblocking send.

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

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

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

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

Start a synchronous mode, nonblocking send.

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

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

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

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

Start a ready mode nonblocking send.

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

| | | |
|---|---|---|
| OUT | buf | initial address of receive buffer (choice) |
| IN | count | number of elements in receive buffer (integer) |
| IN | datatype | datatype of each receive buffer element (handle) |
| IN | source | rank of source (integer) |
| IN | tag | message tag (integer) |
| IN | comm | communicator (handle) |
| OUT | request | communication request (handle) |

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

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

Start a nonblocking receive.

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

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

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

### 3.7.3 Communication Completion

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

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

We shall use the following terminology. A **null** handle is a handle with value MPI_REQUEST_NULL. A persistent request and the handle to it are **inactive** if the request is not associated with any ongoing communication (see Section 3.9). A handle is **active** if it is neither null nor inactive.

MPI_WAIT(request, status)

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

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

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

A call to MPI_WAIT returns when the operation identified by request is complete. If the communication object associated with this request was created by a nonblocking send or receive call, then the object is deallocated by the call to MPI_WAIT and the request handle is set to MPI_REQUEST_NULL. MPI_WAIT is a non-local operation.

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

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

*Rationale.* This makes MPI_WAIT functionally equivalent to MPI_WAITALL with a list of length one and adds some elegance. Status is set in this way so as to prevent errors due to accesses of stale information.

Successful return of MPI_WAIT after a MPI_IBSEND implies that the user send buffer can be reused — i.e., data has been sent out or copied into a buffer attached with MPI_BUFFER_ATTACH. Note that, at this point, we can no longer cancel the send (see Sec. 3.8). If a matching receive is never posted, then the buffer cannot be freed. This runs somewhat counter to the stated goal of MPI_CANCEL (always being able to

free program space that was committed to the communication subsystem). (*End of rationale.*)

*Advice to implementors.* In a multi-threaded environment, a call to MPI_WAIT should block only the calling thread, allowing the thread scheduler to schedule another thread for execution. (*End of advice to implementors.*)

MPI_TEST(request, flag, status)

| | | |
|---|---|---|
| INOUT | request | communication request (handle) |
| OUT | flag | true if operation completed (logical) |
| OUT | status | status object (Status) |

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

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

A call to MPI_TEST returns flag = true if the operation identified by request is complete. In such a case, the status object is set to contain information on the completed operation; if the communication object was created by a nonblocking send or receive, then it is deallocated and the request handle is set to MPI_REQUEST_NULL. The call returns flag = false, otherwise. In this case, the value of the status object is undefined. MPI_TEST is a local operation.

The return status object for a receive operation carries information that can be accessed as described in section 3.2.5. The status object for a send operation carries information that can be accessed by a call to MPI_TEST_CANCELLED (see Section 3.8).

One is allowed to call MPI_TEST with a null or inactive request argument. In such a case the operation returns flag = false.

The functions MPI_WAIT and MPI_TEST can be used to complete both sends and receives.

*Advice to users.* The use of the nonblocking MPI_TEST call allows the user to schedule alternative activities within a single thread of execution. An event-driven thread scheduler can be emulated with periodic calls to MPI_TEST. (*End of advice to users.*)

**Example 3.10** Simple usage of nonblocking operations and MPI_WAIT.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0) THEN
    CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)
    **** do some computation to mask latency ****
    CALL MPI_WAIT(request, status, ierr)
ELSE
    CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr)
    **** do some computation to mask latency ****
```

```
    CALL MPI_WAIT(request, status, ierr)
END IF
```

A request object can be deallocated without waiting for the associated communication to complete, by using the following operation.

MPI_REQUEST_FREE(request)

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

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

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

Mark the request object for deallocation and set request to MPI_REQUEST_NULL. An ongoing communication that is associated with the request will be allowed to complete. The request will be deallocated only after its completion.

*Rationale.* The MPI_REQUEST_FREE mechanism is provided for reasons of performance and convenience on the sending side. (*End of rationale.*)

*Advice to users.* Once a request is freed by a call to MPI_REQUEST_FREE, it is not possible to check for the successful completion of the associated communication with calls to MPI_WAIT or MPI_TEST. Also, if an error occurs subsequently during the communication, an error code cannot be returned to the user — such an error must be treated as fatal. Questions arise as to how one knows when the operations have completed when using MPI_REQUEST_FREE. Depending on the program logic, there may be other ways in which the program knows that certain operations have completed and this makes usage of MPI_REQUEST_FREE practical. For example, an active send request could be freed when the logic of the program is such that the receiver sends a reply to the message sent — the arrival of the reply informs the sender that the send has completed and the send buffer can be reused. An active receive request should never be freed as the receiver will have no way to verify that the receive has completed and the receive buffer can be reused. (*End of advice to users.*)

**Example 3.11** An example using MPI_REQUEST_FREE.

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