IBM Parallel Environment for AIX

MPI Programming and Subroutine Reference

Version 2 Release 4

GC23-3894-03

IBM Parallel Environment for AIX

MPI Programming and Subroutine Reference

Version 2 Release 4

GC23-3894-03

Note

Before using this information and the product it supports, be sure to read the general information under “Notices” on page xi.

| **Fourth Edition (October, 1998)**

| This edition applies to Version 2, Release 4, Modification 0 of the IBM Parallel Environment for AIX (5765-543), and to all
| subsequent releases and modifications until otherwise indicated in new editions or technical newsletters.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

IBM welcomes your comments. A form for your comments appears at the back of this publication. If the form has been removed, address your comments to:

International Business Machines Corporation
Department 55JA, Mail Station P384
522 South Road
Poughkeepsie, NY 12601-5400
United States of America

FAX: (United States and Canada): 914+432-9405
FAX: (Other Countries)
Your International Access Code +1+914+432-9405

IBMLink (United States customers only): IBMUSM10(MHVRCFS)
IBM Mail Exchange: USIB6TC9 at IBMMAIL
Internet: mhvrcfs@us.ibm.com
World Wide Web: http://www.rs6000.ibm.com (select Parallel Computing)

If you would like a reply, be sure to include your name, address, telephone number, or FAX number.

Make sure to include the following in your comment or note:

Title and order number of this book
Page number or topic related to your comment

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Tables

Notices

Trademarks

About This Book

This book lists the subroutines a programmer can use when writing parallel applications along with the associated parameters, and syntax. The IBM Message Passing Interface implementation intends to comply with the requirements of the Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 1.1*, University of Tennessee, Knoxville, Tennessee, June 6, 1995 and *MPI-2: Extensions to the Message-Passing Interface*, University of Tennessee, Knoxville, Tennessee, July 18, 1997. In addition, this book provides brief introductory information regarding parallel programming.

Who Should Use This Book

This book is intended for experienced programmers who want to write parallel applications using either the C or Fortran programming languages. Readers of this book should know C and Fortran and should be familiar with AIX and UNIX commands, file formats, and special files. They should also be familiar with the Message Passing Interface (MPI) concepts. In addition, readers should be familiar with distributed-memory machines.

How This Book Is Organized

Overview of Contents

This book is divided into the following sections:

- Chapter 1, "Table of Subroutines" on page 1 lists the subroutines alphabetically along with their descriptions, type, syntax and so on. MPI_SAMPLE is included which is not an MPI function but a brief description of how each routine is structured.
- Appendix A, "MPI Subroutine Bindings: Quick Reference" on page 327 briefly lists the subroutines and their arguments. Use it as a quick reference. For detailed information on the subroutines refer to Chapter 1, "Table of Subroutines" on page 1.
- Appendix B, "Profiling Message Passing" on page 347 gives information about the name-shifted interface for Message Passing Interface (MPI).
- Appendix C, "MPI Size Limits" on page 353 gives information about the MPI size limits.
- Appendix D, "Reduction Operations" on page 355 gives additional information about reduction functions.
- Appendix E, "Parallel Utility Functions" on page 359 contains the syntax man pages of the user-callable functions that take advantage of the Parallel Operating Environment (POE).
- Appendix F, "Tracing Routines" on page 393 contains the syntax man pages for modifying trace generation for the visualization tool.
- Appendix G, "Programming Considerations for User Applications in POE" on page 411 contains various information for user applications written to run under

the IBM Parallel Environment for AIX. This includes specific considerations for Fortran, threaded, and signal-handling library applications.

- Appendix H, “Using Signals and the IBM PE Programs” on page 431 contains information for understanding how the IBM Parallel Environment for AIX (PE) calls use timer signals to manage message traffic. Sample programs are included. This section applies to the signal-handling version of the Message Passing library.
- Appendix I, “Predefined Datatypes” on page 435 contains a list of the various MPI predefined datatypes that you can use with the signal-handling library.
- Appendix J, “MPI Environment Variables Quick Reference” on page 439 lists and defines the environment variables and flags for the Message Passing Interface.

Typographic Conventions

This book uses the following typographic conventions:

Type Style	Used For
Bold	**Bold** words or characters represent system elements that you must use literally, such as command names, program names, file names, flag names, and path names. **Bold** words also indicate the first use of a term included in the glossary.
Italic	*Italic* words or characters represent variable values that you must supply. *Italics* are also used for book titles and for general emphasis in text.
`Constant width`	Examples and information that the system displays appear in `constant width` typeface.

How Fortran and C Are Documented

C is case-sensitive. Fortran is not case-sensitive. This means that unless you use the XLF complier option **-qmixed**, case does not matter in Fortran subroutine names. However, to ensure MPI standard compliant code, it is suggested that all Fortran subroutine names use uppercase. The C subroutines must be entered exactly as specified.

For the purpose of distinguishing between the C and Fortran syntax in this document, C is documented in mixed case. Fortran subroutines are documented in all upper case and are referred to as Fortran throughout the book.

For both C and Fortran, the Message Passing Interface (MPI) uses the same spelling for function names. The only distinction is in the capitalization. For the purpose of clarity, when referring to a function without specifying C or Fortran version, the function is in all uppercase.

Related Publications

IBM Parallel Environment for AIX Publications

- *IBM Parallel Environment for AIX: General Information*, (GC23-3906)
- *IBM Parallel Environment for AIX: Hitchhiker's Guide*, (GC23-3895)
- *IBM Parallel Environment for AIX: Installation*, (GC28-1981)
- *IBM Parallel Environment for AIX: Messages*, (GC28-1982)
- *IBM Parallel Environment for AIX: Operation and Use, Volume 1*, (SC28-1979)
- *IBM Parallel Environment for AIX: Operation and Use, Volume 2*, (SC28-1980)
  - Part 1: Debugging and Visualizing
  - Part 2: Profiling
- *IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference*, (GC23-3894)
- *IBM Parallel Environment for AIX: MPL Programming and Subroutine Reference*, (GC23-3893)
- *IBM Parallel Environment for AIX: Licensed Program Specifications*, (GC23-3896)

As an alternative to ordering the individual books, you can use SBOF-8588 to order the entire IBM Parallel Environment for AIX library.

Related IBM Publications

- *IBM AIX Technical References*, (SBOF-1852)
- *IBM XL Fortran Compiler for AIX Language Reference*, (SC09-1611)

Related Non-IBM Publications

- Snir, M., Otto, Steve W., Huss-Lederman, Steven, Walker, David W., Dongarra, Jack, *MPI: The Complete Reference*, The MIT Press, 1995, ISBN 0-262-69184-1.
- Gropp, W., Lusk, E., Skejellum, A., *Using MPI*, The MIT Press, 1994.

  As an alternative, you can use SR28-5757-00 to order this book through your IBM representative or IBM branch office serving your locality.
- Koelbel, Charles H., David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel, *The High Performance Fortran Handbook*, The MIT Press, 1993.
- Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 1.1*, University of Tennessee, Knoxville, Tennessee, June 6, 1995.
- Message Passing Interface Forum, *MPI-2: Extensions to the Message-Passing Interface*, University of Tennessee, Knoxville, Tennessee, July 18, 1997.

Permission to copy without fee all or part of Message Passing Interface Forum material is granted, provided the University of Tennessee copyright notice and the title of the document appear, and notice is given that copying is by permission of the University of Tennessee. ©1993, 1997 University of Tennessee, Knoxville, Tennessee.

For more information about the Message Passing Interface Forum and the MPI standards documents, see:

http://www.mpi-forum.org

National Language Support

For National Language Support (NLS), all PE components and tools display messages located in externalized message catalogs. English versions of the message catalogs are shipped with the PE program product, but your site may be using its own translated message catalogs. The AIX environment variable **NLSPATH** is used by the various PE components to find the appropriate message catalog. **NLSPATH** specifies a list of directories to search for message catalogs. The directories are searched, in the order listed, to locate the message catalog. In resolving the path to the message catalog, **NLSPATH** is affected by the values of the environment variables **LC_MESSAGES** and **LANG**. If you get an error saying that a message catalog is not found, and want the default message catalog:

ENTER **export NLSPATH=/usr/lib/nls/msg/%L/%N**

export LANG=C

The PE message catalogs are in English, and are located in the following directories:

/usr/lib/nls/msg/C
/usr/lib/nls/msg/En_US
/usr/lib/nls/msg/en_US

If your site is using its own translations of the message catalogs, consult your system administrator for the appropriate value of **NLSPATH** or **LANG**. For additional information on NLS and message catalogs, see *IBM Parallel Environment for AIX: Messages* and *IBM AIX Version 4 General Programming Concepts: Writing and Debugging Programs*

Accessing Online Information

In order to use the PE man pages or access the PE online (HTML) publications, the **ppe.pedocs** file set must first be installed. To view the PE online publications, you also need access to an HTML document browser such as Netscape. An index to the HTML files that are provided with the **ppe.pedocs** file set is installed in the **/usr/lpp/ppe.pedocs/html** directory.

Online Information Resources

If you have a question about the SP, PSSP, or a related product, the following online information resources make it easy to find the information:

- Access the new SP Resource Center by issuing the command: **/usr/lpp/ssp/bin/resource_center**

  Note that the **ssp.resctr** fileset must be installed before you can do this.

If you have the Resource Center on CD-ROM, see the readme.txt file for information on how to run it.

- Access the RS/6000 Web Site at: **http://www.rs6000.ibm.com**.

Getting the Books and the Examples Online

All of the PE books are available in Portable Document Format (PDF). They are included on the product media (tape or CD-ROM), and are part of the **ppe.pedocs** file set. If you have a question about the location of the PE softcopy books, see your system administrator.

To view the PE PDF publications, you need access to the Adobe Acrobat Reader 3.0.1. The Acrobat Reader is shipped with the AIX Version 4.3 Bonus Pack, or you can download it for free from Adobe's site:

http://www.adobe.com

As stated above, you can also view or download the PE books from the IBM RS/6000 Web site at:

http://www.rs6000.ibm.com

At the time this manual was published, the full path was:

http://www.rs6000.ibm.com/resource/aix_resource/sp_books

However, note that the structure of the RS/6000 Web site can change over time.

What's New in PE 2.4?

AIX 4.3 Support

With PE 2.4, POE supports user programs developed with AIX 4.3. It also supports programs developed with AIX 4.2, intended for execution on AIX 4.3.

Parallel Checkpoint/Restart

This release of PE provides a mechanism for temporarily saving the state of a parallel program at a specific point (*checkpointing*), and then later **restarting** it from the saved state. When a program is checkpointed, the checkpointing function captures the state of the application as well as all data, and saves it in a file. When the program is restarted, the restart function retrieves the application information from the file it saved, and the program then starts running again from the place at which it was saved.

Enhanced Job Management Function

In earlier releases of PE, POE relied on the SP Resource Manager for performing job management functions. These functions included keeping track of which nodes were available or allocated and loading the switch tables for programs performing User Space communications. LoadLeveler, which had only been used for batch job submissions in the past, is now replacing the Resource Manager as the job management system for PE. One notable effect of this change is that LoadLeveler now allows you to run more than one User Space task per node.

MPI I/O

With PE 2.4, the MPI library now includes support for a subset of MPI I/O, described by Chapter 9 of the MPI-2 document: *MPI-2: Extensions to the Message-Passing Interface, Version 2.0*. MPI-I/O provides a common programming interface, improving the portability of code that involves parallel I/O.

1024 Task Support

This release of PE supports a maximum of 1024 tasks per User Space MPI/LAPI job, as opposed to the previous release, which supported a maximum of 512 tasks. For jobs using the IP version of the MPI library, PE supports a maximum of 2048 tasks.

Enhanced Compiler Support

In this release, POE is adding support for the following compilers:

- C
- C++
- Fortran Version 5
- xlhpf

Xprofiler Enhancements

This release includes a variety of enhancements to Xprofiler, including:

- *Save Configuration* and *Load Configuration* options for saving the names of functions, currently in the display, and reloading them later in order to reconstruct the function call tree.
- An *Undo* option that lets you undo operations that involve adding or removing nodes or arcs from the function call tree.

Message Queue Facility

The **pedb** debugger now includes a message queue facility. Part of the **pedb** debugger interface, the message queue viewing feature can help you debug Message Passing Interface (MPI) applications by showing internal message request queue information. With this feature, you can view:

- A summary of the number of active messages for each task in the application. You can select criteria for the summary information based on message type and source, destination, and tag filters.
- Message queue information for a specific task.
- Detailed information about a specific message.

Chapter 1. Table of Subroutines

Table 1 lists the subroutines in alphabetical order. Refer to the appropriate section in Chapter 2 for information related to subroutine purpose, syntax, and other information.

Table 1 (Page 1 of 10). Table of Subroutines

Subroutine C/FORTRAN	Page	Type	Description
MPE_Iallgather MPE_IALLGATHER	14	Nonblocking Collective Communication	Nonblocking allgather operation.
MPE_Iallgatherv MPE_IALLGATHERV	16	Nonblocking Collective Communication	Nonblocking allgatherv operation.
MPE_Iallreduce MPE_IALLREDUCE	18	Nonblocking Collective Communication	Nonblocking allreduce operation.
MPE_Ialltoall MPE_IALLTOALL	20	Nonblocking Collective Communication	Nonblocking alltoall operation.
MPE_Ialltoallv MPE_IALLTOALLV	22	Nonblocking Collective Communication	Nonblocking alltoallv operation.
MPE_Ibarrier MPE_IBARRIER	25	Nonblocking Collective Communication	Nonblocking barrier operation.
MPE_Ibcast MPE_IBCAST	27	Nonblocking Collective Communication	Nonblocking broadcast operation.
MPE_Igather MPE_IGATHER	29	Nonblocking Collective Communication	Nonblocking gather operation.
MPE_Igatherv MPE_IGATHERV	32	Nonblocking Collective Communication	Nonblocking gatherv operation.
MPE_Ireduce MPE_IREDUCE	35	Nonblocking Collective Communication	Nonblocking reduce operation.
MPE_Ireduce_scatter MPE_IREDUCE_SCATTER	37	Nonblocking Collective Communication	Nonblocking reduce_scatter operation.
MPE_Iscan MPE_ISCAN	39	Nonblocking Collective Communication	Nonblocking scan operation.
MPE_Iscatter MPE_ISCATTER	41	Nonblocking Collective Communication	Nonblocking scatter operation.
MPE_Iscatterv MPE_ISCATTERV	44	Nonblocking Collective Communication	Nonblocking scatterv operation.

Table 1 (Page 2 of 10). Table of Subroutines

Subroutine C/FORTRAN	Page	Type	Description
MPI_Abort MPI_ABORT	47	Environment Management	Forces all tasks of an MPI job to terminate.
MPI_Address MPI_ADDRESS	48	Derived Datatype	Returns address of a location in memory.
MPI_Allgather MPI_ALLGATHER	49	Collective Communication	Collects messages from each task and distributes the resulting message to each.
MPI_Allgatherv MPI_ALLGATHERV	51	Collective Communication	Collects messages from each task and distributes the resulting message to all tasks. Messages can have variable sizes and displacements.
MPI_Allreduce MPI_ALLREDUCE	53	Collective Communication	Applies a reduction operation.
MPI_Alltoall MPI_ALLTOALL	55	Collective Communication	Sends a distinct message from each task to every task.
MPI_Alltoallv MPI_ALLTOALLV	57	Collective Communication	Sends a distinct message from each task to every task. Messages can have different sizes and displacements.
MPI_Attr_delete MPI_ATTR_DELETE	59	Communicator	Removes an attribute value from a communicator.
MPI_Attr_get MPI_ATTR_GET	60	Communicator	Retrieves an attribute value from a communicator.
MPI_Attr_put MPI_ATTR_PUT	62	Communicator	Associates an attribute value with a communicator.
MPI_Barrier MPI_BARRIER	64	Collective Communication	Blocks each task until all tasks have called it.
MPI_Bcast MPI_BCAST	65	Collective Communication	Broadcasts a message from **root** to all tasks in the group.
MPI_Bsend MPI_BSEND	67	Point-to-Point	Blocking buffered mode send.
MPI_Bsend_init MPI_BSEND_INIT	69	Point-to-Point	Creates a persistent buffered mode send request.
MPI_Buffer_attach MPI_BUFFER_ATTACH	71	Point-to-Point	Provides MPI with a message buffer for sending.
MPI_Buffer_detach MPI_BUFFER_DETACH	72	Point-to-Point	Detaches the current buffer.
MPI_Cancel | MPI_CANCEL	74	Point-to-Point File	Marks a nonblocking operation for cancellation.
MPI_Cart_coords MPI_CART_COORDS	76	Topology	Translates task rank in a communicator into cartesian task coordinates.

Table 1 (Page 3 of 10). Table of Subroutines

Subroutine C/FORTRAN	Page	Type	Description
MPI_Cart_create MPI_CART_CREATE	78	Topology	Creates a communicator containing topology information.
MPI_Cart_get MPI_CART_GET	80	Topology	Retrieves cartesian topology information from a communicator.
MPI_Cart_map MPI_CART_MAP	82	Topology	Computes placement of tasks on the physical machine.
MPI_Cart_rank MPI_CART_RANK	84	Topology	Translates task coordinates into a task rank.
MPI_Cart_shift MPI_CART_SHIFT	86	Topology	Returns shifted source and destination ranks for a task.
MPI_Cart_sub MPI_CART_SUB	88	Topology	Partitions a cartesian communicator into lower-dimensional subgroups.
MPI_Cartdim_get MPI_CARTDIM_GET	90	Topology	Retrieves the number of cartesian dimensions from a communicator.
MPI_Comm_compare MPI_COMM_COMPARE	91	Communicator	Compares the groups and contexts of two communicators.
MPI_Comm_create MPI_COMM_CREATE	92	Communicator	Creates a new intracommunicator with a given group.
MPI_Comm_dup MPI_COMM_DUP	94	Communicator	Creates a new communicator that is a duplicate of an existing communicator.
MPI_Comm_free MPI_COMM_FREE	96	Communicator	Marks a communicator for deallocation.
MPI_Comm_group MPI_COMM_GROUP	97	Task Group	Returns the group handle associated with a communicator.
MPI_Comm_rank MPI_COMM_RANK	98	Communicator	Returns the rank of the local task in the group associated with a communicator.
MPI_Comm_remote_group MPI_COMM_REMOTE_GROUP	99	Communicator	Returns the handle of the remote group of an intercommunicator.
MPI_Comm_remote_size MPI_COMM_REMOTE_SIZE	100	Communicator	Returns the size of the remote group of an intercommunicator.
MPI_Comm_size MPI_COMM_SIZE	101	Communicator	Returns the size of the group associated with a communicator.
MPI_Comm_split MPI_COMM_SPLIT	103	Communicator	Splits a communicator into multiple communicators based on **color** and **key**.
MPI_Comm_test_inter MPI_COMM_TEST_INTER	105	Communicator	Returns the type of a communicator (intra or inter).
MPI_Dims_create MPI_DIMS_CREATE	106	Topology	Defines a cartesian grid to balance tasks.

Table 1 (Page 4 of 10). Table of Subroutines

Subroutine C/FORTRAN	Page	Type	Description
MPI_Errorhandler_create MPI_ERRORHANDLER_CREATE	108	Environment Management	Registers a user defined error handler.
MPI_Errorhandler_free MPI_ERRORHANDLER_FREE	110	Environment Management	Marks an error handler for deallocation.
MPI_Errorhandler_get MPI_ERRORHANDLER_GET	111	Environment Management	Gets an error handler associated with a communicator.
MPI_Errorhandler_set MPI_ERRORHANDLER_SET	112	Environment Management	Associates a new error handler with a communicator.
MPI_Error_class MPI_ERROR_CLASS	114	Environment Management	Returns the error class for the corresponding error code.
MPI_Error_string MPI_ERROR_STRING	117	Environment Management	Returns the error string for a given error code.
MPI_File_close MPI_FILE_CLOSE	118	File	Closes a file.
MPI_File_create_errhandler MPI_FILE_CREATE_ERRHANDLER	120	Environment Management	Registers a user-defined error handler that you can associate with an open file.
MPI_File_delete MPI_FILE_DELETE	122	File	Deletes a file after pending operations to the file complete.
MPI_File_get_amode MPI_FILE_GET_AMODE	124	File	Retrieves the access mode specified when the file was opened.
MPI_File_get_atomicity MPI_FILE_GET_ATOMICITY	125	File	Retrieves the current atomicity mode in which the file is accessed
MPI_File_get_errhandler MPI_FILE_GET_ERRHANDLER	126	Environment Management	Retrieves the error handler currently associated with a file handle.
MPI_File_get_group MPI_FILE_GET_GROUP	127	File	Retrieves the group of tasks that opened the file.
MPI_File_get_info MPI_FILE_GET_INFO	128	File	Returns a new **info** object identifying the hints associated with a file.
MPI_File_get_size MPI_FILE_GET_SIZE	130	File	Retrieves the current file size.
MPI_File_get_view MPI_FILE_GET_VIEW	132	File	Retrieves the current file view.
MPI_File_iread_at MPI_FILE_IREAD_AT	134	File	Nonblocking read operation using an explicit offset.
MPI_File_iwrite_at MPI_FILE_IWRITE_AT	137	File	Nonblocking write operation using an explicit offset.
MPI_File_open MPI_FILE_OPEN	140	File	Opens a file.

Table 1 (Page 5 of 10). Table of Subroutines

Subroutine C/FORTRAN	Page	Type	Description
MPI_File_read_at MPI_FILE_READ_AT	144	File	Nonblocking read operation using an explicit offset.
MPI_File_read_at_all MPI_FILE_READ_AT_ALL	146	File	Collective version of MPI_FILE_READ_AT.
MPI_File_set_errhandler MPI_FILE_SET_ERRHANDLER	148	Environment Management	Associates a new error handler with a file.
MPI_File_set_info MPI_FILE_SET_INFO	150	File	Specifies new hints for an open file.
MPI_File_set_size MPI_FILE_SET_SIZE	151	File	Expands or truncates an open file.
MPI_File_set_view MPI_FILE_SET_VIEW	153	File	Associates a new view with an open file.
MPI_File_sync MPI_FILE_SYNC	155	File	Commits file updates of an open file to storage device(s).
MPI_File_write_at MPI_FILE_WRITE_AT	157	File	Nonblocking write operation using an explicit offset.
MPI_File_write_at_all MPI_FILE_WRITE_AT_ALL	159	File	Collective version of MPI_FILE_WRITE_AT.
MPI_Finalize MPI_FINALIZE	161	Environment Management	Terminates all MPI processing.
MPI_Gather MPI_GATHER	163	Collective Communication	Collects individual messages from each task in a group at the **root** task.
MPI_Gatherv MPI_GATHERV	165	Collective Communication	Collects individual messages from each task in **comm** at the **root** task. Messages can have different sizes and displacements.
MPI_Get_count MPI_GET_COUNT	167	Point-to-Point	Returns the number of elements in a message.
MPI_Get_elements MPI_GET_ELEMENTS	168	Derived Datatype	Returns the number of basic elements in a message.
MPI_Get_processor_name MPI_GET_PROCESSOR_NAME	170	Environment Management	Returns the name of the local processor.
MPI_Get_version MPI_GET_VERSION	171	Environment Management	Returns the version of MPI standard supported.
MPI_Graph_create MPI_GRAPH_CREATE	172	Topology	Creates a new communicator containing graph topology information.
MPI_Graph_get MPI_GRAPH_GET	174	Topology	Retrieves graph topology information from a communicator.
MPI_Graph_map MPI_GRAPH_MAP	82	Topology	Computes placement of tasks on the physical machine.

Table 1 (Page 6 of 10). Table of Subroutines

Subroutine C/FORTRAN	Page	Type	Description
MPI_Graph_neighbors MPI_GRAPH_NEIGHBORS	177	Topology	Returns the neighbors of the given task.
MPI_Graph_neighbors_count MPI_GRAPH_NEIGHBORS_COUNT	178	Topology	Returns the number of neighbors of the given task.
MPI_Graphdims_get MPI_GRAPHDIMS_GET	179	Topology	Retrieves graph topology information from a communicator.
MPI_Group_compare MPI_GROUP_COMPARE	180	Task Group	Compares the contents of two task groups.
MPI_Group_difference MPI_GROUP_DIFFERENCE	181	Task Group	Creates a new group that is the difference of two existing groups.
MPI_Group_excl MPI_GROUP_EXCL	182	Task Group	Removes selected tasks from an existing group to create a new group.
MPI_Group_free MPI_GROUP_FREE	184	Task Group	Marks a group for deallocation.
MPI_Group_incl MPI_GROUP_INCL	185	Task Group	Creates a new group consisting of selected tasks from an existing group.
MPI_Group_intersection MPI_GROUP_INTERSECTION	187	Task Group	Creates a new group that is the intersection of two existing groups.
MPI_Group_range_excl MPI_GROUP_RANGE_EXCL	188	Task Group	Creates a new group by excluding selected tasks of an existing group.
MPI_Group_range_incl MPI_GROUP_RANGE_INCL	190	Task Group	Creates a new group consisting of selected ranges of tasks from an existing group.
MPI_Group_rank MPI_GROUP_RANK	192	Task Group	Returns the rank of the local task with respect to group.
MPI_Group_size MPI_GROUP_SIZE	193	Task Group	Returns the number of tasks in a group.
MPI_Group_translate_ranks MPI_GROUP_TRANSLATE_RANKS	194	Task Group	Converts task ranks of one group into ranks of another group.
MPI_Group_union MPI_GROUP_UNION	195	Task Group	Creates a new group that is the union of two existing groups.
MPI_Ibsend MPI_IBSEND	196	Point-to-Point	Nonblocking buffered send.
MPI_Info_create MPI_INFO_CREATE	198	Info	Creates a new empty **info** object.
MPI_Info_delete MPI_INFO_DELETE	199	Info	Deletes a (**key**, **value**) pair from an **info** object.
MPI_Info_dup MPI_INFO_DUP	200	Info	Duplicates an **info** object.

Table 1 (Page 7 of 10). Table of Subroutines

Subroutine C/FORTRAN	Page	Type	Description
MPI_Info_free MPI_INFO_FREE	201	Info	Frees an **info** object and sets its handle to MPI_INFO_NULL.
MPI_Info_get MPI_INFO_GET	202	Info	Retrieves the value associated with **key** in an **info** object.
MPI_Info_get_nkeys MPI_INFO_GET_NKEYS	204	Info	Returns the number of keys defined in an **info** object.
MPI_Info_get_nthkey MPI_INFO_GET_NTHKEY	205	Info	Retrieves the **n**th key defined in an **info** object.
MPI_Info_get_valuelen MPI_INFO_GET_VALUELEN	207	Info	Retrieves the length of the value associated with a key of an **info** object.
MPI_Info_set MPI_INFO_SET	209	Info	Adds a pair (**key**, **value**) to an **info** object.
MPI_Init MPI_INIT	211	Environment Management	Initializes MPI.
MPI_Initialized MPI_INITIALIZED	213	Environment Management	Determines if MPI is initialized.
MPI_Intercomm_create MPI_INTERCOM_CREATE	214	Communicator	Returns the handle of the remote group of an intercommunicator.
MPI_Intercomm_merge MPI_INTERCOMM_MERGE	216	Communicator	Creates an intracommunicator by merging the local and the remote groups of an intercommunicator.
MPI_Iprobe MPI_IPROBE	218	Point-to-Point	Checks if a message matching **source**, **tag**, and **comm** has arrived.
MPI_Irecv MPI_IRECV	220	Point-to-Point	Nonblocking receive.
MPI_Irsend MPI_IRSEND	222	Point-to-Point	Nonblocking ready send.
MPI_Isend MPI_ISEND	224	Point-to-Point	Nonblocking standard mode send.
MPI_Issend MPI_ISSEND	226	Point-to-Point	Nonblocking synchronous mode send.
MPI_Keyval_create MPI_KEYVAL_CREATE	228	Communicator	Generates a new attribute key.
MPI_Keyval_free MPI_KEYVAL_FREE	230	Communicator	Marks an attribute key for deallocation.
MPI_Op_create MPI_OP_CREATE	231	Collective Communication	Binds a user defined reduction operation to an **op** handle.
MPI_Op_free MPI_OP_FREE	233	Collective Communication	Marks a user defined reduction operation for deallocation.

Table 1 (Page 8 of 10). Table of Subroutines

Subroutine C/FORTRAN	Page	Type	Description
MPI_Pack MPI_PACK	234	Derived Datatype	Packs the message in the specified send buffer into the specified buffer space.
MPI_Pack_size MPI_PACK_SIZE	236	Dervived Datatype	Returns the number of bytes required to hold the data.
MPI_Pcontrol MPI_PCONTROL	237	Environment Management	Provides profile control.
MPI_Probe MPI_PROBE	238	Point-to-Point	Waits until a message matching **source**, **tag**, and **comm** arrives.
MPI_Recv MPI_RECV	240	Point-to-Point	Blocking receive
MPI_Recv_init MPI_RECV_INIT	242	Point-to-Point	Creates a persistent receive request.
MPI_Reduce MPI_REDUCE	244	Collective Communication	Reduces tasks specified and places the result in **recvbuf** on **root**.
MPI_Reduce_scatter MPI_REDUCE_SCATTER	246	Collective Communication	Applies a reduction operation to the vector **sendbuf** over the set of tasks specified by **comm** and scatters the result according to the values in **recvcounts**.
MPI_Request_free MPI_REQUEST_FREE	248	Point-to-Point	Marks a request for deallocation.
MPI_Rsend MPI_RSEND	249	Point-to-Point	Blocking ready mode send.
MPI_Rsend_init MPI_RSEND_INIT	251	Point-to-Point	Creates a persistent ready mode send request.
MPI_Sample MPI_SAMPLE	12	Sample	This is not an MPI function but a brief description of how each routine is structured.
MPI_Scan MPI_SCAN	253	Collective Communication	Performs a parallel prefix reduction on data distributed across a group.
MPI_Scatter MPI_SCATTER	255	Collective Communication	Distributes individual messages from **root** to each task in **comm**.
MPI_Scatterv MPI_SCATTERV	257	Collective Communication	Distributes individual messages from **root** to each task in **comm**. Messages can have different sizes and displacements.
MPI_Send MPI_SEND	259	Point-to-Point	Blocking standard mode send.
MPI_Send_init MPI_SEND_INIT	261	Point-to-Point	Creates a persistent standard mode send request.

Table 1 (Page 9 of 10). Table of Subroutines

Subroutine C/FORTRAN	Page	Type	Description
MPI_Sendrecv MPI_SENDRECV	263	Point-to-Point	A blocking send and receive operation.
MPI_Sendrecv_replace MPI_SENDRECV_REPLACE	265	Point-to-Point	Blocking send and receive operation using a common buffer.
MPI_Ssend MPI_SSEND	267	Point-to-Point	Blocking synchronous mode send.
MPI_Ssend_init MPI_SSEND_INIT	269	Point-to-Point	Creates a persistent synchronous mode send request.
MPI_Start MPI_START	271	Point-to-Point	Activates a persistent request operation.
MPI_Startall MPI_STARTALL	272	Point-to-Point	Activates a collection of persistent request operations.
MPI_Test | MPI_TEST	274	Point-to-Point File	Checks to see if a nonblocking operation has completed.
MPI_Test_cancelled | MPI_TEST_CANCELLED	276	Point-to-Point File	Tests whether a nonblocking operation was cancelled.
MPI_Testall | MPI_TESTALL	277	Point-to-Point File	Tests a collection of nonblocking operations for completion.
MPI_Testany | MPI_TESTANY	279	Point-to-Point File	Tests for the completion of any specified nonblocking operation.
MPI_Testsome | MPI_TESTSOME	281	Point-to-Point File	Tests a collection of nonblocking operations for completion.
MPI_Topo_test MPI_TOPO_TEST	283	Topology	Returns the type of virtual topology associated with a communicator.
MPI_Type_commit MPI_TYPE_COMMIT	284	Derived Datatype	Makes a datatype ready for use in communications.
MPI_Type_contiguous MPI_TYPE_CONTIGUOUS	286	Derived Datatype	Returns a new datatype that represents the concatenation of **count** instances of **oldtype**.
| MPI_Type_create_darray | MPI_TYPE_CREATE_DARRAY	288	Derived Datatype	Generates the datatypes corresponding to an HPF-like distribution of an **ndims**-dimensional array of **oldtype** elements onto an **ndims**-dimensional grid of logical tasks.
| MPI_Type_create_subarray | MPI_TYPE_CREATE_SUBARRAY	291	Derived Datatype	Returns a new datatype that represents an **ndims**-dimensional subarray of an **ndims**-dimensional array.
MPI_Type_extent MPI_TYPE_EXTENT	293	Derived Datatype	Returns the extent of any defined datatype.
| MPI_Type_free | MPI_TYPE_FREE	294	Derived Datatype	Marks a derived datatype for deallocation and sets its handle to MPI_DATATYPE_NULL.

Table 1 (Page 10 of 10). Table of Subroutines

Subroutine C/FORTRAN	Page	Type	Description
MPI_Type_get_contents MPI_TYPE_GET_CONTENTS	295	Derived Datatype	Obtains the arguments used in the creation of the datatype.
MPI_Type_get_envelope MPI_TYPE_GET_ENVELOPE	299	Derived Datatype	Determines the constructor that was used to create the datatype.
MPI_Type_hindexed MPI_TYPE_HINDEXED	301	Derived Datatype	Returns a new datatype that represents **count** distinct blocks with offsets expressed in bytes.
MPI_Type_hvector MPI_TYPE_HVECTOR	303	Derived Datatype	Returns a new datatype of **count** blocks with **stride** expressed in bytes.
MPI_Type_indexed MPI_TYPE_INDEXED	305	Derived Datatype	Returns a new datatype that represents **count** blocks with stride in terms of defining type.
MPI_Type_lb MPI_TYPE_LB	307	Derived Datatype	Returns the lower bound of a datatype.
MPI_Type_size MPI_TYPE_SIZE	308	Derived Datatype	Returns the number of bytes represented by any defined datatype.
MPI_Type_struct MPI_TYPE_STRUCT	309	Derived Datatype	Returns a new datatype that represents **count** blocks each with a distinct format and offset.
MPI_Type_ub MPI_TYPE_UB	311	Derived Datatype	Returns the upper bound of a datatype.
MPI_Type_vector MPI_TYPE_VECTOR	312	Derived Datatype	Returns a new datatype that represents equally spaced blocks of replicated data.
MPI_Unpack MPI_UNPACK	314	Derived Datatype	Unpacks the message into the specified receive buffer from the specified packed buffer.
MPI_Wait MPI_WAIT	316	Point-to-Point File	Waits for a nonblocking operation to complete.
MPI_Waitall MPI_WAITALL	318	Point-to-Point File	Waits for a collection of nonblocking operations to complete.
MPI_Waitany MPI_WAITANY	320	Point-to-Point File	Waits for any specified nonblocking operation to complete.
MPI_Waitsome MPI_WAITSOME	322	Point-to-Point File	Waits for at least one of a list of nonblocking operations to complete.
MPI_Wtick MPI_WTICK	324	Environment Management	Returns the resolution of MPI_Wtime in seconds.
MPI_Wtime MPI_WTIME	325	Environment Management	Returns the current value of **time** as a floating point value.

Chapter 2. Descriptions of Subroutines

This chapter includes descriptions of the subroutines available for parallel programming. The subroutines are listed in alphabetical order. For each subroutine, a purpose, C synopsis, Fortran synopsis, description, notes, and error conditions are provided. Review the following sample subroutine before proceeding to better understand how the subroutine descriptions are structured.

A_SAMPLE, A_Sample

Purpose

Shows how the subroutines described in this book are structured.

C Synopsis

Header file *mpi.h* supplies ANSI-C prototypes for every function described in the message passing subroutine section of this manual.

```
#include <mpi.h>
int A_Sample (one or more parameters);
```

In the C prototype, a declaration of **void *** indicates that a pointer to any datatype is allowable.

Fortran Synopsis

```
include 'mpif.h'
A_SAMPLE (ONE OR MORE PARAMETERS);
```

In the Fortran routines, formal parameters are described using a subroutine prototype format, even though Fortran does not support prototyping. The term *CHOICE* indicates that any Fortran datatype is valid.

Parameters

| Argument or parameter definitions appear below:

parameter1 parameter description (type)

...

parameter4 parameter description (type)

Parameter types:

IN - call uses but does not update an argument
OUT - call returns information via an argument but does not use its input value
INOUT - call uses and updates an argument

Description

This section contains a more detailed description of the subroutine or function.

Notes

If applicable, this section contains notes about the IBM MPI implementation and its relationship to the requirements of the MPI Standard. The IBM implementation intends to comply fully with the requirements of the MPI Standard. There are issues, however, which the Standard leaves open to the implementation's choice.

Errors

For non-file-handle errors, a single list appears here.

For errors on a file handle, up to 3 lists appear:

- *Fatal Errors:*

  Non-recoverable errors are listed here.

- *Returning Errors (MPI Error Class):*

  Errors that by default return an error code to the caller appear here. These are normally recoverable errors and the error class is specified to allow you to identify the failure cause.

- *Errors Returned By Completion Routine (MPI Error Class):*

  Errors that by default return an error code to the caller at one of the WAIT or TEST calls appear here. These are normally recoverable errors and the error class is specified to allow you to identify the failure cause.

In almost every routine, the C version is invoked as a function returning integer. In the Fortran version, the routine is called as a subroutine; that is, it has no return value. The Fortran version includes a return code parameter **IERROR** as the last parameter.

Related Information

This section contains a list of related functions or routines in this book.

For both C and Fortran, the Message-Passing Interface (MPI) uses the same spelling for function names. The only distinction is the capitalization. For the purpose of clarity, when referring to a function without specifying Fortran or C version, all uppercase letters are used.

Fortran refers to Fortran 77 (F77) bindings, which are officially supported for MPI. However, F77 bindings for MPI can be used by Fortran 90. Fortran 90 and High Performance Fortran (HPF) offer array section and assumed shape arrays as parameters on calls. These are not safe with MPI.

MPE_IALLGATHER, MPE_Iallgather

Purpose

Performs a nonblocking allgather operation.

C Synopsis

```
#include <mpi.h>
int MPE_Iallgather(void* sendbuf,int sendcount,MPI_Datatype sendtype,
    void* recvbuf,int recvcount,MPI_Datatype recvtype,MPI_Comm comm,
    MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPE_IALLGATHER(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,
    CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER COMM,
    INTEGER REQUEST,INTEGER IERROR)
```

Parameters

sendbuf	is the starting address of the send buffer (choice) (IN)
sendcount	is the number of elements in the send buffer (integer) (IN)
sendtype	is the datatype of the send buffer elements (handle) (IN)
recvbuf	is the address of the receive buffer (choice) (OUT)
recvcount	is the number of elements received from any task (integer) (IN)
recvtype	is the datatype of the receive buffer elements (handle) (IN)
comm	is the communicator (handle) (IN)
request	is the communication request (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine is a nonblocking version of MPI_ALLGATHER. It performs the same function as MPI_ALLGATHER except that it returns a **request** handle that must be explicitly completed by using one of the MPI wait or test operations.

Notes

The MPE prefix used with this routine indicates that it is an IBM extension to the MPI standard and is not part of the standard itself. MPE routines are provided to enhance the function and the performance of your applications, but applications that use them will not be directly portable to other MPI implementations.

Nonblocking collective communication routines allow for increased efficiency and flexibility in some applications. Because these routines do not synchronize the participating tasks like blocking collective communication routines generally do, tasks running at different speeds do not waste time waiting for each other.

When it is expected that tasks will be reasonably synchronized, the blocking collective communication routines provided by standard MPI will commonly give better performance then the nonblocking versions.

The nonblocking collective routines can be used in conjunction with the MPI blocking collective routines and can be completed by any of the MPI wait or test functions. Use of MPI_REQUEST_FREE and MPI_CANCEL is not supported.

| Beginning with Parallel Environment for AIX Version 2.4, the thread library has a limit of 7 outstanding nonblocking collective calls. A nonblocking call is considered outstanding between the time the call is made and the time the wait is completed. This restriction does not apply to the signal library. It does not apply to any call defined by the MPI standard.

Applications using nonblocking collective calls often provide their best performance when run in interrupt mode.

When you use this routine in a threaded application, make sure all collective operations on a particular communicator are started in the same order at each task. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid communicator	
Invalid communicator type	must be intracommunicator
Invalid count(s)	**count** < 0
Invalid datatype(s)	
Type not committed	
Unequal message length	
MPI not initialized	
MPI already finalized	

Develop mode error if:

Inconsistent message length

Related Information

MPI_ALLGATHER

MPE_IALLGATHERV, MPE_Iallgatherv

Purpose

Performs a nonblocking allgatherv operation.

C Synopsis

```
#include <mpi.h>
int MPE_Iallgatherv(void* sendbuf,int sendcount,
    MPI_Datatype sendtype,void* recvbuf,int recvcounts,
    int *displs,MPI_Datatype recvtype,
    MPI_Comm comm,MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPE_IALLGATHERV(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,
     CHOICE RECVBUF,INTEGER RECVCOUNTS(*),INTEGER DISPLS(*),
     INTEGER RECVTYPE,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Parameters

sendbuf	is the starting address of the send buffer (choice) (IN)
sendcount	is the number of elements in the send buffer (integer) (IN)
sendtype	is the datatype of the send buffer elements (handle) (IN)
recvbuf	is the address of the receive buffer (choice) (OUT)
recvcounts	integer array (of length group size) that contains the number of elements received from each task (IN)
displs	integer array (of length group size). Entry **i** specifies the displacement (relative to **recvbuf**) at which to place the incoming data from task **i** (IN)
recvtype	is the datatype of the receive buffer elements (handle) (IN)
comm	is the communictor (handle) (IN)
request	is the communication request (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine is a nonblocking version of MPI_ALLGATHERV. It performs the same function as MPI_ALLGATHERV except that it returns a **request** handle that must be explicitly completed by using one of the MPI wait or test operations.

Notes

The MPE prefix used with this routine indicates that it is an IBM extension to the MPI standard and is not part of the standard itself. MPE routines are provided to enhance the function and the performance of user applications, but applications that use them will not be directly portable to other MPI implementations.

Nonblocking collective communication routines allow for increased efficiency and flexibility in some applications. Because these routines do not synchronize the participating tasks like blocking collective routines generally do, tasks running at different speeds do not waste time waiting for each other.

When it is expected that tasks will be reasonably synchronized, the blocking collective communication routines provided by standard MPI will commonly give better performance then the nonblocking versions.

The nonblocking collective routines can be used in conjunction with the MPI blocking collective routines and can be completed by any of the MPI wait or test functions. Use of MPI_REQUEST_FREE and MPI_CANCEL is not supported.

| Beginning with Parallel Environment for AIX Version 2.4, the thread library has a
| limit of 7 outstanding nonblocking collective calls. A nonblocking call is considered
| outstanding between the time the call is made and the time the wait is completed.
| This restriction does not apply to the signal library. It does not apply to any call
| defined by the MPI standard.

Applications using nonblocking collective calls often provide their best performance when run in interrupt mode.

When you use this routine in a threaded application, make sure all collective operations on a particular communicator are started in the same order at each task. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid communicator	
Invalid communicator type	must be intracommunicator
Invalid count(s)	**count** < 0
Invalid datatype(s)	
Type not committed	
Unequal message length	
MPI not initialized	
MPI already finalized	

Develop mode error if:

None

Related Information

MPI_ALLGATHERV

MPE_IALLREDUCE, MPE_Iallreduce

Purpose

Performs a nonblocking allreduce operation.

C Synopsis

```
#include <mpi.h>
int MPE_Iallreduce(void* sendbuf,void* recvbuf,int count,
       MPI_Datatype datatype,MPI_Op op,MPI_Comm comm,
       MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPE_IALLREDUCE(CHOICE SENDBUF,CHOICE RECVBUF,INTEGER COUNT,
    INTEGER DATATYPE,INTEGER OP,INTEGER COMM,INTEGER REQUEST,
    INTEGER IERROR)
```

Parameters

sendbuf	is the starting address of the send buffer (choice) (IN)
recvbuf	is the starting address of the receive buffer (choice) (OUT)
count	is the number of elements in the send buffer (integer) (IN)
datatype	is the datatype of elements in the send buffer (handle) (IN)
op	is the reduction operation (handle) (IN)
comm	is the communicator (handle) (IN)
request	is the communication request (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine is a nonblocking version of MPI_ALLREDUCE. It performs the same function as MPI_ALLREDUCE except that it returns a **request** handle that must be explicitly completed by using one of the MPI wait or test operations.

Notes

The MPE prefix used with this routine indicates that it is an IBM extension to the MPI standard and is not part of the standard itself. MPE routines are provided to enhance the function and the performance of user applications, but applications that use them will not be directly portable to other MPI implementations.

Nonblocking collective communication routines allow for increased efficiency and flexibility in some applications. Because these routines do not synchronize the participating tasks like blocking collective routines generally do, tasks running at different speeds do not waste time waiting for each other.

When it is expected that tasks will be reasonably synchronized, the blocking collective communication routines provided by standard MPI will commonly give better performance then the nonblocking versions.

The nonblocking collective routines can be used in conjunction with the MPI blocking collective routines and can be completed by any of the MPI wait or test functions. Use of MPI_REQUEST_FREE and MPI_CANCEL is not supported.

| Beginning with Parallel Environment for AIX Version 2.4, the thread library has a
| limit of 7 outstanding nonblocking collective calls. A nonblocking call is considered
| outstanding between the time the call is made and the time the wait is completed.
| This restriction does not apply to the signal library. It does not apply to any call
| defined by the MPI standard.

Applications using nonblocking collective calls often provide their best performance when run in interrupt mode.

When you use this routine in a threaded application, make sure all collective operations on a particular communicator are started in the same order at each task. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid count	**count** < 0
Invalid datatype	
Type not committed	
Invalid op	
Invalid communicator	
Invalid communicator type	must be intracommunicator
Unequal message length	
MPI not initialized	
MPI already finalized	

Develop mode error if:

Inconsistent op

Inconsistent datatype

Inconsistent message length

Related Information

MPI_ALLREDUCE

MPE_IALLTOALL, MPE_Ialltoall

Purpose

Performs a nonblocking alltoall operation.

C Synopsis

```
#include <mpi.h>
int MPE_Ialltoall(void* sendbuf,int sendcount,MPI_Datatype sendtype,
    void* recvbuf,int recvcount,MPI_Datatype recvtype,MPI_Comm comm,
    MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPE_IALLTOALL(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,
    CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER COMM,
    INTEGER REQUEST,INTEGER IERROR)
```

Parameters

sendbuf	is the starting address of the send buffer (choice) (IN)
sendcount	is the number of elements sent to each task (integer) (IN)
sendtype	is the datatype of the send buffer elements (handle) (IN)
recvbuf	is the address of the receive buffer (choice) (OUT)
recvcount	is the number of elements received from any task (integer) (IN)
recvtype	is the datatype of the receive buffer elements (handle) (IN)
comm	is the communicator (handle) (IN)
request	is the communication request (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine is a nonblocking version of MPI_ALLTOALL. It performs the same function as MPI_ALLTOALL except that it returns a **request** handle that must be explicitly completed by using one of the MPI wait or test operations.

Notes

The MPE prefix used with this routine indicates that it is an IBM extension to the MPI standard and is not part of the standard itself. MPE routines are provided to enhance the function and the performance of user applications, but applications that use them will not be directly portable to other MPI implementations.

Nonblocking collective communication routines allow for increased efficiency and flexibility in some applications. Because these routines do not synchronize the participating tasks like blocking collective routines generally do, tasks running at different speeds do not waste time waiting for each other.

When it is expected that tasks will be reasonably synchronized, the blocking collective communication routines provided by standard MPI will commonly give better performance then the nonblocking versions.

Nonblocking collective function can be used in conjunction with the MPI blocking collective routines and can be completed by any of the MPI wait or test functions. Use of MPI_REQUEST_FREE and MPI_CANCEL is not supported.

| Beginning with Parallel Environment for AIX Version 2.4, the thread library has a
| limit of 7 outstanding nonblocking collective calls. A nonblocking call is considered
| outstanding between the time the call is made and the time the wait is completed.
| This restriction does not apply to the signal library. It does not apply to any call
| defined by the MPI standard.

Applications using nonblocking collective calls often provide their best performance when run in interrupt mode.

When you use this routine in a threaded application, make sure all collective operations on a particular communicator are started in the same order at each task. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid count(s)	**count** < 0
Invalid datatype(s)	
Type not committed	
Invalid communicator	
Invalid communicator type	must be intracommunicator
Unequal message lengths	
MPI not initialized	
MPI already finalized	

Develop mode error if:

Inconsistent message lengths

Related Information

MPI_ALLTOALL

MPE_IALLTOALLV, MPE_Ialltoallv

Purpose

Performs a nonblocking alltoallv operation.

C Synopsis

```
#include <mpi.h>
int MPE_Ialltoallv(void* sendbuf,int *sendcounts,int *sdispls,
    MPI_Datatype sendtype,void* recvbuf,int *recvcounts,int *rdispls,
    MPI_Datatype recvtype,MPI_Comm comm,MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPE_ALLTOALLV(CHOICE SENDBUF,INTEGER SENDCOUNTS(*),
    INTEGER SDISPLS(*),INTEGER SENDTYPE,CHOICE RECVBUF,
    INTEGER RECVCOUNTS(*),INTEGER RDISPLS(*),INTEGER RECVTYPE,
    INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Parameters

sendbuf	is the starting address of the send buffer (choice) (IN)
sendcounts	integer array (of length group size) specifying the number of elements to send to each task (IN)
sdispls	integer array (of length group size). Entry **j** specifies the displacement relative to **sendbuf** from which to take the outgoing data destined for task **j**. (IN)
sendtype	is the datatype of the send buffer elements (handle) (IN)
recvbuf	is the address of the receive buffer (choice) (OUT)
recvcounts	integer array (of length group size) specifying the number of elements that can be received from each task (IN)
rdispls	integer array (of length group size). Entry **i** specifies the displacement relative to **recvbuf** at which to place the incoming data from task **i**. (IN)
recvtype	is the datatype of the receive buffer elements (handle) (IN)
comm	is the communicator (handle) (IN)
request	is the communication request (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine is a nonblocking version of MPI_ALLTOALLV. It performs the same function as MPI_ALLTOALLV except that it returns a **request** handle that must be explicitly completed by using one of the MPI wait or test operations.

Notes

The MPE prefix used with this routine indicates that it is an IBM extension to the MPI standard and is not part of the standard itself. MPE routines are provided to enhance the function and the performance of user applications, but applications that use them will not be directly portable to other MPI implementations.

Nonblocking collective communication routines allow for increased efficiency and flexibility in some applications. Because these routines do not synchronize the participating tasks like blocking collective routines generally do, tasks running at different speeds do not waste time waiting for each other.

When it is expected that tasks will be reasonably synchronized, the blocking collective communication routines provided by standard MPI will commonly give better performance then the nonblocking versions.

The nonblocking collective routines can be used in conjunction with the MPI blocking collective routines and can be completed by any of the MPI wait or test functions. Use of MPI_REQUEST_FREE and MPI_CANCEL is not supported.

Applications using nonblocking collective calls often provide their best performance when run in interrupt mode.

Beginning with Parallel Environment for AIX Version 2.4, the thread library has a limit of 7 outstanding nonblocking collective calls. A nonblocking call is considered outstanding between the time the call is made and the time the wait is completed. This restriction does not apply to the signal library. It does not apply to any call defined by the MPI standard.

When you use this routine in a threaded application, make sure all collective operations on a particular communicator are started in the same order at each task. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid count(s)	**count** < 0
Invalid datatype(s)	
Type not committed	
Invalid communicator	
Invalid communicator type	must be intracommunicator
A send and receive have unequal message lengths	
MPI not initialized	
MPI already finalized	

Related Information

MPI_ALLTOALLV

MPE_IBARRIER, MPE_Ibarrier

Purpose

Performs a nonblocking barrier operation.

C Synopsis

```
#include <mpi.h>
int MPE_Ibarrier(MPI_Comm comm,MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPE_IBARRIER(INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Parameters

comm	is a communicator (handle) (IN)
request	is the communication request (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine is a nonblocking version of MPI_BARRIER. It returns immediately, without blocking, but will not complete (via MPI_WAIT or MPI_TEST) until all group members have called it.

Notes

The MPE prefix used with this routine indicates that it is an IBM extension to the MPI standard and is not part of the standard itself. MPE routines are provided to enhance the function and the performance of user applications, but applications that use them will not be directly portable to other MPI implementations.

When it is expected that tasks will be reasonably synchronized, the blocking collective communication routines provided by standard MPI will commonly give better performance then the nonblocking versions.

A typical use of MPE_IBARRIER is to make a call to it, and then periodically test for completion with MPI_TEST. Completion indicates that all tasks in **comm** have arrived at the barrier. Until then, computation can continue.

| Beginning with Parallel Environment for AIX Version 2.4, the thread library has a
| limit of 7 outstanding nonblocking collective calls. A nonblocking call is considered
| outstanding between the time the call is made and the time the wait is completed.
| This restriction does not apply to the signal library. It does not apply to any call
| defined by the MPI standard.

Applications using nonblocking collective calls often provide their best performance when run in interrupt mode.

When you use this routine in a threaded application, make sure all collective operations on a particular communicator are started in the same order at each task.

See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid communicator

Invalid communicator type must be intracommunicator

MPI not initialized

MPI already finalized

Related Information

MPI_BARRIER

MPE_IBCAST, MPE_Ibcast

Purpose

Performs a nonblocking broadcast operation.

C Synopsis

```
#include <mpi.h>
int MPE_Ibcast(void* buffer,int count,MPI_Datatype datatype,
    int root,MPI_Comm comm,MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPE_IBCAST(CHOICE BUFFER,INTEGER COUNT,INTEGER DATATYPE,INTEGER ROOT,
    INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Parameters

buffer	is the starting address of the buffer (choice) (INOUT)
count	is the number of elements in the buffer (integer) (IN)
datatype	is the datatype of the buffer elements (handle) (IN)
root	is the rank of the root task (integer) (IN)
comm	is the communicator (handle) (IN)
request	is the communication request (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine is a nonblocking version of MPI_BCAST. It performs the same function as MPI_BCAST except that it returns a **request** handle that must be explicitly completed by using one of the MPI wait or test operations.

Notes

The MPE prefix used with this routine indicates that it is an IBM extension to the MPI standard and is not part of the standard itself. MPE routines are provided to enhance the function and the performance of user applications, but applications that use them will not be directly portable to other MPI implementations.

Nonblocking collective communication routines allow for increased efficiency and flexibility in some applications. Because these routines do not synchronize the participating tasks like blocking collective routines generally do, tasks running at different speeds do not waste time waiting for each other.

When it is expected that tasks will be reasonably synchronized, the blocking collective communication routines provided by standard MPI will commonly give better performance then the nonblocking versions.

The nonblocking collective routines can be used in conjunction with the MPI blocking collective routines and can be completed by any of the MPI wait or test functions. Use of MPI_REQUEST_FREE and MPI_CANCEL is not supported.

| Beginning with Parallel Environment for AIX Version 2.4, the thread library has a limit of 7 outstanding nonblocking collective calls. A nonblocking call is considered outstanding between the time the call is made and the time the wait is completed. This restriction does not apply to the signal library. It does not apply to any call defined by the MPI standard.

Applications using nonblocking collective calls often provide their best performance when run in interrupt mode.

When you use this routine in a threaded application, make sure all collective operations on a particular communicator are started in the same order at each task. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Error Conditions:

Invalid communicator	
Invalid communicator type	must be intracommunicator
Invalid count	**count** < 0
Invalid datatype	
Type not committed	
Invalid root	**root** < 0 or **root** >= groupsize
Unequal message length	
MPI not initialized	
MPI already finalized	

Develop mode error if:

Inconsistent root

Inconsistent message length

Related Information

MPI_BCAST

MPE_IGATHER, MPE_Igather

Purpose

Performs a nonblocking gather operation.

C Synopsis

```
#include <mpi.h>
int MPE_Igather(void* sendbuf,int sendcount,MPI_Datatype sendtype,
    void* recvbuf,int recvcount,MPI_Datatype recvtype,int root,
    MPI_Comm comm,MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPE_IGATHER(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,
    CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER ROOT,
    INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Parameters

sendbuf	is the starting address of the send buffer (choice) (IN)
sendcount	is the number of elements in the send buffer (integer) (IN)
sendtype	is datatype of the send buffer elements (integer) (IN)
recvbuf	is the address of the receive buffer (choice, significant only at **root**) (OUT)
recvcount	is the number of elements for any single receive (integer, significant only at **root**) (IN)
recvtype	is the datatype of the receive buffer elements (handle, significant at **root**) (IN)
root	is the rank of the receiving task (integer) (IN)
comm	is the communicator (handle) (IN)
request	is the communication request (handle) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine is a nonblocking version of MPI_GATHER. It performs the same function as MPI_GATHER except that it returns a **request** handle that must be explicitly completed by using one of the MPI wait or test operations.

Notes

The MPE prefix used with this routine indicates that it is an IBM extension to the MPI standard and is not part of the standard itself. MPE routines are provided to enhance the function and the performance of user applications, but applications that use them will not be directly portable to other MPI implementations.

Nonblocking collective communication routines allow for increased efficiency and flexibility in some applications. Because these routines do not synchronize the participating tasks like blocking collective routines generally do, tasks running at different speeds do not waste time waiting for each other.

When it is expected that tasks will be reasonably synchronized, the blocking collective communication routines provided by standard MPI will commonly give better performance then the nonblocking versions.

The nonblocking collective routines can be used in conjunction with the MPI blocking collective routines and can be completed by any of the MPI wait or test functions. Use of MPI_REQUEST_FREE and MPI_CANCEL is not supported.

Beginning with Parallel Environment for AIX Version 2.4, the thread library has a limit of 7 outstanding nonblocking collective calls. A nonblocking call is considered outstanding between the time the call is made and the time the wait is completed. This restriction does not apply to the signal library. It does not apply to any call defined by the MPI standard.

Applications using nonblocking collective calls often provide their best performance when run in interrupt mode.

When you use this routine in a threaded application, make sure all collective operations on a particular communicator are started in the same order at each task. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid communicator	
Invalid communicator type	must be intracommunicator
Invalid count(s)	**count** < 0
Invalid datatype(s)	
Type not committed	
Invalid root	**root** <0 or **root** >= groupsize
Unequal message lengths	
MPI not initialized	
MPI already finalized	

Develop mode error if:

Inconsistent root

Inconsistent message lengths

Related Information

MPI_GATHER

MPE_IGATHERV, MPE_Igatherv

Purpose

Performs a nonblocking gatherv operation.

C Synopsis

```
#include <mpi.h>
int MPE_Igatherv(void* sendbuf,int sendcount,MPI_Datatype sendtype,
    void* recvbuf,int recvcounts,int *displs,MPI_Datatype recvtype,
    int root,MPI_Comm comm,MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPE_IGATHERV(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,
    CHOICE RECVBUF,INTEGER RECVCOUNTS(*),INTEGER DISPLS(*),
    INTEGER RECVTYPE,INTEGER ROOT,INTEGER COMM,INTEGER REQUEST,
    INTEGER IERROR)
```

Parameters

sendbuf	is the starting address of the send buffer (choice) (IN)
sendcount	is the number of elements to be sent (integer) (IN)
sendtype	is the datatype of the send buffer elements (handle) (IN)
recvbuf	is the address of the receive buffer (choice, significant only at **root**) (OUT)
recvcounts	integer array (of length group size) that contains the number of elements received from each task (significant only at **root**) (IN)
displs	integer array (of length group size). Entry **i** specifies the displacement relative to **recvbuf** at which to place the incoming data from task **i** (significant only at **root**) (IN)
recvtype	is the datatype of the receive buffer elements (handle, significant only at **root**) (IN)
root	is the rank of the receiving task (integer) (IN)
comm	is the communicator (handle) (IN)
request	is the communication request (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine is a nonblocking version of MPI_GATHERV. It performs the same function as MPI_GATHERV except that it returns a **request** handle that must be explicitly completed by using one of the MPI wait or test operations.

Notes

The MPE prefix used with this routine indicates that it is an IBM extension to the MPI standard and is not part of the standard itself. MPE routines are provided to enhance the function and the performance of user applications, but applications that use them will not be directly portable to other MPI implementations.

Nonblocking collective communication routines allow for increased efficiency and flexibility in some applications. Because these routines do not synchronize the participating tasks like blocking collective routines generally do, tasks running at different speeds do not waste time waiting for each other.

When it is expected that tasks will be reasonably synchronized, the blocking collective communication routines provided by standard MPI will commonly give better performance then the nonblocking versions.

The nonblocking collective routines can be used in conjunction with the MPI blocking collective routines and can be completed by any of the MPI wait or test functions. Use of MPI_REQUEST_FREE and MPI_CANCEL is not supported.

| Beginning with Parallel Environment for AIX Version 2.4, the thread library has a
| limit of 7 outstanding nonblocking collective calls. A nonblocking call is considered
| outstanding between the time the call is made and the time the wait is completed.
| This restriction does not apply to the signal library. It does not apply to any call
| defined by the MPI standard.

Applications using nonblocking collective calls often provide their best performance when run in interrupt mode.

When you use this routine in a threaded application, make sure all collective operations on a particular communicator are started in the same order at each task. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid communicator	
Invalid communicator type	must be intracommunicator
Invalid count(s)	
Invalid datatype(s)	
Type not committed	
Invalid root	**root** < 0 or **root** >= groupsize
A send and receive have unequal message lengths	
MPI not initialized	
MPI already finalized	

Develop mode error if:

Inconsistent root

Related Information

MPI_GATHERV

MPE_IREDUCE, MPE_Ireduce

Purpose

Performs a nonblocking reduce operation.

C Synopsis

```
#include <mpi.h>
int MPE_Ireduce(void* sendbuf,void* recvbuf,int count,
    MPI_Datatype datatype,MPI_Op op,int root,MPI_Comm comm,
    MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPE_IREDUCE(CHOICE SENDBUF,CHOICE RECVBUF,INTEGER COUNT,
    INTEGER DATATYPE,INTEGER OP,INTEGER ROOT,INTEGER COMM,
    INTEGER REQUEST,INTEGER IERROR)
```

Parameters

sendbuf	is the address of the send buffer (choice) (IN)
recvbuf	is the address of the receive buffer (choice, significant only at root) (OUT)
count	is the number of elements in the send buffer (integer) (IN)
datatype	is the datatype of elements of the send buffer (handle) (IN)
op	is the reduction operation (handle) (IN)
root	is the rank of the root task (integer) (IN)
comm	is the communicator (handle) (IN)
request	is the communication request (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine is a nonblocking version of MPI_REDUCE. It performs the same function as MPI_REDUCE except that it returns a **request** handle that must be explicitly completed by using one of the MPI wait or test operations.

Notes

The MPE prefix used with this routine indicates that it is an IBM extension to the MPI standard and is not part of the standard itself. MPE routines are provided to enhance the function and the performance of user applications, but applications that use them will not be directly portable to other MPI implementations.

Nonblocking collective communication routines allow for increased efficiency and flexibility in some applications. Because these routines do not synchronize the participating tasks like blocking collective routines generally do, tasks running at different speeds do not waste time waiting for each other.

When it is expected that tasks will be reasonably synchronized, the blocking collective communication routines provided by standard MPI will commonly give better performance then the nonblocking versions.

The nonblocking collective routines can be used in conjunction with the MPI blocking collective routines and can be completed by any of the MPI wait or test functions. Use of MPI_REQUEST_FREE and MPI_CANCEL is not supported.

| Beginning with Parallel Environment for AIX Version 2.4, the thread library has a
| limit of 7 outstanding nonblocking collective calls. A nonblocking call is considered
| outstanding between the time the call is made and the time the wait is completed.
| This restriction does not apply to the signal library. It does not apply to any call
| defined by the MPI standard.

Applications using nonblocking collective calls often provide their best performance when run in interrupt mode.

When you use this routine in a threaded application, make sure all collective operations on a particular communicator are started in the same order at each task. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid count	**count** < 0
Invalid datatype	
Type not committed	
Invalid op	
Invalid root	**root** < 0 or **root** > = groupsize
Invalid communicator	
Invalid communicator type	must be intracommunicator
Unequal message lengths	
MPI not initialized	
MPI already finalized	

Develop mode error if:

Inconsistent op

Inconsistent datatype

Inconsistent root

Inconsistent message length

Related Information

MPI_REDUCE

MPE_IREDUCE_SCATTER, MPE_Ireduce_scatter

Purpose

Performs a nonblocking reduce_scatter operation.

C Synopsis

```
#include <mpi.h>
int MPE_Ireduce_scatter(void* sendbuf,void* recvbuf,int *recvcounts,
    MPI_Datatype datatype,MPI_Op op,MPI_Comm comm,
    MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPE_IREDUCE_SCATTER(CHOICE SENDBUF,CHOICE RECVBUF,
     INTEGER RECVCOUNTS(*),INTEGER DATATYPE,INTEGER OP,
     INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Parameters

sendbuf	is the starting address of the send buffer (choice) (IN)
recvbuf	is the starting address of the receive buffer (choice) (OUT)
recvcounts	integer array specifying the number of elements in result distributed to each task. Must be identical on all calling tasks. (IN)
datatype	is the datatype of elements in the input buffer (handle) (IN)
op	is the reduction operation (handle) (IN)
comm	is the communicator (handle) (IN)
request	is the communication request (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine is a nonblocking version of MPI_REDUCE_SCATTER. It performs the same function as MPI_REDUCE_SCATTER except that it returns a **request** handle that must be explicitly completed by using one of the MPI wait or test operations.

Notes

The MPE prefix used with this routine indicates that it is an IBM extension to the MPI standard and is not part of the standard itself. MPE routines are provided to enhance the function and the performance of user applications, but applications that use them will not be directly portable to other MPI implementations.

Nonblocking collective communication routines allow for increased efficiency and flexibility in some applications. Because these routines do not synchronize the participating tasks like blocking collective routines generally do, tasks running at different speeds do not waste time waiting for each other.

When it is expected that tasks will be reasonably synchronized, the blocking collective communication routines provided by standard MPI will commonly give better performance then the nonblocking versions.

The nonblocking collective routines can be used in conjunction with the MPI blocking collective routines and can be completed by any of the MPI wait or test functions. Use of MPI_REQUEST_FREE and MPI_CANCEL is not supported.

| Beginning with Parallel Environment for AIX Version 2.4, the thread library has a
| limit of 7 outstanding nonblocking collective calls. A nonblocking call is considered
| outstanding between the time the call is made and the time the wait is completed.
| This restriction does not apply to the signal library. It does not apply to any call
| defined by the MPI standard.

Applications using nonblocking collective calls often provide their best performance when run in interrupt mode.

When you use this routine in a threaded application, make sure all collective operations on a particular communicator are started in the same order at each task. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid recvcount(s)	**recvcounts(i)** < 0
Invalid datatype	
Type not committed	
Invalid op	
Invalid communicator	
Invalid communicator type	must be intracommunicator
Unequal message lengths	
MPI not initialized	
MPI already finalized	

Develop mode error if:

Inconsistent op

Inconsistent datatype

Related Information

MPI_REDUCE_SCATTER

MPE_ISCAN, MPE_Iscan

Purpose

Performs a nonblocking scan operation.

C Synopsis

```
#include <mpi.h>
int MPE_Iscan(void* sendbuf,void* recvbuf,int count,
    MPI_Datatype datatype,MPI_Op op,MPI_Comm comm,
    MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPE_ISCAN(CHOICE SENDBUF,CHOICE RECVBUF,INTEGER COUNT,
    INTEGER DATATYPE,INTEGER OP,INTEGER COMM,INTEGER REQUEST,
    INTEGER IERROR)
```

Parameters

sendbuf	is the starting address of the send buffer (choice) (IN)
recvbuf	is the starting address of the receive buffer (choice) (OUT)
count	is the number of elements in **sendbuf** (integer) (IN)
datatype	is the datatype of elements in **sendbuf** (handle) (IN)
op	is the reduction operation (handle) (IN)
comm	is the communicator (IN)
request	is the communication request (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine is a nonblocking version of MPI_SCAN. It performs the same function as MPI_SCAN except that it returns a **request** handle that must be explicitly completed by using one of the MPI wait or test operations.

Notes

The MPE prefix used with this routine indicates that it is an IBM extension to the MPI standard and is not part of the standard itself. MPE routines are provided to enhance the function and the performance of user applications, but applications that use them will not be directly portable to other MPI implementations.

Nonblocking collective communication routines allow for increased efficiency and flexibility in some applications. Because these routines do not synchronize the participating tasks like blocking collective routines generally do, tasks running at different speeds do not waste time waiting for each other.

When it is expected that tasks will be reasonably synchronized, the blocking collective communication routines provided by standard MPI will commonly give better performance then the nonblocking versions.

The nonblocking collective routines can be used in conjunction with the MPI blocking collective routines and can be completed by any of the MPI wait or test functions. Use of MPI_REQUEST_FREE and MPI_CANCEL is not supported.

| Beginning with Parallel Environment for AIX Version 2.4, the thread library has a
| limit of 7 outstanding nonblocking collective calls. A nonblocking call is considered
| outstanding between the time the call is made and the time the wait is completed.
| This restriction does not apply to the signal library. It does not apply to any call
| defined by the MPI standard.

Applications using nonblocking collective calls often provide their best performance when run in interrupt mode.

When you use this routine in a threaded application, make sure all collective operations on a particular communicator are started in the same order at each task. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid count	**count** < 0
Invalid datatype	
Type not committed	
Invalid op	
Invalid communicator	
Invalid communicator type	must be intracommunicator
Unequal message lengths	
MPI not initialized	
MPI already finalized	

Develop mode error if:

Inconsistent op

Inconsistent datatype

Inconsistent message length

Related Information

MPI_SCAN

MPE_ISCATTER, MPE_Iscatter

Purpose

Performs a nonblocking scatter operation.

C Synopsis

```
#include <mpi.h>
int MPE_Iscatter(void* sendbuf,int sendcount,MPI_Datatype sendtype,
    void* recvbuf,int recvcount,MPI_Datatype recvtype,int root,
    MPI_Comm comm,MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPE_ISCATTER(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,
    CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER ROOT,
    INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Parameters

sendbuf	is the address of the send buffer (choice, significant only at **root**) (IN)
sendcount	is the number of elements to be sent to each task (integer, significant only at **root**) (IN)
sendtype	is the datatype of the send buffer elements (handle, significant only at **root**) (IN)
recvbuf	is the address of the receive buffer (choice) (OUT)
recvcount	is the number of elements in the receive buffer (integer) (IN)
recvtype	is the datatype of the receive buffer elements (handle) (IN)
root	is the rank of the sending task (integer) (IN)
comm	is the communicator (handle) (IN)
request	communication request (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine is a nonblocking version of MPI_SCATTER. It performs the same function as MPI_SCATTER except that it returns a **request** handle that must be explicitly completed by using one of the MPI wait or test operations.

Notes

The MPE prefix used with this routine indicates that it is an IBM extension to the MPI standard and is not part of the standard itself. MPE routines are provided to enhance the function and the performance of user applications, but applications that use them will not be directly portable to other MPI implementations.

Nonblocking collective communication routines allow for increased efficiency and flexibility in some applications. Because these routines do not synchronize the participating tasks like blocking collective routines generally do, tasks running at different speeds do not waste time waiting for each other.

When it is expected that tasks will be reasonably synchronized, the blocking collective communication routines provided by standard MPI will commonly give better performance then the nonblocking versions.

The nonblocking collective routines can be used in conjunction with the MPI blocking collective routines and can be completed by any of the MPI wait or test functions. Use of MPI_REQUEST_FREE and MPI_CANCEL is not supported.

| Beginning with Parallel Environment for AIX Version 2.4, the thread library has a
| limit of 7 outstanding nonblocking collective calls. A nonblocking call is considered
| outstanding between the time the call is made and the time the wait is completed.
| This restriction does not apply to the signal library. It does not apply to any call
| defined by the MPI standard.

Applications using nonblocking collective calls often provide their best performance when run in interrupt mode.

When you use this routine in a threaded application, make sure all collective operations on a particular communicator are started in the same order at each task. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid communicator	
Invalid communicator type	must be intracommunicator
Invalid count(s)	**count** < 0
Invalid datatype(s)	
Type not committed	
Invalid root	**root** < 0 or **root** >= groupsize
Unequal message lengths	
MPI not initialized	
MPI already finalized	

Develop mode error if:

Inconsistent root

Inconsistent message length

Related Information

MPI_SCATTER

MPE_ISCATTERV, MPE_Iscatterv

Purpose

Performs a nonblocking scatterv operation.

C Synopsis

```
#include <mpi.h>
int MPE_Iscatterv(void* sendbuf,int *sendcounts,int *displs,
    MPI_Datatype sendtype,void* recvbuf,int recvcount,
    MPI_Datatype recvtype,int root,MPI_Comm comm,MPI_Comm *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPE_ISCATTERV(CHOICE SENDBUF,INTEGER SENDCOUNTS(*),INTEGER DISPLS(*),
    INTEGER SENDTYPE,CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,
    INTEGER ROOT,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Parameters

sendbuf	is the address of the send buffer (choice, significant only at **root**) (IN)
sendcounts	integer array (of length group size) that contains the number of elements to send to each task (significant only at **root**) (IN)
displs	integer array (of length group size). Entry **i** specifies the displacement relative to **sendbuf** from which to take the outgoing data to task **i** (significant only at **root**) (IN)
sendtype	is the datatype of the send buffer elements (handle, significant only at **root**) (IN)
recvbuf	is the address of the receive buffer (choice) (OUT)
recvcount	is the number of elements in the receive buffer (integer) (IN)
recvtype	is the datatype of the receive buffer elements (handle) (IN)
root	is the rank of the sending task (integer) (IN)
comm	is the communicator (handle) (IN)
request	is the communication request (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine is a nonblocking version of MPI_SCATTERV. It performs the same function as MPI_SCATTERV except that it returns a **request** handle that must be explicitly completed by using one of the MPI wait or test operations.

Notes

The MPE prefix used with this routine indicates that it is an IBM extension to the MPI standard and is not part of the standard itself. MPE routines are provided to enhance the function and the performance of user applications, but applications that use them will not be directly portable to other MPI implementations.

Nonblocking collective communication routines allow for increased efficiency and flexibility in some applications. Because these routines do not synchronize the participating tasks like blocking collective routines generally do, tasks running at different speeds do not waste time waiting for each other.

When it is expected that tasks will be reasonably synchronized, the blocking collective communication routines provided by standard MPI will commonly give better performance then the nonblocking versions.

The nonblocking collective routines can be used in conjunction with the MPI blocking collective routines and can be completed by any of the MPI wait or test functions. Use of MPI_REQUEST_FREE and MPI_CANCEL is not supported.

| Beginning with Parallel Environment for AIX Version 2.4, the thread library has a
| limit of 7 outstanding nonblocking collective calls. A nonblocking call is considered
| outstanding between the time the call is made and the time the wait is completed.
| This restriction does not apply to the signal library. It does not apply to any call
| defined by the MPI standard.

Applications using nonblocking collective calls often provide their best performance when run in interrupt mode.

When you use this routine in a threaded application, make sure all collective operations on a particular communicator are started in the same order at each task. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid communicator	
Invalid communicator type	must be intracommunicator
Invalid count(s)	**count** < 0
Invalid datatype(s)	
Type not committed	
Invalid root	**root** < 0 or **root** >= groupsize
Unequal message lengths	
MPI not initialized	
MPI already finalized	

Develop mode error if:

Inconsistent root

Related Information

MPI_SCATTERV

MPI_ABORT, MPI_Abort

Purpose

Forces all tasks of an MPI job to terminate.

C Synopsis

```
#include <mpi.h>
int MPI_Abort(MPI_Comm comm,int errorcode);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_ABORT(INTEGER COMM,INTEGER ERRORCODE,INTEGER IERROR)
```

Parameters

comm	is the communicator of the tasks to abort. (IN)
errorcode	is the error code returned to the invoking environment. (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine forces an MPI program to terminate all tasks in the job. **comm** currently is not used. All tasks in the job are aborted. The low order 8 bits of **errorcode** are returned as an AIX return code.

Notes

MPI_ABORT causes *all* tasks to exit immediately.

Errors

MPI already finalized

MPI not initialized

MPI_ADDRESS, MPI_Address

Purpose

Returns the address of a variable in memory.

C Synopsis

```
#include <mpi.h>
int MPI_Address(void* location,MPI_Aint *address);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_ADDRESS(CHOICE LOCATION,INTEGER ADDRESS,INTEGER IERROR)
```

Parameters

location	is the location in caller memory (choice) (IN)
address	is the address of location (integer) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine returns the byte address of **location**.

Notes

On the IBM RS/6000 SP, this is equivalent to **address**= **(MPI_Aint) location** in C, but the MPI_ADDRESS routine is portable to machines with less straightforward addressing.

Errors

MPI not initialized

MPI already finalized

Related Information

MPI_TYPE_INDEXED
MPI_TYPE_HINDEXED
MPI_TYPE_STRUCT

MPI_ALLGATHER, MPI_Allgather

Purpose

Gathers individual messages from each task in **comm** and distributes the resulting message to each task.

C Synopsis

```
#include <mpi.h>
int MPI_Allgather(void* sendbuf,int sendcount,MPI_Datatype sendtype,
    void* recvbuf,int recvcount,MPI_Datatype recvtype,MPI_Comm comm);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_ALLGATHER(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,
    CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER COMM,
    INTEGER IERROR)
```

Parameters

sendbuf	is the starting address of the send buffer (choice) (IN)
sendcount	is the number of elements in the send buffer (integer) (IN)
sendtype	is the datatype of the send buffer elements (handle) (IN)
recvbuf	is the address of the receive buffer (choice) (OUT)
recvcount	is the number of elements received from any task (integer) (IN)
recvtype	is the datatype of the receive buffer elements (handle) (IN)
comm	is the communicator (handle) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_ALLGATHER is similar to MPI_GATHER except that all tasks receive the result instead of just the **root**.

The block of data sent from task **j** is received by every task and placed in the **j**th block of the buffer **recvbuf**.

The type signature associated with **sendcount**, **sendtype** at a task must be equal to the type signature associated with **recvcount**, **recvtype** at any other task.

When you use this routine in a threaded application, make sure all collective operations on a particular communicator occur in the same order at each task. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid communicator	
Invalid communicator type	must be intracommunicator
Invalid count(s)	**count** < 0
Invalid datatype(s)	
Type not committed	
Unequal message length	
MPI not initialized	
MPI already finalized	

Develop mode error if:

Inconsistent message length

Related Information

MPE_IALLGATHER
MPI_ALLGATHER
MPI_GATHER

MPI_ALLGATHERV, MPI_Allgatherv

Purpose

Collects individual messages from each task in **comm** and distributes the resulting message to all tasks. Messages can have different sizes and displacements.

C Synopsis

```
#include <mpi.h>
int MPI_Allgatherv(void* sendbuf,int sendcount,MPI_Datatype sendtype,
    void* recvbuf,int *recvcounts,int *displs,MPI_Datatype recvtype,
    MPI_Comm comm);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_ALLGATHERV(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,
    CHOICE RECVBUF,INTEGER RECVCOUNTS(*),INTEGER DISPLS(*),
    INTEGER RECVTYPE,INTEGER COMM,INTEGER IERROR)
```

Parameters

sendbuf	is the starting address of the send buffer (choice) (IN)
sendcount	is the number of elements in the send buffer (integer) (IN)
sendtype	is the datatype of the send buffer elements (handle) (IN)
recvbuf	is the address of the receive buffer (choice) (OUT)
recvcounts	integer array (of length group size) that contains the number of elements received from each task (IN)
displs	integer array (of length group size). Entry **i** specifies the displacement (relative to **recvbuf**) at which to place the incoming data from task **i** (IN)
recvtype	is the datatype of the receive buffer elements (handle) (IN)
comm	is the communictor (handle) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine collects individual messages from each task in **comm** and distributes the resulting message to all tasks. Messages can have different sizes and displacements.

The block of data sent from task **j** is **recvcounts[j]** elements long, and is received by every task and placed in **recvbuf** at offset **displs[j]**.

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

When you use this routine in a threaded application, make sure all collective operations on a particular communicator occur in the same order at each task. See

Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid communicator	
Invalid communicator type	must be intracommunicator
Invalid count(s)	**count** < 0
Invalid datatype(s)	
Type not committed	
Unequal message lengths	
MPI not initialized	
MPI already finalized	

Develop mode error if:

None

Related Information

MPE_IALLGATHERV
MPI_ALLGATHER

MPI_ALLREDUCE, MPI_Allreduce

Purpose

Applies a reduction operation to the vector **sendbuf** over the set of tasks specified by **comm** and places the result in **recvbuf** on all of the tasks in **comm**.

C Synopsis

```
#include <mpi.h>
int MPI_Allreduce(void* sendbuf,void* recvbuf,int count,
    MPI_Datatype datatype,MPI_Op op,MPI_Comm comm);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_ALLREDUCE(CHOICE SENDBUF,CHOICE RECVBUF,INTEGER COUNT,
    INTEGER DATATYPE,INTEGER OP,INTEGER COMM,INTEGER IERROR)
```

Parameters

sendbuf	is the starting address of the send buffer (choice) (IN)
recvbuf	is the starting address of the receive buffer (choice) (OUT)
count	is the number of elements in the send buffer (integer) (IN)
datatype	is the datatype of elements in the send buffer (handle) (IN)
op	is the reduction operation (handle) (IN)
comm	is the communicator (handle) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine applies a reduction operation to the vector **sendbuf** over the set of tasks specified by **comm** and places the result in **recvbuf** on all of the tasks.

This routine is similar to MPI_REDUCE except the result is returned to the receive buffer of all the group members.

When you use this routine in a threaded application, make sure all collective operations on a particular communicator occur in the same order at each task. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Notes

See Appendix D, “Reduction Operations” on page 355.

Errors

Invalid count	**count** < 0
Invalid datatype	
Type not committed	
Invalid op	
Invalid communicator	
Invalid communicator type	must be intracommunicator
Unequal message lengths	
MPI not initialized	
MPI already finalized	

Develop mode error if:

Inconsistent op

Inconsistent datatype

Inconsistent message length

Related Information

MPE_IALLREDUCE
MPI_REDUCE
MPI_REDUCE_SCATTER
MPI_OP_CREATE

MPI_ALLTOALL, MPI_Alltoall

Purpose

Sends a distinct message from each task to every task.

C Synopsis

```
#include <mpi.h>
int MPI_Alltoall(void* sendbuf,int sendcount,MPI_Datatype sendtype,
    void* recvbuf,int recvcount,MPI_Datatype recvtype,
    MPI_Comm comm):
```

Fortran Synopsis

```
include 'mpif.h'
MPI_ALLTOALL(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,
    CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER COMM,
    INTEGER IERROR)
```

Parameters

sendbuf	is the starting address of the send buffer (choice) (IN)
sendcount	is the number of elements sent to each task (integer) (IN)
sendtype	is the datatype of the send buffer elements (handle) (IN)
recvbuf	is the address of the receive buffer (choice) (OUT)
recvcount	is the number of elements received from any task (integer) (IN)
recvtype	is the datatype of the receive buffer elements (handle) (IN)
comm	is the communicator (handle) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_ALLTOALL sends a distinct message from each task to every task.

The **j**th block of data sent from task **i** is received by task **j** and placed in the **i**th block of the buffer **recvbuf**.

The type signature associated with **sendcount**, **sendtype**, at a task must be equal to the type signature associated with **recvcount**, **recvtype** at any other task. This means the amount of data sent must be equal to the amount of data received, pair wise between every pair of tasks. The type maps can be different.

All arguments on all tasks are significant.

When you use this routine in a threaded application, make sure all collective operations on a particular communicator occur in the same order at each task. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Unequal lengths	
Invalid count(s)	**count** < 0
Invalid datatype(s)	
Type not committed	
Invalid communicator	
Invalid communicator type	must be intracommunicator
Unequal message lengths	
MPI not initialized	
MPI already finalized	

Develop mode error if:

Inconsistent message lengths

Related Information

MPE_IALLTOALL
MPI_ALLTOALLV

MPI_ALLTOALLV, MPI_Alltoallv

Purpose

Sends a distinct message from each task to every task. Messages can have different sizes and displacements.

C Synopsis

```
#include <mpi.h>
int MPI_Alltoallv(void* sendbuf,int *sendcounts,int *sdispls,
    MPI_Datatype sendtype,void* recvbuf,int *recvcounts,int *rdispls,
    MPI_Datatype recvtype,MPI_Comm comm);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_ALLTOALLV(CHOICE SENDBUF,INTEGER SENDCOUNTS(*),
    INTEGER SDISPLS(*),INTEGER SENDTYPE,CHOICE RECVBUF,
    INTEGER RECVCOUNTS(*),INTEGER RDISPLS(*),INTEGER RECVTYPE,
    INTEGER COMM,INTEGER IERROR)
```

Parameters

sendbuf	is the starting address of the send buffer (choice) (IN)
sendcounts	integer array (of length group size) specifying the number of elements to send to each task (IN)
sdispls	integer array (of length group size). Entry **j** specifies the displacement relative to **sendbuf** from which to take the outgoing data destined for task **j**. (IN)
sendtype	is the datatype of the send buffer elements (handle) (IN)
recvbuf	is the address of the receive buffer (choice) (OUT)
recvcounts	integer array (of length group size) specifying the number of elements to be received from each task (IN)
rdispls	integer array (of length group size). Entry **i** specifies the displacement relative to **recvbuf** at which to place the incoming data from task **i**. (IN)
recvtype	is the datatype of the receive buffer elements (handle) (IN)
comm	is the communicator (handle) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_ALLTOALLV sends a distinct message from each task to every task. Messages can have different sizes and displacements.

This routine is similar to MPI_ALLTOALL with the following differences. MPI_ALLTOALLV allows you the flexibility to specify the location of the data for the send with **sdispls** and the location of where the data will be placed on the receive with **rdispls**.

The block of data sent from task **i** is **sendcounts[j]** elements long, and is received by task **j** and placed in **recvbuf** at offset offset **rdispls[i]**. These blocks do not have to be the same size.

The type signature associated with **sendcount[j]**, **sendtype** at task **i** must be equal to the type signature associated with **recvcounts[i]**, **recvtype** at task **j**. This means the amount of data sent must be equal to the amount of data received, pair wise between every pair of tasks. Distinct type maps between sender and receiver are allowed.

All arguments on all tasks are significant.

When you use this routine in a threaded application, make sure all collective operations on a particular communicator occur in the same order at each task. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid count(s)	**count** < 0
Invalid datatype(s)	
Type not committed	
Invalid communicator	
Invalid communicator type	must be intracommunicator
A send and receive hand unequal message lengths	
MPI not initialized	
MPI already finalized	

Related Information

MPE_IALLTOALLV
MPI_ALLTOALL

MPI_ATTR_DELETE, MPI_Attr_delete

Purpose

Removes an attribute value from a communicator.

C Synopsis

```
#include <mpi.h>
int MPI_Attr_delete(MPI_Comm comm,int keyval);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_ATTR_DELETE(INTEGER COMM,INTEGER KEYVAL,INTEGER IERROR)
```

Parameters

comm	is the communicator that the attribute is attached (handle) (IN)
keyval	is the key value of the deleted attribute (integer) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine deletes an attribute from cache by key. MPI_ATTR_DELETE also invokes the attribute delete function **delete_fn** specified when the **keyval** is created.

Errors

A delete_fn did not return MPI_SUCCESS

Invalid communicator

Invalid keyval keyval is undefined

Invalid keyval keyval is predefined

MPI not initialized

MPI already finalized

Related Information

MPI_KEYVAL_CREATE

MPI_ATTR_GET, MPI_Attr_get

Purpose

Retrieves an attribute value from a communicator.

C Synopsis

```
#include <mpi.h>
int MPI_Attr_get(MPI_Comm comm,int keyval,void *attribute_val,
    int *flag);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_ATTR_GET(INTEGER COMM,INTEGER KEYVAL,INTEGER ATTRIBUTE_VAL,
    LOGICAL FLAG,INTEGER IERROR)
```

Parameters

comm	is the communicator to which attribute is attached (handle) (IN)
keyval	is the key value (integer) (IN)
attribute_val	is the attribute value unless **flag** = **false** (OUT)
flag	is **true** if an attribute value was extracted and *false* if no attribute is associated with the key. (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This function retrieves an attribute value by key. If there is no key with value **keyval**, the call is erroneous. However, the call is valid if there is a key value **keyval**, but no attribute is attached on **comm** for that key. In this case, the call returns **flag** = **false**.

Notes

The implementation of the MPI_ATTR_PUT and MPI_ATTR_GET involves saving a single word of information in the communicator. The languages C and Fortran have different approaches to using this capability:

In C: As the programmer, you normally define a struct which holds arbitrary "attribute" information. Before calling MPI_ATTR_PUT, you allocate some storage for the attribute structure and then call MPI_ATTR_PUT to record the address of this structure. You must assure that the structure remains intact as long as it may be useful. As the programmer, you will also declare a variable of type "pointer to attribute structure" and pass the address of this variable when calling MPI_ATTR_GET. Both MPI_ATTR_PUT and MPI_ATTR_GET take a void* parameter but this does not imply the same parameter is passed to either one.

In Fortran: MPI_ATTR_PUT records an INTEGER*4 and MPI_ATTR_GET returns the INTEGER*4. As the programmer, you may choose to encode all attribute information in this integer or maintain a some kind of database in which the integer can index. Either of these approaches will port to other MPI implementations.

XL Fortran has an additional feature which will allow some of the same function a C programmer would use. This is the POINTER type which is described in the *IBM XL Fortran Compiler V3.2 for AIX Language Reference* Use of this will impact the program's portability.

Errors

Invalid communicator	
Invalid keyval	keyval is undefined
MPI not initialized	
MPI already finalized	

Related Information

MPI_ATTR_PUT

MPI_ATTR_PUT, MPI_Attr_put

Purpose

Stores an attribute value in a communicator.

C Synopsis

```
#include <mpi.h>
int MPI_Attr_put(MPI_Comm comm,int keyval,void* attribute_val);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_ATTR_PUT(INTEGER COMM,INTEGER KEYVAL,INTEGER ATTRIBUTE_VAL,
INTEGER IERROR)
```

Parameters

comm	is the communicator to which attribute will be attached (handle) (IN)
keyval	is the key value as returned by MPI_KEYVAL_CREATE (integer) (IN)
attribute_val	is the attribute value (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine stores the attribute value for retrieval by MPI_ATTR_GET. Any previous value is deleted with the attribute **delete_fn** being called and the new value is stored. If there is no key with value **keyval**, the call is erroneous.

Notes

The implementation of the MPI_ATTR_PUT and MPI_ATTR_GET involves saving a single word of information in the communicator. The languages C and Fortran have different approaches to using this capability:

In C: As the programmer, you normally define a struct which holds arbitrary "attribute" information. Before calling MPI_ATTR_PUT, you allocate some storage for the attribute structure and then call MPI_ATTR_PUT to record the address of this structure. You must assure that the structure remains intact as long as it may be useful. As the programmer, you will also declare a variable of type "pointer to attribute structure" and pass the address of this variable when calling MPI_ATTR_GET. Both MPI_ATTR_PUT and MPI_ATTR_GET take a void* parameter, but this does not imply the same parameter is passed to either one.

In Fortran: MPI_ATTR_PUT records an INTEGER*4 and MPI_ATTR_GET returns the INTEGER*4. As the programmer, you may choose to encode all attribute information in this integer or maintain a some kind of database in which the integer can index. Either of these approaches will port to other MPI implementations.

XL Fortran has an additional feature which will allow some of the same function a C programmer would use. This is the POINTER type which is described in the *IBM*

XL Fortran Compiler V3.2 for AIX Language Reference Use of this will impact the program's portability.

Errors

A delete_fn did not return MPI_SUCCESS

Invalid communicator

Invalid keyval keyval is undefined

Predefined keyval cannot modify predefined attributes

MPI not initialized

MPI already finalized

Related Information

MPI_ATTR_GET
MPI_KEYVAL_CREATE

MPI_BARRIER, MPI_Barrier

Purpose

Blocks each task in **comm** until all tasks have called it.

C Synopsis

```
#include <mpi.h>
int MPI_Barrier(MPI_Comm comm);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_BARRIER(INTEGER COMM,INTEGER IERROR)
```

Parameters

comm	is a communicator (handle) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine blocks until all tasks have called it. Tasks cannot exit the operation until all group members have entered.

When you use this routine in a threaded application, make sure all collective operations on a particular communicator occur in the same order at each task. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid communicator

Invalid communicator type must be intracommunicator

MPI not initialized

MPI already finalized

Related Information

MPE_IBARRIER

MPI_BCAST, MPI_Bcast

Purpose

Broadcasts a message from **root** to all tasks in **comm**.

C Synopsis

```
#include <mpi.h>
int MPI_Bcast(void* buffer,int count,MPI_Datatype datatype,
    int root,MPI_Comm comm);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_BCAST(CHOICE BUFFER,INTEGER COUNT,INTEGER DATATYPE,INTEGER ROOT,
      INTEGER COMM,INTEGER IERROR)
```

Parameters

buffer	is the starting address of the buffer (choice) (INOUT)
count	is the number of elements in the buffer (integer) (IN)
datatype	is the datatype of the buffer elements (handle) (IN)
root	is the rank of the root task (integer) (IN)
comm	is the communicator (handle) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine broadcasts a message from **root** to all tasks in **comm**. The contents of **root**'s communication buffer is copied to all tasks on return.

The type signature of **count**, **datatype** on any task must be equal to the type signature of **count**, **datatype** at the root. This means the amount of data sent must be equal to the amount of data received, pair wise between each task and the root. Distinct type maps between sender and receiver are allowed.

When you use this routine in a threaded application, make sure all collective operations on a particular communicator occur in the same order at each task. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid communicator	
Invalid communicator type	must be intracommunicator
Invalid count	**count** < 0
Invalid datatype	
Type not committed	

Invalid root	**root** < 0 or **root** >= groupsize
Unequal message lengths	
MPI not initialized	
MPI already finalized	

Develop mode error if:

Inconsistent root

Inconsistent message length

Related Information

MPE_IBCAST

MPI_BSEND, MPI_Bsend

Purpose

Performs a blocking buffered mode send operation.

C Synopsis

```
#include <mpi.h>
int MPI_Bsend(void* buf,int count,MPI_Datatype datatype,
      int dest,int tag,MPI_Comm comm);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_BSEND(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,
      INTEGER TAG,INTEGER COMM,INTEGER IERROR)
```

Parameters

buf	is the initial address of the send buffer (choice) (IN)
count	is the number of elements in the send buffer (integer) (IN)
datatype	is the datatype of each send buffer element (handle) (IN)
dest	is the rank of destination (integer) (IN)
tag	is the message tag (integer) (IN)
comm	is the communicator (handle) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine is a blocking buffered mode send. This is a local operation. It does not depend on the occurrence of a matching receive in order to complete. If a send operation is started and no matching receive is posted, the outgoing message is buffered to allow the send call to complete.

Make sure you have enough buffer space available. An error occurs if the message must be buffered and there is there is insufficient buffer space.

Return from an MPI_BSEND does not guarantee the message was sent. It may remain in the buffer until a matching receive is posted. MPI_BUFFER_DETACH will block until all messages are received.

Errors

Invalid count	**count** < 0
Invalid datatype	
Type not committed	
Invalid destination	**dest** < 0 or **dest** > = groupsize
Invalid tag	**tag** < 0

Invalid comm

Insufficient buffer space

MPI not initialized

MPI already finalized

Related Information

MPI_IBSEND
MPI_SEND
MPI_BUFFER_ATTACH
MPI_BUFFER_DETACH

MPI_BSEND_INIT, MPI_Bsend_init

Purpose

Creates a persistent buffered mode send request.

C Synopsis

```
#include <mpi.h>
int MPI_Bsend_init(void* buf,int count,MPI_Datatype datatype,
    int dest,int tag,MPI_Comm comm,MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_BSEND_INIT(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,
      INTEGER DEST,INTEGER TAG,INTEGER COMM,INTEGER REQUEST,
      INTEGER IERROR)
```

Parameters

buf	is the initial address of the send buffer (choice) (IN)
count	is the number of elements to be sent (integer) (IN)
datatype	is the type of each element (handle) (IN)
dest	is the rank of the destination task (integer) (IN)
tag	is the message tag (integer) (IN)
comm	is the communicator (handle) (IN)
request	is the communication request (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine creates a persistent communication request for a buffered mode send operation. MPI_START or MPI_STARTALL must be called to activate the send.

Notes

See MPI_BSEND for additional information.

Because it is the MPI_START which initiates communication, any error related to insufficient buffer space occurs at the MPI_START.

Errors

Invalid count	**count** < 0
Invalid datatype	
Type not committed	
Invalid destination	**dest** < 0 or **dest** > &equals groupsize
Invalid tag	**tag** < 0

Invalid comm

MPI not initialized

MPI already finalized

Related Information

MPI_START
MPI_IBSEND

MPI_BUFFER_ATTACH, MPI_Buffer_attach

Purpose

Provides MPI with a buffer to use for buffering messages sent with MPI_BSEND and MPI_IBSEND.

C Synopsis

```
#include <mpi.h>
int MPI_Buffer_attach(void* buffer,int size);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_BUFFER_ATTACH(CHOICE BUFFER,INTEGER SIZE,INTEGER IERROR)
```

Parameters

buffer	is the initial buffer address (choice) (IN)
size	is the buffer size in bytes (integer) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine provides MPI a buffer in the user's memory which is used for buffering outgoing messages. This buffer is used only by messages sent in buffered mode, and only one buffer is attached to a task at any time.

Notes

MPI uses part of the buffer space to store information about the buffered messages. The number of bytes required by MPI for each buffered message is given by MPI_BSEND_OVERHEAD.

If a buffer is already attached, it must be detached by MPI_BUFFER_DETACH before a new buffer can be attached.

Errors

Invalid size	**size** < 0
Buffer is already attached	
MPI not initialized	
MPI already finalized	

Related Information

MPI_BUFFER_DETACH
MPI_BSEND
MPI_IBSEND

MPI_BUFFER_DETACH, MPI_Buffer_detach

Purpose

Detaches the current buffer.

C Synopsis

```
#include <mpi.h>
int MPI_Buffer_detach(void* buffer,int *size);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_BUFFER_DETACH(CHOICE BUFFER,INTEGER SIZE,INTEGER IERROR)
```

Parameters

buffer	is the initial buffer address (choice) (OUT)
size	is the buffer size in bytes (integer) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine detaches the current buffer. Blocking occurs until all messages in the active buffer are transmitted. Once this function returns, you can reuse or deallocate the space taken by the buffer. There is an implicit MPI_BUFFER_DETACH inside MPI_FINALIZE. Because a buffer detach can block, the impicit detach creates some risk that an incorrect program will hang in MPI_FINALIZE.

If there is no active buffer, MPI acts as if a buffer of size 0 is associated with the task.

Notes

It is important to detach an attached buffer *before* it is deallocated. If this is not done, any buffered message may be lost.

In Fortran 77, the **buffer** argument for MPI_BUFFER_DETACH cannot return a useful value because Fortran 77 does not support pointers. If a fully portable MPI program written in Fortran calls MPI_BUFFER_DETACH, it either passes the name of the original buffer or a throwaway temp as the **buffer** argument.

If a buffer was attached, this implementation of MPI returns the address of the freed buffer in the first word of the **buffer** argument. If the **size** being returned is zero to four bytes, MPI_BUFFER_DETACH will not modify the **buffer** argument. This implementation is harmless for a program that uses either the original buffer or a throwaway temp of at least word size as **buffer**. It also allows the programmer who wants to use an XL Fortran POINTER as the **buffer** argument to do so. Using the POINTER type will affect portability.

Errors

MPI not initialized

MPI already finalized

Related Information

MPI_BUFFER_ATTACH
MPI_BSEND
MPI_IBSEND

MPI_CANCEL, MPI_Cancel

Purpose

| Marks a nonblocking request for cancellation.

C Synopsis

```
#include <mpi.h>
int MPI_Cancel(MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_CANCEL(INTEGER REQUEST,INTEGER IERROR)
```

Parameters

request	is a communication request (handle) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

| This routine marks a nonblocking request for cancellation. The cancel call is local. It returns immediately; it can return even before the communication is actually
| cancelled. It is necessary to complete an operation marked for cancellation by
| using a call to MPI_WAIT or MPI_TEST (or any other wait or test call).

You can use MPI_CANCEL to cancel a persistent request in the same way it is used for nonpersistent requests. A successful cancellation cancels the active communication, but not the request itself. After the call to MPI_CANCEL and the subsequent call to MPI_WAIT or MPI_TEST, the request becomes inactive and can be activated for a new communication. It is erroneous to cancel an inactive persistent request.

The successful cancellation of a buffered send frees the buffer space occupied by the pending message.

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

| If the operation has been cancelled successfully, information to that effect is returned in the status argument of the operation that completes the communication, and may be retrieved by a call to MPI_TEST_CANCELLED.

Notes

Nonblocking collective communication requests cannot be cancelled. MPI_CANCEL may be called on non-blocking file operation requests. The eventual call to MPI_TEST_CANCELLED will show that the cancellation did not succeed.

Errors

Invalid request

CCL request

Cancel inactive persistent request

MPI not initialized

MPI already finalized

Related Information

MPI_TEST_CANCELLED
MPI_WAIT

MPI_CART_COORDS, MPI_Cart_coords

Purpose

Translates task rank in a communicator into cartesian task coordinates.

C Synopsis

```
#include <mpi.h>
MPI_Cart_coords(MPI_Comm comm,int rank,int maxdims,int *coords);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_CART_COORDS(INTEGER COMM,INTEGER RANK,INTEGER MAXDIMS,
      INTEGER COORDS(*),INTEGER IERROR)
```

Parameters

comm	is a communicator with cartesian topology (handle) (IN)
rank	is the rank of a task within group **comm** (integer) (IN)
maxdims	is the length of array **coords** in the calling program (integer) (IN)
coords	is an integer array specifying the cartesian coordinates of a task. (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine translates task rank in a communicator into task coordinates.

Notes

Task coordinates in a cartesian structure begin their numbering at 0. Row-major numbering is always used for the tasks in a cartesian structure.

Errors

MPI not initialized	
MPI already finalized	
Invalid communicator	
No topology	
Invalid topology	type must be cartesian
Invalid rank	**rank** < 0 or **rank** > = groupsize
Invalid array size	**maxdims** < 0

Related Information

MPI_CART_RANK
MPI_CART_CREATE

MPI_CART_CREATE, MPI_Cart_create

Purpose

Creates a communicator containing topology information.

C Synopsis

```
#include <mpi.h>
int MPI_Cart_create(MPI_Comm comm_old,int ndims,int *dims,
    int *periods,int reorder,MPI_Comm *comm_cart);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_CART_CREATE(INTEGER COMM_OLD,INTEGER NDIMS,INTEGER DIMS(*),
    INTEGER PERIODS(*),INTEGER REORDER,INTEGER COMM_CART,INTEGER IERROR)
```

Parameters

comm_old	is the input communicator (handle) (IN)
ndims	is the number of cartesian dimensions in grid (integer) (IN)
dims	is an integer array of size *ndims* specifying the number of tasks in each dimension (IN)
periods	is a logical array of size *ndims* specifying if the grid is periodic or not in each dimension (IN)
reorder	if true, ranking may be reordered. If false, then rank in **comm_cart** must be the same as in **comm_old**. (logical) (IN)
comm_cart	is a communicator with new cartesian topology (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine creates a new communicator containing cartesian topology information defined by *ndims*, **dims**, **periods** and **reorder**. MPI_CART_CREATE returns a handle for this new communicator in **comm_cart**. If there are more tasks in **comm** than required by the grid, some tasks are returned **comm_cart** = MPI_COMM_NULL. **comm_old** must be an intracommunicator.

Notes

The reorder argument is ignored.

Errors

MPI not initialized

| **Conflicting collective operations on communicator**

MPI already finalized

Invalid communicator

Invalid communicator type	must be intracommunicator
Invalid ndims	*ndims* < 0 or *ndims* > groupsize
Invalid dimension	

Related Information

MPI_CART_SUB
MPI_GRAPH_CREATE

MPI_CART_GET, MPI_Cart_get

Purpose

Retrieves cartesian topology information from a communicator.

C Synopsis

```
#include <mpi.h>
MPI_Cart_get(MPI_Comm comm,int maxdims,int *dims,int *periods,int *coords);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_CART_GET(INTEGER COMM,INTEGER MAXDIMS,INTEGER DIMS(*),
      INTEGER PERIODS(*),INTEGER COORDS(*),INTEGER IERROR)
```

Parameters

comm	is a communicator with cartesian topology (handle) (IN)
maxdims	is the length of **dims, periods,** and **coords** in the calling program (integer) (IN)
dims	is the number of tasks for each cartesian dimension (array of integer) (OUT)
periods	is a logical array specifying if each cartesian dimension is periodic or not. (OUT)
coords	is the coordinates of the calling task in the cartesian structure (array of integer) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine retrieves the cartesian topology information associated with a communicator in **dims, periods** and **coords**.

Errors

MPI not initialized	
MPI already finalized	
Invalid communicator	
No topology	
Invalid topology type	must be cartesian
Invalid array size	**maxdims** < 0

Related Information

MPI_CARTDIM_GET
MPI_CART_CREATE

MPI_CART_MAP, MPI_Cart_map

Purpose

Computes placement of tasks on the physical machine.

C Synopsis

```
#include <mpi.h>
MPI_Cart_map(MPI_Comm comm,int ndims,int *dims,int *periods,
      int *newrank);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_CART_MAP(INTEGER COMM,INTEGER NDIMS,INTEGER DIMS(*),
      INTEGER PERIODS(*),INTEGER NEWRANK,INTEGER IERROR)
```

Parameters

comm	is the input communicator (handle) (IN)
ndims	is the number of dimensions of the cartesian structure (integer) (IN)
dims	is an integer array of size *ndims* specifying the number of tasks in each coordinate direction (IN)
periods	is a logical array of size *ndims* specifying the periodicity in each coordinate direction (IN)
newrank	is the reordered rank or MPI_UNDEFINED if the calling task does not belong to the grid (integer) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_CART_MAP allows MPI to compute an optimal placement for the calling task on the physical machine by reordering the tasks in **comm**.

Notes

No reordering is done by this function; it would serve no purpose on an SP. MPI_CART_MAP returns **newrank** as the original rank of the calling task if it belongs to the grid, or MPI_UNDEFINED if it does not.

Errors

MPI not initialized

MPI already finalized

Invalid communicator

Invalid communicator type	must be intracommunicator
Invalid ndims	*ndims* < 1 or *ndims* > groupsize
Invalid dimension	*ndims*[i] <= 0

Invalid grid size	**n**< 0 or **n** > groupsize, where **n** is the product of **dims[i]**

MPI_CART_RANK, MPI_Cart_rank

Purpose

Translates task coordinates into a task rank.

C Synopsis

```
#include <mpi.h>
MPI_Cart_rank(MPI_Comm comm,int *coords,int *rank);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_CART_RANK(INTEGER COMM,INTEGER COORDS(*),INTEGER RANK,
      INTEGER IERROR)
```

Parameters

comm	is a communicator with cartesian topology (handle) (IN)
coords	is an integer array of size *ndims* specifying the cartesian coordinates of a task (IN)
rank	is an integer specifying the rank of specified task (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine translates cartesian task coordinates into a task rank.

For dimension **i** with **periods(i)** = **true**, if the coordinate **coords(i)** is out of range, that is, **coords(i)** < 0 or **coords(i)** ≥ **dims(i)**, it is shifted back to the interval 0 ≥ **coords(i)** < **dims(i)** automatically. Out of range coordinates are erroneous for non-periodic dimensions.

Notes

Task coordinates in a cartesian structure begin their numbering at 0. Row-major numbering is always used for the tasks in a cartesian structure.

Errors

MPI not initialized	
MPI already finalized	
Invalid communicator	
No topology	
Invalid topology type	must be cartesian
Invalid coordinates	refer to Description above

Related Information

MPI_CART_CREATE
MPI_CART_COORDS

MPI_CART_SHIFT, MPI_Cart_shift

Purpose

Returns shifted source and destination ranks for a task.

C Synopsis

```
#include <mpi.h>
MPI_Cart_shift(MPI_Comm comm,int direction,int disp,
      int *rank_source,int *rank_dest);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_CART_SHIFT(INTEGER COMM,INTEGER DIRECTION,INTEGER DISP,
      INTEGER RANK_SOURCE,INTEGER RANK_DEST,INTEGER IERROR)
```

Parameters

comm	is a communicator with cartesian topology (handle) (IN)
direction	is the coordinate dimension of shift (integer) (IN)
disp	is the displacement (>0 = upward shift, <0 = downward shift) (integer) (IN)
rank_source	is the rank of the source task (integer) (OUT)
rank_dest	is the rank of the destination task (integer) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine shifts the local rank along a specified coordinate dimension to generate source and destination ranks.

rank_source is obtained by subtracting **disp** from the **n**th coordinate of the local task, where **n** is equal to **direction**. Similarly, **rank_dest** is obtained by adding **disp** to the **n**th coordinate. Coordinate dimensions (**direction**) are numbered starting with **0**.

If the dimension specified by **direction** is non-periodic, off-end shifts result in the value MPI_PROC_NULL being returned for **rank_source** and/or **rank_dest**.

Notes

In C and Fortran, the coordinate is identified by counting from 0. For example, Fortran **A(X,Y)** or C **A[x] [y]** both have **x** as direction 0.

Errors

MPI not initialized

MPI already finalized

Invalid communicator

Invalid topology type	must be cartesian
No topology	

Related Information

MPI_CART_RANK
MPI_CART_COORDS
MPI_CART_CREATE

MPI_CART_SUB, MPI_Cart_sub

Purpose

Partitions a cartesian communicator into lower-dimensional subgroups.

C Synopsis

```
#include <mpi.h>
MPI_Cart_sub(MPI_Comm comm,int *remain_dims,MPI_Comm *newcomm);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_CART_SUB(INTEGER COMM,LOGICAL REMAIN_DIMS(*),INTEGER NEWCOMM,
      INTEGER IERROR)
```

Parameters

comm	is a communicator with cartesian topology (handle) (IN)
remain_dims	the **i**th entry of **remain_dims** specifies whether the **i**th dimension is kept in the subgrid or is dropped. (logical vector) (IN)
newcomm	is the communicator containing the subgrid that includes the calling task (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

If a cartesian topology was created with MPI_CART_CREATE, you can use the function MPI_CART_SUB:

- to partition the communicator group into subgroups forming lower-dimensional cartesian subgrids, and
- to build a communicator with the associated subgrid cartesian topology for each of those subgroups.

(This function is closely related to MPI_COMM_SPLIT.)

For example, MPI_CART_CREATE (..., **comm**) defined a 2 × 3 × 4 grid. Let **remain_dims** = (true, false, true). Then a call to:

```
    MPI_CART_SUB(comm,remain_dims,comm_new),
```

creates three communicators. Each has eight tasks in a 2 × 4 cartesian topology. If **remain_dims** = (false, false, true), then the call to:

```
    MPI_CART_SUB(comm,remain_dims,comm_new),
```

creates six non-overlapping communicators, each with four tasks in a one-dimensional cartesian topology.

Errors

MPI not initialized	
MPI already finalized	
Invalid communicator	
Invalid topology	must be cartesian
No topology	

Related Information

MPI_CART_CREATE
MPI_COMM_SPLIT

MPI_CARTDIM_GET, MPI_Cartdim_get

Purpose

Retrieves the number of cartesian dimensions from a communicator.

C Synopsis

```
#include <mpi.h>
MPI_Cartdim_get(MPI_Comm comm,int *ndims);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_CARTDIM_GET(INTEGER COMM,INTEGER NDIMS,INTEGER IERROR)
```

Parameters

comm	is a communicator with cartesian topology (handle) (IN)
ndims	is an integer specifying the number of dimensions of the cartesian topology (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine retrieves the number of dimensions in a cartesian topology.

Errors

Invalid communicator

No topology

Invalid topology type must be cartesian

MPI not initialized

MPI already finalized

Related Information

MPI_CART_GET
MPI_CART_CREATE

MPI_COMM_COMPARE, MPI_Comm_compare

Purpose

Compares the groups and context of two communicators.

C Synopsis

```
#include <mpi.h>
int MPI_Comm_compare(MPI_Comm comm1,MPI_Comm comm2,int *result);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_COMM_COMPARE(INTEGER COMM1,INTEGER COMM2,INTEGER RESULT,INTEGER IERROR)
```

Parameters

comm1	is the first communicator (handle) (IN)
comm2	is the second communicator (handle) (IN)
result	is an integer specifying the result. The defined values are: MPI_IDENT, MPI_CONGRUENT, MPI_SIMILAR, and MPI_UNEQUAL. (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine compares the groups and contexts of two communicators. The following is an explanation of each MPI_COMM_COMPARE defined value:

MPI_IDENT comm1 and **comm2** are handles for the identical object

MPI_CONGRUENT the underlying groups are identical in constituents and rank order (both local and remote groups for intercommunications), but are different in context

MPI_SIMILAR the group members of both communicators are the same but are different in rank order (both local and remote groups for intercommunications),

MPI_UNEQUAL if none of the above.

Errors

Invalid communicator(s)

MPI not initialized

MPI already finalized

Related Information

MPI_GROUP_COMPARE

MPI_COMM_CREATE, MPI_Comm_create

Purpose

Creates a new intracommunicator with a given group.

C Synopsis

```
#include <mpi.h>
int MPI_Comm_create(MPI_Comm comm,MPI_Group group,MPI_Comm *newcomm);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_COMM_CREATE(INTEGER COMM,INTEGER GROUP,INTEGER NEWCOMM,
      INTEGER IERROR)
```

Parameters

comm	is the communicator (handle) (IN)
group	is Group which is a subset of the group of **comm** (handle) (IN)
newcomm	is the new communicator (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_COMM_CREATE is a collective function that is invoked by all tasks in the group associated with **comm**. This routine creates a new intracommunicator **newcomm** with communication group defined by **group** and a new context. Cached information is not propagated from **comm** to **newcomm**.

For tasks that are not in **group**, MPI_COMM_NULL is returned. The call is erroneous if **group** is not a subset of the group associated with **comm**. The call is executed by all tasks in **comm** even if they do not belong to the new group.

This call applies only to intracommunicators.

Notes

MPI_COMM_CREATE provides a way to subset a group of tasks for the purpose of separate MIMD computation with separate communication space. You can use **newcomm** in subsequent calls to MPI_COMM_CREATE or other communicator constructors to further subdivide a computation into parallel sub-computations.

Errors

Conflicting collective operations on communicator	
Invalid communicator	
Invalid group	**group** is not a subset of the group associated with **comm**
MPI not initialized	

MPI already finalized

Related Information

MPI_COMM_DUP
MPI_COMM_SPLIT

MPI_COMM_DUP, MPI_Comm_dup

Purpose

Creates a new communicator that is a duplicate of an existing communicator.

C Synopsis

```
#include <mpi.h>
int MPI_Comm_dup(MPI_Comm comm,MPI_Comm *newcomm);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_COMM_DUP(INTEGER COMM,INTEGER NEWCOMM,INTEGER IERROR)
```

Parameters

comm	is the communicator (handle) (IN)
newcomm	is the copy of **comm** (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_COMM_DUP is a collective function that is invoked by the group associated with **comm**. This routine duplicates the existing communicator **comm** with its associated key values.

For each key value the respective copy callback function determines the attribute value associated with this key in the new communicator. One action that a copy callback may take is to delete the attribute from the new communicator. Returns in **newcomm** a new communicator with the same group and any copied cached information, but a new context.

This call applies to both intra and inter communicators.

Notes

Use this operation to produce a duplicate communication space that has the same properties as the original communicator. This includes attributes and topologies.

This call is valid even if there are pending point to point communications involving the communicator **comm**.

Remember that MPI_COMM_DUP is collective on the input communicator, so it is erroneous for a thread to attempt to duplicate a communicator that is simultaneously involved in an MPI_COMM_DUP or any collective on some other thread.

Errors

| **Conflicting collective operations on communicator**

A copy_fn did not return MPI_SUCCESS

A delete_fn did not return MPI_SUCCESS

Invalid communicator

MPI not initialized

MPI already finalized

Related Information

MPI_KEYVAL_CREATE

MPI_COMM_FREE, MPI_Comm_free

Purpose

Marks a communicator for deallocation.

C Synopsis

```
#include <mpi.h>
int MPI_Comm_free(MPI_Comm *comm);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_COMM_FREE(INTEGER COMM,INTEGER IERROR)
```

Parameters

comm	is the communicator to be freed (handle) (INOUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This collective function marks either an intra or an inter communicator object for deallocation. MPI_COMM_FREE sets the handle to MPI_COMM_NULL. Actual deallocation of the communicator object occurs when active references to it have completed. The delete callback functions for all cached attributes are called in arbitrary order. The delete functions are called immediately and not deferred until deallocation.

Errors

A delete_fn did not return MPI_SUCCESS

Invalid communicator

MPI not initialized

MPI already finalized

Related Information

MPI_KEYVAL_CREATE

MPI_COMM_GROUP, MPI_Comm_group

Purpose

Returns the group handle associated with a communicator.

C Synopsis

```
#include <mpi.h>
int MPI_Comm_group(MPI_Comm comm,MPI_Group *group);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_COMM_GROUP(INTEGER COMM,INTEGER GROUP,INTEGER IERROR)
```

Parameters

comm	is the communicator (handle) (IN)
group	is the group corresponding to **comm** (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine returns the group handle associated with a communicator.

Notes

If **comm** is an intercommunicator, then **group** is set to the local group. To determine the remote group of an intercommunicator, use MPI_COMM_REMOTE_GROUP.

Errors

Invalid communicator

MPI not initialized

MPI already finalized

Related Information

MPI_COMM_REMOTE_GROUP

MPI_COMM_RANK, MPI_Comm_rank

Purpose

Returns the rank of the local task in the group associated with a communicator.

C Synopsis

```
#include <mpi.h>
int MPI_Comm_rank(MPI_Comm comm,int *rank);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_COMM_RANK(INTEGER COMM,INTEGER RANK,INTEGER IERROR)
```

Parameters

comm	is the communicator (handle) (IN)
rank	is an integer specifying the rank of the calling task in group of **comm** (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine returns the rank of the local task in the group associated with a communicator.

You can use this routine with MPI_COMM_SIZE to determine the amount of concurrency available for a specific job. MPI_COMM_RANK indicates the rank of the task that calls it in the range from 0...**size** – 1, where **size** is the return value of MPI_COMM_SIZE.

This routine is a shortcut to accessing the communicator's group with MPI_COMM_GROUP, computing the rank using MPI_GROUP_RANK and freeing the temporary group by using MPI_GROUP_FREE.

If **comm** is an intercommunicator, **rank** is the rank of the local task in the local group.

Errors

Invalid communicator

MPI not initialized

MPI already finalized

Related Information

MPI_GROUP_RANK

MPI_COMM_REMOTE_GROUP, MPI_Comm_remote_group

Purpose

Returns the handle of the remote group of an intercommunicator.

C Synopsis

```
#include <mpi.h>
int MPI_Comm_remote_group(MPI_Comm comm,MPI_group *group);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_COMM_REMOTE_GROUP(INTEGER COMM,MPI_GROUP GROUP,INTEGER IERROR)
```

Parameters

comm	is the intercommunicator (handle) (IN)
group	is the remote group corresponding to **comm**. (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine is a local operation that returns the handle of the remote group of an intercommunicator.

Notes

To determine the local group of an intercommunicator, use MPI_COMM_GROUP.

Errors

Invalid communicator

Invalid communicator type it must be intercommunicator

MPI not initialized

MPI already finalized

Related Information

MPI_COMM_GROUP

MPI_COMM_REMOTE_SIZE, MPI_Comm_remote_size

Purpose

Returns the size of the remote group of an intercommunicator.

C Synopsis

```
#include <mpi.h>
int MPI_Comm_remote_size(MPI_Comm comm,int *size);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_COMM_REMOTE_SIZE(INTEGER COMM,INTEGER SIZE,INTEGER IERROR)
```

Parameters

comm	is the intercommunicator (handle) (IN)
size	is an integer specifying the number of tasks in the remote group of **comm**. (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine is a local operation that returns the size of the remote group of an intercommunicator.

Notes

To determine the size of the local group of an intercommunicator, use MPI_COMM_SIZE.

Errors

Invalid communicator

Invalid communicator type it must be intercommunicator

MPI not initialized

MPI already finalized

Related Information

MPI_COMM_SIZE

MPI_COMM_SIZE, MPI_Comm_size

Purpose

Returns the size of the group associated with a communicator.

C Synopsis

```
#include <mpi.h>
int MPI_Comm_size(MPI_Comm comm,int *size);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_COMM_SIZE(INTEGER COMM,INTEGER SIZE,INTEGER IERROR)
```

Parameters

comm	is the communicator (handle) (IN)
size	is an integer specifying the number of tasks in the group of **comm** (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine returns the size of the group associated with a communicator. This routine is a shortcut to:

- accessing the communicator's group with MPI_COMM_GROUP,
- computing the size using MPI_GROUP_SIZE, and
- freeing the temporary group using MPI_GROUP_FREE.

If **comm** is an intercommunicator, **size** will be the size of the local group. To determine the size of the remote group of an intercommunicator, use MPI_COMM_REMOTE_SIZE.

You can use this routine with MPI_COMM_RANK to determine the amount of concurrency available for a specific library or program. MPI_COMM_RANK indicates the rank of the task that calls it in the range from 0...**size** – 1, where **size** is the return value of MPI_COMM_SIZE. The rank and size information can then be used to partition work across the available tasks.

Notes

This function indicates the number of tasks in a communicator. For MPI_COMM_WORLD, it indicates the total number of tasks available.

Errors

Invalid communicator

MPI not initialized

MPI already finalized

Related Information

MPI_GROUP_SIZE
MPI_COMM_GROUP
MPI_COMM_RANK
MPI_COMM_REMOTE_SIZE
MPI_GROUP_FREE

MPI_COMM_SPLIT, MPI_Comm_split

Purpose

Splits a communicator into multiple communicators based on **color** and **key**.

C Synopsis

```
#include <mpi.h>
int MPI_Comm_split(MPI_Comm comm,int color,int key,MPI_Comm *newcomm);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_COMM_SPLIT(INTEGER COMM,INTEGER COLOR,INTEGER KEY,
    INTEGER NEWCOMM,INTEGER IERROR)
```

Parameters

comm	is the communicator (handle) (IN)
color	is an integer specifying control of subset assignment (IN)
key	is an integer specifying control of rank assignment (IN)
newcomm	is the new communicator (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_COMM_SPLIT is a collective function that partitions the group associated with **comm** into disjoint subgroups, one for each value of **color**. Each subgroup contains all tasks of the same color. Within each subgroup, the tasks are ranked in the order defined by the value of the argument **key**. Ties are broken according to their rank in the old group. A new communicator is created for each subgroup and returned in **newcomm**. If a task supplies the color value MPI_UNDEFINED, **newcomm** returns MPI_COMM_NULL. Even though this is a collective call, each task is allowed to provide different values for **color** and **key**.

This call applies only to intracommunicators.

The value of **color** must be greater than or equal to zero.

Errors

| **Conflicting collective operations on communicator**

Invalid color	**color** < 0
Invalid communicator	
Invalid communicator type	it must be intracommunicator
MPI not initialized	
MPI already finalized	

Related Information

MPI_CART_SUB

MPI_COMM_TEST_INTER, MPI_Comm_test_inter

Purpose

Returns the type of a communicator (intra or inter).

C Synopsis

```
#include <mpi.h>
int MPI_Comm_test_inter(MPI_Comm comm,int *flag);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_COMM_TEST_INTER(INTEGER COMM,LOGICAL FLAG,INTEGER IERROR)
```

Parameters

comm	is the communicator (handle) (INOUT)
flag	communicator type (logical)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine is used to determine if a communicator is an inter or an intracommunicator.

If **comm** is an intercommunicator, the call returns **true**. If **comm** is an intracommunicator, the call returns **false**.

Notes

An intercommunicator can be used as an argument to some of the communicator access routines. However, intercommunicators cannot be used as input to some of the constructor routines for intracommunicators, such as MPI_COMM_CREATE.

Errors

Invalid communicator

MPI not initialized

MPI already finalized

MPI_DIMS_CREATE, MPI_Dims_create

Purpose

Defines a cartesian grid to balance tasks.

C Synopsis

```
#include <mpi.h>
MPI_Dims_create(int nnodes,int ndims,int *dims);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_DIMS_CREATE(INTEGER NNODES,INTEGER NDIMS,INTEGER DIMS(*),
      INTEGER IERROR)
```

Parameters

nnodes	is an integer specifying the number of nodes in a grid (IN)
ndims	is an integer specifying the number of cartesian dimensions (IN)
dims	is an integer array of size *ndims* that specifies the number of nodes in each dimension. (INOUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine creates a cartesian grid with a given number of dimensions and a given number of nodes. The dimensions are constrained to be as close to each other as possible.

If **dims[i]** is a positive number when MPI_DIMS_CREATE is called, the routine will not modify the number of nodes in dimension **i**. Only those entries where **dims[i]**=**0** are modified by the call.

Notes

MPI_DIMS_CREATE chooses dimensions so that the resulting grid is as close as possible to being an *ndims*–dimensional **cube**.

Errors

MPI not initialized

MPI already finalized

Invalid *ndims*	*ndims* < 0
Invalid nnodes	**nnodes**<0
Invalid dimension	**dims[i]** < 0 or **nnodes** is not a multiple of the non-zero entries of **dims**

Related Information

MPI_CART_CREATE

MPI_ERRHANDLER_CREATE, MPI_Errhandler_create

Purpose

Registers a user-defined error handler.

C Synopsis

```
#include <mpi.h>
int MPI_Errhandler_create(MPI_Handler_function *function,
    MPI_Errhandler *errhandler);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_ERRHANDLER_CREATE(EXTERNAL FUNCTION,INTEGER ERRHANDLER,
      INTEGER IERROR)
```

Parameters

function	is a user defined error handling procedure (IN)
errhandler	is an MPI error handler (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_ERRHANDLER_CREATE registers the user routine **function** for use as an MPI error handler.

You can associate an error handler with a communicator. MPI will use the specified error handling routine for any exception that takes place during a call on this communicator. Different tasks can attach different error handlers to the same communicator. MPI calls not related to a specific communicator are considered as attached to the communicator MPI_COMM_WORLD.

Notes

The MPI standard specifies the following error handler prototype. A correct user error handler would be coded as:

```
void my_handler(MPI_Comm *comm, int *errcode, ...){}
```

The Parallel Environment for AIX implementation of MPI passes additional arguments to an error handler. The MPI standard allows this and urges an MPI implementation that does so to document the additional arguments. These additional arguments will be ignored by fully portable user error handlers. Anyone who wants to use the extra errhandler arguments can do so by using the C varargs (or stdargs) facility, but will be writing code that does not port cleanly to other MPI implementations, which happen to have different additional arguments.

The effective prototype for an error handler in IBM's implementation is:

```
typedef void (MPI_Handler_function)
  (MPI_Comm *comm, int *code, char *routine_name, int *flag, int *badval)
```

The additional arguments are:

	routine_name	the name of the MPI routine in which the error occurred
	flag	TRUE if *badval* is meaningful, FALSE if not
	badval	the non-valid integer value that triggered the error

| The interpretation of *badval* is context-dependent, so *badval* is not likely to be
| useful to a user error handler function that cannot identify this context. The
| *routine_name* string is more likely to be useful.

Errors

NULL function

MPI not initialized

MPI already finalized

Related Information

MPI_ERRHANDLER_SET
MPI_ERRHANDLER_GET
MPI_ERRHANDLER_FREE

MPI_ERRHANDLER_FREE, MPI_Errhandler_free

Purpose

Marks an error handler for deallocation.

C Synopsis

```
#include <mpi.h>
int MPI_Errhandler_free(MPI_Errhandler *errhandler);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_ERRHANDLER_FREE(INTEGER ERRHANDLER,INTEGER IERROR)
```

Parameters

errhandler	is an MPI error handler (handle) (INOUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine marks error handler **errhandler** for deallocation and sets **errhandler** to MPI_ERRHANDLER_NULL. Actual deallocation occurs when all communicators associated with the error handler have been deallocated.

Errors

Invalid error handler

MPI not initialized

MPI already finalized

Related Information

MPI_ERRHANDLER_CREATE

MPI_ERRHANDLER_GET, MPI_Errhandler_get

Purpose

Gets an error handler associated with a communicator.

C Synopsis

```
#include <mpi.h>
int MPI_Errhandler_get(MPI_Comm comm,MPI_Errhandler *errhandler);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_ERRHANDLER_GET(INTEGER COMM,INTEGER ERRHANDLER,INTEGER IERROR)
```

Parameters

comm	is a communicator (handle) (IN)
errhandler	is the MPI error handler currently associated with **comm** (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine returns the error handler **errhandler** currently associated with communicator **comm**.

Errors

Invalid communicator

MPI not initialized

MPI already finalized

Related Information

MPI_ERRHANDLER_SET
MPI_ERRHANDLER_CREATE

MPI_ERRHANDLER_SET, MPI_Errhandler_set

Purpose

Associates a new error handler with a communicator.

C Synopsis

```
#include <mpi.h>
int MPI_Errhandler_set(MPI_Comm comm,MPI_Errhandler errhandler);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_ERRHANDLER_SET(INTEGER COMM, INTEGER ERRHANDLER, INTEGER IERROR)
```

Parameters

comm	is a communicator (handle) (IN)
errhandler	is a new MPI error handler for **comm** (handle) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine associates error handler **errhandler** with communicator **comm**. The association is local.

MPI will use the specified error handling routine for any exception that takes place during a call on this communicator. Different tasks can attach different error handlers to the same communicator. MPI calls not related to a specific communicator are considered as attached to the communicator MPI_COMM_WORLD.

Notes

An error handler that does not end in the MPI job being terminated, creates undefined risks. Some errors are harmless while others are catastrophic. For example, an error detected by one member of a collective operation can result in other members waiting indefinitely for an operation which will never occur.

It is also important to note that the MPI standard does not specify the state the MPI library should be in after an error occurs. MPI does not provide a way for users to determine how much, if any, damage has been done to the MPI state by a particular error.

The default error handler is MPI_ERRORS_ARE_FATAL, which behaves as if it contains a call to MPI_ABORT. MPI_ERRHANDLER_SET allows users to replace MPI_ERRORS_ARE_FATAL with an alternate error handler. The MPI standard provides MPI_ERRORS_RETURN, and IBM adds the non-standard MPE_ERRORS_WARN. These are pre-defined handlers that cause the error code to be returned and MPI to continue to run. Error handlers that are written by MPI users may call MPI_ABORT. If they do not abort, they too will cause MPI to deliver an error return code to the caller and continue to run.

Error handlers that let MPI return should only be used if every MPI call checks its return code. Continuing to use MPI after an error involves undefined risks. You may do cleanup after an MPI error is detected, as long as it doesn't use MPI calls. This should normally be followed by a call to MPI_ABORT.

| The error **Invalid error handler** will be raised if **errhandler** is a file error handler
| (created with the routine MPI_FILE_CREATE_ERRHANDLER). Predefined error
| handlers, MPI_ERRORS_ARE_FATAL and MPI_ERRORS_RETURN, can be
| associated with both communicators and file handles.

Errors

Invalid Communicator

Invalid error handler

MPI not initialized

MPI already finalized

Related Information

MPI_ERRHANDLER_GET
MPI_ERRHANDLER_CREATE

MPI_ERROR_CLASS, MPI_Error_class

Purpose

Returns the error class for the corresponding error code.

C Synopsis

```
#include <mpi.h>
int MPI_Error_class(int errorcode,int *errorclass);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_ERROR_CLASS(INTEGER ERRORCODE,INTEGER ERRORCLASS,INTEGER IERROR)
```

Parameters

errorcode	is the error code returned by an MPI routine (IN)
errorclass	is the error class for the **errorcode** (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine returns the error class corresponding to an error code.

| Table 2 lists the valid error classes for threaded and non-threaded libraries.

| *Table 2 (Page 1 of 2). MPI Error Classes: Threaded and Non-Threaded Libraries*

Error Classes	**Description**
MPI_SUCCESS	No error
MPI_ERR_BUFFER	Non-valid buffer pointer
MPI_ERR_COUNT	Non-valid count argument
MPI_ERR_TYPE	Non-valid datatype argument
MPI_ERR_TAG	Non-valid tag argument
MPI_ERR_COMM	Non-valid communicator
MPI_ERR_RANK	Non-valid rank
MPI_ERR_REQUEST	Non-valid request (handle)
MPI_ERR_ROOT	Non-valid root
MPI_ERR_GROUP	Non-valid group
MPI_ERR_OP	Non-valid operation
MPI_ERR_TOPOLOGY	Non-valid topology
MPI_ERR_DIMS	Non-valid dimension argument
MPI_ERR_ARG	Non-valid argument
MPI_ERR_IN_STATUS	Error code is in status
| MPI_ERR_PENDING	Pending request
MPI_ERR_TRUNCATE	Message truncated on receive

Table 2 (Page 2 of 2). MPI Error Classes: Threaded and Non-Threaded Libraries

Error Classes	Description
MPI_ERR_INTERN	Internal MPI error
MPI_ERR_OTHER	Known error not provided
MPI_ERR_UNKNOWN	Unknown error
MPI_ERR_LASTCODE	Last standard error code

Table 3 lists the valid error classes for threaded libraries only.

Table 3. MPI Error Classes: Threaded Libraries Only

Error Classes	Description
MPI_ERR_FILE	Non-valid file handle
MPI_ERR_NOT_SAME	Collective argument is not identical on all tasks
MPI_ERR_AMODE	Error related to the **amode** passed to MPI_FILE_OPEN
MPI_ERR_UNSUPPORTED_DATAREP	Unsupported **datarep** passed to MPI_FILE_SET_VIEW
MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on a file that supports sequential access only
MPI_ERR_NO_SUCH_FILE	File does not exist
MPI_ERR_FILE_EXISTS	File exists
MPI_ERR_BAD_FILE	Non-valid file name (the path name is too long, for example)
MPI_ERR_ACCESS	Permission denied
MPI_ERR_NO_SPACE	Not enough space
MPI_ERR_QUOTA	Quota exceeded
MPI_ERR_READ_ONLY	Read-only file or file system
MPI_ERR_FILE_IN_USE	File operation could not be completed because the file is currently opened by some task
MPI_ERR_DUP_DATAREP	Conversion functions could not be registered because a previously-defined data representation was passed to MPI_REGISTER_DATAREP
MPI_ERR_CONVERSION	An error occurred in a user-supplied data conversion function
MPI_ERR_IO	Other I/O error

Notes

For this implementation of MPI, refer to the *IBM Parallel Environment for AIX: Messages*, which provides a listing of all the error messages issued as well as the error class to which the message belongs. Be aware that the MPI standard is not explicit enough about error classes to guarantee that every implementation of MPI will use the same error class for every detectable user error.

Errors

MPI not initialized

MPI already finalized

Related Information

MPI_ERROR_STRING

MPI_ERROR_STRING, MPI_Error_string

Purpose

Returns the error string for a given error code.

C Synopsis

```
#include <mpi.h>
int MPI_Error_string(int errorcode,char *string,
      int *resultlen);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_ERROR_STRING(INTEGER ERRORCODE,CHARCTER STRING(*),
     INTEGER RESULTLEN,INTEGER IERROR)
```

Parameters

errorcode	is the error code returned by an MPI routine (IN)
string	is the error message for the **errorcode** (OUT)
resultlen	is the character length of **string** (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine returns the error string for a given error code. The returned **string** is null terminated with the terminating byte not counted in **resultlen**.

Storage for **string** must be at least MPI_MAX_ERROR_STRING characters long. The number of characters actually written is returned in **resultlen**.

Errors

Invalid error code	**errorcode** is not defined
MPI not initialized	
MPI already finalized	

Related Information

MPI_ERROR_CLASS

MPI_FILE_CLOSE, MPI_File_close

Purpose

Closes the file referred to by its file handle **fh**. It may also delete the file if the appropriate mode was set when the file was opened.

C Synopsis

```
#include <mpi.h>
int MPI_File_close (MPI_File *fh);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_FILE_CLOSE(INTEGER FH,INTEGER IERROR)
```

Parameters

fh	is the file handle of the file to be closed (handle) (INOUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_FILE_CLOSE closes the file referred to by **fh** and deallocates associated internal data structures. This is a collective operation. The file is also deleted if MPI_MODE_DELETE_ON_CLOSE was set when the file was opened. In this situation, if other tasks have already opened the file and are still accessing it concurrently, these accesses will proceed normally, as if the file had not been deleted, until the tasks close the file. However, new open operations on the file will fail. If I/O operations are pending on **fh**, an error is returned to all the participating tasks, the file is neither closed nor deleted, and **fh** remains a valid file handle.

Notes

You are responsible for making sure all outstanding nonblocking requests and split collective operations associated with **fh** made by a task have completed before that task calls MPI_FILE_CLOSE.

If you call MPI_FINALIZE before all files are closed, an error will be raised on MPI_COMM_WORLD.

MPI_FILE_CLOSE deallocates the file handle object and sets **fh** to MPI_FILE_NULL.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Returning Errors (MPI Error Class):

Invalid file handle (MPI_ERR_FILE)
: **fh** is not a valid file handle

Pending I/O operations (MPI_ERR_OTHER)
: There are pending I/O operations

Internal close failed (MPI_ERR_IO)
: An internal **close** operation on the file failed

Returning Errors When a File Is To Be Deleted (MPI Error Class):

Permission denied (MPI_ERR_ACCESS)
: Write access to the directory containing the file is denied

File does not exist (MPI_ERR_NO_SUCH_FILE)
: The file that is to be deleted does not exist

Read-only file system (MPI_ERR_READ_ONLY)
: The directory containing the file resides on a read-only file system

Internal unlink failed (MPI_ERR_IO)
: An internal **unlink** operation on the file failed

Related Information

MPI_FILE_OPEN
MPI_FILE_DELETE
MPI_FINALIZE

MPI_FILE_CREATE_ERRHANDLER, MPI_File_create_errhandler

Purpose

Registers a user-defined error handler that you can associate with an open file.

C Synopsis

```
#include <mpi.h>
int MPI_File_create_errhandler (MPI_File_errhandler_fn *function,
    MPI_Errhandler *errhandler);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_FILE_CREATE_ERRHANDLER(EXTERNAL FUNCTION,INTEGER ERRHANDLER,
    INTEGER IERROR)
```

Parameters

function	is a user defined file error handling procedure (IN)
errhandler	is an MPI error handler (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_FILE_CREATE_ERRHANDLER registers the user routine **function** for use as an MPI error handler that can be associated with a file handle. Once associated with a file handle, MPI uses the specified error handling routine for any exception that takes place during a call on this file handle.

Notes

Different tasks can associate different error handlers with the same file. MPI_ERRHANDLER_FREE is used to free any error handler.

The MPI standard specifies the following error handler prototype:

```
typedef void (MPI_File_errhandler_fn) (MPI_File *, int *, ...);
```

A correct user error handler would be coded as:

```
void my_handler(MPI_File *fh, int *errcode,...){}
```

The Parallel Environment for AIX implementation of MPI passes additional arguments to an error handler. The MPI standard allows this and urges an MPI implementation that does so to document the additional arguments. These additional arguments will be ignored by fully portable user error handlers. Anyone who wants to use the extra errhandler arguments can do so by using the C varargs (or stdargs) facility, but will be writing code that does not port cleanly to other MPI implementations, which happen to have different additional arguments.

The effective prototype for an error handler in IBM's implementation is:

```
 typedef void (MPI_File_errhandler_fn)
   (MPI_File *fh, int *code, char *routine_name, int *flag, int *badval)
```

The additional arguments are:

routine_name	the name of the MPI routine in which the error occurred
flag	TRUE if *badval* is meaningful, FALSE if not
badval	the non-valid integer value that triggered the error

The interpretation of *badval* is context-dependent, so *badval* is not likely to be useful to a user error handler function that cannot identify this context. The *routine_name* string is more likely to be useful.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Null function not allowed **function** cannot be NULL.

Related Information

MPI_FILE_SET_ERRHANDLER
MPI_FILE_GET_ERRHANDLER
MPI_ERRHANDLER_FREE

MPI_FILE_DELETE, MPI_File_delete

Purpose

Deletes the file referred to by **filename** after pending operations on the file complete. New operations cannot be initiated on the file.

C Synopsis

```
#include <mpi.h>
int MPI_File_delete (char *filename,MPI_Info info);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_FILE_DELETE(CHARACTER*(*) FILENAME,INTEGER INFO,
    INTEGER IERROR)
```

Parameters

filename	is the name of the file to be deleted (string) (IN)
info	is an **info** object specifying file hints (handle) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine deletes the file referred to by **filename**. If other tasks have already opened the file and are still accessing it concurrently, these accesses will proceed normally, as if the file had not been deleted, until the tasks close the file. However, new open operations on the file will fail. There are no hints defined for MPI_FILE_DELETE.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Returning Errors (MPI Error Class):

Pathname too long (MPI_ERR_BAD_FILE)
A **filename** must contain less than 1024 characters.

Invalid file system type (MPI_ERR_OTHER)
filename refers to a file belonging to a file system of an unsupported type.

Invalid info (MPI_ERR_INFO)
info is not a valid **info** object.

Permission denied (MPI_ERR_ACCESS)
Write access to the directory containing the file is denied.

File or directory does not exist (MPI_ERR_NO_SUCH_FILE)
The file that is to be deleted does not exist, or a directory in the path does not exist.

Read-only file system (MPI_ERR_READ_ONLY)
The directory containing the file resides on a read-only file system.

Internal unlink failed (MPI_ERR_IO)
An internal **unlink** operation on the file failed.

Related Information

MPI_FILE_CLOSE

MPI_FILE_GET_AMODE, MPI_File_get_amode

Purpose

Retrieves the access mode specified when the file was opened.

C Synopsis

```
#include <mpi.h>
int MPI_File_get_amode (MPI_File fh,int *amode);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_FILE_GET_AMODE(INTEGER FH,INTEGER AMODE,INTEGER IERROR)
```

Parameters

fh	is the file handle (handle) (IN)
amode	is the file access mode used to open the file (integer) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_FILE_GET_AMODE allows you to retrieve the access mode specified when the file referred to by **fh** was opened.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Returning Errors (MPI Error Class):

Invalid file handle (MPI_ERR_FILE)
fh is not a valid file handle.

Related Information

MPI_FILE_OPEN

MPI_FILE_GET_ATOMICITY, MPI_File_get_atomicity

Purpose

Retrieves the current atomicity mode in which the file is accessed.

C Synopsis

```
#include <mpi.h>
int MPI_File_get_atomicity (MPI_File fh,int *flag);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_FILE_GET_ATOMICITY (INTEGER FH,LOGICAL FLAG,INTEGER IERROR)
```

Parameters

fh	is the file handle (handle) (IN)
flag	TRUE if atomic mode, FALSE if non-atomic mode (boolean) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_FILE_GET_ATOMICITY returns in **flag** 1 if the atomic mode is enabled for the file referred to by **fh**, otherwise **flag** returns 0.

Notes

The atomic mode is set to FALSE by default when the file is first opened. In MPI-2, MPI_FILE_SET_ATOMICITY is defined as the way to set atomicity. However, it is not provided in this release.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Returning Errors (MPI Error Class):

Invalid file handle (MPI_ERR_FILE)
fh is not a valid file handle.

Related Information

MPI_FILE_OPEN

MPI_FILE_GET_ERRHANDLER, MPI_File_get_errhandler

Purpose

Retrieves the error handler currently associated with a file handle.

C Synopsis

```
#include <mpi.h>
int MPI_File_get_errhandler (MPI_File file,MPI_Errhandler *errhandler);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_FILE_GET_ERRHANDLER (INTEGER FILE,INTEGER ERRHANDLER,
    INTEGER IERROR)
```

Parameters

fh	is a file handle or MPI_FILE_NULL (handle)(IN)
errhandler	is the error handler currently associated with **fh** or the current default file error handler (handle)(OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

If **fh** is MPI_FILE_NULL, then MPI_FILE_GET_ERRHANDLER returns in **errhandler** the default file error handler currently assigned to the calling task. If **fh** is a valid file handle, then MPI_FILE_GET_ERRHANDLER returns in **errhandler**, the error handler currently associated with the file handle **fh**. Error handlers may be different at each task.

Notes

At MPI_INIT time, the default file error handler is MPI_ERRORS_RETURN. You can alter the default by calling the routine MPI_FILE_SET_ERRHANDLER and passing MPI_FILE_NULL as the file handle parameter. Any program that uses MPI_ERRORS_RETURN should check function return codes.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Invalid file handle	**fh** must be a valid file handle or MPI_FILE_NULL.

Related Information

MPI_FILE_CREATE_ERRHANDLER
MPI_FILE_SET_ERRHANDLER
MPI_ERRHANDLER_FREE

MPI_FILE_GET_GROUP, MPI_File_get_group

Purpose

Retrieves the group of tasks that opened the file.

C Synopsis

```
#include <mpi.h>
int MPI_File_get_group (MPI_File fh,MPI_Group *group);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_FILE GET_GROUP (INTEGER FH,INTEGER GROUP,INTEGER IERROR)
```

Parameters

fh	is the file handle (handle) (IN)
group	is the group which opened the file handle (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_FILE_GET_GROUP lets you retrieve in **group** the group of tasks that opened the file referred to by **fh**. You are responsible for freeing **group** via MPI_GROUP_FREE.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Returning Errors (MPI Error Class):

Invalid file handle (MPI_ERR_FILE)
fh is not a valid file handle.

Related Information

MPI_FILE_OPEN

MPI_FILE_GET_INFO, MPI_File_get_info

Purpose

Returns a new info object identifying the hints associated with **fh**.

C Synopsis

```
#include <mpi.h>
int MPI_File_get_info (MPI_File fh,MPI_Info *info_used);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_FILE_GET_INFO (INTEGER FH,INTEGER INFO_USED,
    INTEGER IERROR)
```

Parameters

fh	is the file handle (handle) (IN)
info_used	is the new **info** object (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

Because no file hints are defined in this release, MPI_FILE_GET_INFO simply creates a new empty **info** object and returns its handle in **info_used** after checking for the validity of the file handle **fh**. You are responsible for freeing **info_used** via MPI_INFO_FREE.

Notes

File hints can be specified by the user through the info parameter of routines: MPI_FILE_SET_INFO, MPI_FILE_OPEN, MPI_FILE_SET_VIEW. MPI can also assign default values to file hints it supports when these hints are not specified by the user.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Returning Errors (MPI Error Class):

Invalid file handle (MPI_ERR_FILE)
fh is not a valid file handle.

| ## Related Information

| MPI_FILE_SET_INFO
| MPI_FILE_OPEN
| MPI_FILE_SET_VIEW
| MPI_INFO_FREE

MPI_FILE_GET_SIZE, MPI_File_get_size

Purpose

Retrieves the current file size.

C Synopsis

```
#include <mpi.h>
int MPI_File_get_size (MPI_File fh,MPI_Offset size);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_FILE_GET_SIZE (INTEGER FH,INTEGER(KIND=MPI_OFFSET_KIND) SIZE,
    INTEGER IERROR)
```

Parameters

fh	is the file handle (handle) (IN)
size	is the size of the file in bytes (long long) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_FILE_GET_SIZE returns in **size** the current length in bytes of the open file referred to by **fh**.

Notes

You can alter the size of the file by calling the routine MPI_FILE_SET_SIZE. The size of the file will also be altered when a write operation to the file results in adding data beyond the current end of the file.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Returning Errors (MPI Error Class):

Invalid file handle (MPI_ERR_FILE)
fh is not a valid file handle.

Internal fstat failed (MPI_ERR_IO)
An internal **fstat** operation on the file failed.

| ## Related Information

| MPI_FILE_SET_SIZE
| MPI_FILE_WRITE_AT
| MPI_FILE_WRITE_AT_ALL
| MPI_FILE_IWRITE_AT

MPI_FILE_GET_VIEW, MPI_File_get_view

Purpose

Retrieves the current file view.

C Synopsis

```
#include <mpi.h>
int MPI_File_get_view (MPI_File fh,MPI_Offset *disp,
    MPI_Datatype *etype,MPI_Datatype *filetype,char *datarep);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_FILE_GET_VIEW (INTEGER FH,INTEGER(KIND=MPI_OFFSET_KIND) DISP,
    INTEGER ETYPE,INTEGER FILETYPE,INTEGER DATAREP,INTEGER IERROR)
```

Parameters

fh	is the file handle (handle) (IN)
disp	is the displacement (long long) (OUT)
etype	is the elementary datatype (handle) (OUT).
filetype	is the file type (handle) (OUT).
datarep	is the data representation (string) (OUT).
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_FILE_GET_VIEW retrieves the current view associated with the open file referred to by **fh**. The current view displacement is returned in **disp**. A reference to the current elementary datatype is returned in **etype** and a reference to the current file type is returned in **filetype**. The current data representation is returned in **datarep**. If **etype** and **filetype** are named types, they cannot be freed. If either one is a user-defined types, it should be freed. Use MPI_TYPE_GET_ENVELOPE to identify which types should be freed via MPI_TYPE_FREE. Freeing the MPI_Datatype reference returned by MPI_FILE_GET_VIEW invalidates only this reference.

Notes

- The default view is associated with the file when the file is opened. This view corresponds to a byte stream starting at file offset 0 (zero) and using the native data representation, which is:

  disp equals 0(zero)
  etype equals MPI_BYTE
  filetype equals MPI_BYTE
  datarep equals “native”

  To alter the view of the file, you can call the routine MPI_FILE_SET_VIEW.

- An MPI type constructor, such as MPI_TYPE_CONTIGUOUS, creates a datatype object within MPI and gives a handle for that object to the caller. This handle represents one reference to the object. In this implementation of MPI, the MPI datatypes obtained with calls to MPI_TYPE_GET_VIEW are new handles for the existing datatype objects. The number of handles (references) given to the user is tracked by a reference counter in the object. MPI cannot discard a datatype object unless MPI_TYPE_FREE has been called on every handle the user has obtained.

  The use of reference-counted objects is encouraged, but not mandated, by the MPI standard. Another MPI implementation may create new objects instead. The user should be aware of a side effect of the reference count approach. Suppose mytype was created by a call to MPI_TYPE_VECTOR and used so that a later call to MPI_TYPE_GET_VIEW returns its handle in hertype. Because both handles identify the same datatype object, attribute changes made with either handle are changes in the single object. That object will exist at least until MPI_TYPE_FREE has been called on both mytype and hertype. Freeing either handle alone will leave the object intact and the other handle will remain valid.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Returning Errors (MPI Error Class):

Invalid file handle (MPI_ERR_FILE)
fh is not a valid file handle.

Related Information

MPI_FILE_OPEN
MPI_FILE_SET_VIEW
MPI_TYPE_FREE

MPI_FILE_IREAD_AT, MPI_File_iread_at

Purpose

A nonblocking version of MPI_FILE_READ_AT. The call returns immediately with a request handle that you can use to check for the completion of the read operation.

C Synopsis

```
#include <mpi.h>
int MPI_File_iread_at (MPI_File fh,MPI_Offset offset,void *buf,
    int count,MPI_Datatype datatype,MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_FILE_IREAD_AT (INTEGER FH,INTEGER (KIND=MPI_OFFSET_KIND) OFFSET,
    CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER REQUEST,
    INTEGER IERROR)
```

Parameters

fh	is the file handle (handle) (IN).
offset	is the file offset (long long) (IN).
buf	is the initial address of buffer (choice) (OUT).
count	is the number of elements in the buffer (integer) (IN).
datatype	is the datatype of each buffer element (handle) (IN).
request	is the request object (handle) (OUT).
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine, MPI_FILE_IREAD_AT, is the nonblocking version of MPI_FILE_READ_AT and it performs the same function as MPI_FILE_READ_AT except it immediately returns in **request** a handle. This request handle can be used to either test or wait for the completion of the read operation or it can be used to cancel the read operation. The memory buffer **buf** cannot be accessed until the request has completed via a completion routine call. Completion of the request guarantees that the read operation is complete.

When MPI_FILE_IREAD_AT completes, the actual number of bytes read is stored in the completion routine's **status** argument. If an error occurs during the read operation, the error is returned by the completion routine through its return value or in the appropriate index of the **array_of_statuses** argument.

If the completion routine is associated with multiple requests, it returns when requests complete successfully. Or, if one of the requests fails, the errorhandler associated with that request is triggered. If that is an "error return" errorhandler, each element of the **array_of_statuses** argument is updated to contain MPI_ERR_PENDING for each request that did not yet complete. The first error dictates the outcome of the entire completion routine whether the error is on a file request or a communication request. The order in which requests are processed is not defined.

Notes

A valid call to MPI_CANCEL on the request will return MPI_SUCCESS. The eventual call to MPI_TEST_CANCELLED on the status will show that the cancel was unsuccessful.

Note that when you specify a value for the **offset** argument, constants of the appropriate type should be used. In Fortran, constants of type INTEGER(KIND=8) should be used, for example, 45_8.

Passing MPI_STATUS_IGNORE for the status argument or MPI_STATUSES_IGNORE for the **array_of_statuses** argument in the completion routine call is not supported in this release.

If an error occurs during the read operation, the number of bytes contained in the status argument of the completion routine is meaningless.

For additional information, see MPI_FILE_READ_AT.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Returning Errors (MPI Error Class):

Permission denied (MPI_ERR_ACCESS)
The file was opened in write-only mode.

Invalid file handle (MPI_ERR_FILE)
fh is not a valid file handle.

Invalid count (MPI_ERR_COUNT)
count is an invalid count.

MPI_DATATYPE_NULL not valid (MPI_ERR_TYPE)
datatype has already been freed.

Undefined datatype (MPI_ERR_TYPE)
datatype is not a defined datatype.

Invalid datatype (MPI_ERR_TYPE)
datatype can be neither MPI_LB nor MPI_UB.

Uncommitted datatype (MPI_ERR_TYPE)
datatype must be committed.

Unsupported operation on sequential access file (MPI_ERR_UNSUPPORTED_OPERATION)
MPI_MODE_SEQUENTIAL was set when the file was opened.

Invalid offset (MPI_ERR_ARG)
offset is an invalid offset.

| *Error Returned By Completion Routine (MPI Error Class):*

| **Internal read failed (MPI_ERR_IO)** An internal **read** operation failed.

| **Internal lseek failed (MPI_ERR_IO)** An internal **lseek** operation failed.

| Related Information

| MPI_FILE_READ_AT
| MPI_WAIT
| MPI_TEST
| MPI_CANCEL

MPI_FILE_IWRITE_AT, MPI_File_iwrite_at

Purpose

A nonblocking version of MPI_FILE_WRITE_AT. The call returns immediately with a request handle that you can use to check for the completion of the write operation.

C Synopsis

```
#include <mpi.h>
int MPI_File_iwrite_at (MPI_File fh,MPI_Offset offset,void *buf,
    int count,MPI_Datatype datatype,MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_FILE_IWRITE_AT(INTEGER FH,INTEGER(KIND=MPI_OFFSET_KIND) OFFSET,
   CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER REQUEST,
   INTEGER IERROR)
```

Parameters

fh	is the file handle (handle) (INOUT).
offset	is the file offset (long long) (IN).
buf	is the initial address of buffer (choice) (IN).
count	is the number of elements in buffer (integer) (IN).
datatype	is the datatype of elements in **count** (handle) (IN).
request	is the request object (handle) (OUT).
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine, MPI_FILE_IWRITE_AT, is the nonblocking version of MPI_FILE_WRITE_AT and it performs the same function as MPI_FILE_WRITE_AT except it immediately returns in **request** a handle. This request handle can be used to either test or wait for the completion of the write operation or it can be used to cancel the write operation. The memory buffer **buf** cannot be modified until the request has completed via a completion routine call. For example, MPI_WAIT, MPI_TEST, or one of the other MPI wait or test functions. Completion of the request does not guarantee that the data has been written to the storage device(s). In particular, written data may still be present in system buffers. However, it guarantees that the memory buffer can be safely reused.

When MPI_FILE_IWRITE_AT completes, the actual number of bytes written is stored in the completion routine's **status** argument. If an error occurs during the write operation, then the error is returned by the completion routine through its return code or in the appropriate index of the **array_of_statuses** argument.

If the completion routine is associated with multiple requests, it returns when all requests complete successfully. Or, if one of the requests fails, the errorhandler associated with that request is triggered. If that is an "error return" errorhandler, each element of the **array_of_statuses** argument is updated to contain MPI_ERR_PENDING for each request that did not yet complete. The first error dictates the outcome of the entire completion routine whether the error is on a file request or a communication request. The order in which requests are processed is not defined.

Notes

A valid call to MPI_CANCEL on the request will return MPI_SUCCESS. The eventual call to MPI_TEST_CANCELLED on the status will show that the cancel was unsuccessful.

Note that when you specify a value for the **offset** argument, constants of the appropriate type should be used. In Fortran, constants of type INTEGER(KIND=8) should be used, for example, 45_8.

Passing MPI_STATUSES_IGNORE for the **status** argument or MPI_STATUSES_IGNORE for the **array_of_statuses** argument in the completion routine call is not supported in this release.

If an error occurs during the write operation, the number of bytes contained in the status argument of the completion routine is meaningless.

For more information, see MPI_FILE_WRITE_AT.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Returning Errors (MPI Error Class):

Permission denied (MPI_ERR_ACCESS)
The file was opened in read-only mode.

Invalid file handle (MPI_ERR_FILE)
fh is not a valid file handle.

Invalid count (MPI_ERR_COUNT)
count is an invalid count.

MPI_DATATYPE_NULL not valid (MPI_ERR_TYPE)
datatype has already been freed.

Undefined datatype (MPI_ERR_TYPE)
datatype is not a defined datatype.

Invalid datatype (MPI_ERR_TYPE)
datatype can be neither MPI_LB nor MPI_UB.

Uncommitted datatype (MPI_ERR_TYPE)
datatype must be committed.

| **Unsupported operation on sequential access file (MPI_ERR_UNSUPPORTED_OPERATION)**
| MPI_MODE_SEQUENTIAL was set when the file was opened.

| **Invalid offset (MPI_ERR_ARG)**
| **offset** is an invalid offset.

| *Errors Returned By Completion Routine (MPI Error Class):*

| **Not enough space in file system (MPI_ERR_NO_SPACE)** The file system on which the file resides is full.

| **File too big (MPI_ERR_OTHER)** The file has reached the maximum size allowed.

| **Internal write failed (MPI_ERR_IO)** An internal **write** operation failed.

| **Internal lseek failed (MPI_ERR_IO)** An internal **lseek** operation failed.

| Related Information

| MPI_FILE_WRITE_AT
| MPI_FILE_WAIT
| MPI_FILE_TEST
| MPI_FILE_CANCEL

MPI_FILE_OPEN, MPI_File_open

Purpose

Opens the file called *filename*.

C Synopsis

```
#include <mpi.h>
int MPI_File_open (MPI_Comm comm,char *filename,int amode,MPI_info,
    MPI_File *fh);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_FILE_OPEN(INTEGER COMM,CHARACTER FILENAME(*),INTEGER AMODE,
    INTEGER INFO,INTEGER FH,INTEGER IERROR)
```

Parameters

comm	is the communicator (handle) (IN)
filename	is the name of the file to open (string) (IN)
amode	is the file access mode (integer) (IN)
info	is the **info** object (handle) (IN)
fh	is the new file handle (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_FILE_OPEN opens the file referred to by **filename**, sets the default view on the file, and sets the access mode **amode**. MPI_FILE_OPEN returns a file handle **fh** used for all subsequent operations on the file. The file handle **fh** remains valid until the file is closed (MPI_FILE_CLOSE). The default view is similar to a linear byte stream in the native representation starting at file offset 0. You can call MPI_FILE_SET_VIEW to set a different view of the file.

MPI_FILE_OPEN is a collective operation. **comm** must be a valid intracommunicator. Values specified for **amode** by all participating tasks must be identical. The program is erroneous when participating tasks do not refer to the same file through their own instances of **filename**.

No hints are defined in this release; therefore, **info** is presumed to be empty.

Notes

This implementation is targeted to the IBM Generalized Parallel File System (GPFS) for production use. It requires that a single GPFS file system be available across all tasks of the MPI job. It can also be used for development purposes on any other file system that supports the POSIX interface (AFS, DFS, JFS, or NFS), as long as the application runs on only one node or workstation.

For AFS, DFS, and NFS, MPI-IO uses file locking for all accesses by default. If other tasks on the same node share the file and also use file locking, file *consistency is preserved. If the MPI_FILE_OPEN is done with mode MPI_MODE_UNIQUE_OPEN, file locking is not done.*

If you call MPI_FINALIZE before all files are closed, an error will be raised on MPI_COMM_WORLD.

The following access modes (specified in **amode**), are supported:

MPI_MODE_RDONLY - read only
MPI_MODE_RDWR - reading and writing
MPI_MODE_WRONLY - write only
MPI_MODE_CREATE - create the file if it does not exist
MPI_MODE_EXCL - raise an error if the file already exists and MPI_MODE_CREATE is specified
MPI_MODE_DELETE_ON_CLOSE - delete file on close
MPI_MODE_UNIQUE_OPEN - file will not be concurrently opened elsewhere
MPI_MODE_SEQUENTIAL - file will only be accessed sequentially
MPI_MODE_APPEND - set initial position of all file pointers to end of file

In C and C++: You can use bit vector OR to combine these integer constants.

In Fortran: You can use the bit vector IOR intrinsic to combine these integers. If addition is used, each constant should only appear once.

MPI-IO depends on hidden threads that use MPI message passing. MPI-IO cannot be used with MP_SINGLE_THREAD set to **yes**.

The default for MP_CSS_INTERRUPT is **no**. If you do not override the default, MPI-IO enables interrupts while files are open. If you have forced interrupts to **yes** or **no**, MPI-IO does not alter your selection.

Parameter consistency checking is only performed if the environment variable MP_EUIDEVELOP is set to **yes**. If this variable is set and the amodes specified are not identical, the error **Inconsistent amodes** will be raised on some tasks. Similarly, if this variable is set and the file inodes associated with the file names are not identical, the error **Inconsistent file inodes** will be raised on some tasks. In either case, the error **Consistency error occurred on another task** will be raised on the other tasks.

When MPI-IO is used correctly, a file name will be represented at every task by the same file system. In one detectable error situation, a file will appear to be on different file system types. For example, a particular file could be visible to some tasks as a GPFS file and to others as NFS-mounted.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Invalid communicator **comm** is not a valid communicator.

Can't use an intercommunicator
comm is an intercommunicator.

Conflicting collective operations on communicator

Returning Errors (MPI Error Class):

Pathname too long (MPI_ERR_BAD_FILE)
File name must contain less than 1024 characters.

Invalid access mode (MPI_ERR_AMODE)
amode is not a valid access mode.

Invalid file system type (MPI_ERR_OTHER)
filename refers to a file belonging to a file system of an unsupported type.

Invalid info (MPI_ERR_INFO)
info is not a valid **info** object.

Locally detected error occurred on another task (MPI_ERR_ARG)
Local parameter check failed on other task(s).

Inconsistent file inodes (MPI_ERR_NOT_SAME)
Local filename corresponds to a file inode that is not consistent with that associated with the filename of other task(s).

Inconsistent file system types (MPI_ERR_NOT_SAME)
Local file system type associated with **filename** is not identical to that of other task(s).

Inconsistent amodes (MPI_ERR_NOT_SAME)
Local **amode** is not consistent with the **amode** of other task(s).

Consistency error occurred on another task (MPI_ERR_ARG)
Consistency check failed on other task(s).

Permission denied (MPI_ERR_ACCESS)
Access to the file was denied.

File already exists (MPI_ERR_FILE_EXISTS)
MPI_MODE_CREATE and MPI_MODE_EXCL are set and the file exists.

File or directory does not exist (MPI_ERR_NO_SUCH_FILE)
The file does not exist and MPI_MODE_CREATE is not set, or a directory in the path does not exist.

Not enough space in file system (MPI_ERR_NO_SPACE)
The directory or the file system is full.

File is a directory (MPI_ERR_BAD_FILE)
The file is a directory.

Read-only file system (MPI_ERR_READ_ONLY)
The file resides in a read-only file system and write access is required.

Internal open failed (MPI_ERR_IO)
An internal **open** operation on the file failed.

| **Internal stat failed (MPI_ERR_IO)**
| An internal **stat** operation on the file failed.

| **Internal fstat failed (MPI_ERR_IO)**
| An internal **fstat** operation on the file failed.

| **Internal fstatvfs failed (MPI_ERR_IO)**
| An internal **fstatvfs** operation on the file failed.

| Related Information

| MPI_FILE_CLOSE
| MPI_FILE_SET_VIEW
| MPI_FINALIZE

MPI_FILE_READ_AT, MPI_File_read_at

Purpose

Reads a file starting at the position specified by *offset*.

C Synopsis

```
#include <mpi.h>
int MPI_File_read_at (MPI_File fh,MPI_Offset offset,void *buf,
    int count,MPI_Datatype datatype,MPI_Status *status);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_FILE_READ_AT(INTEGER FH,INTEGER(KIND=MPI_OFFSET_KIND) OFFSET,
    CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,
    INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
```

Parameters

fh	is the file handle (handle) (IN).
offset	is the file offset (long long) (IN).
buf	is the initial address of buffer (choice) (OUT).
count	is the number of items in buffer (integer) (IN).
datatype	is the datatype of each buffer element (handle) (IN).
status	is the status object (status) (OUT).
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_FILE_READ_AT attempts to read from the file referred to by **fh** **count** items of type **datatype** into the buffer **buf**, starting at the offset **offset**, relative to the current view. The call returns only when data is available in **buf**. **status** contains the number of bytes successfully read and accessor functions MPI_GET_COUNT and MPI_GET_ELEMENTS allow you to extract from **status** the number of items and the number of intrinsic MPI elements successfully read, respectively. You can check for a read beyond the end of file condition by comparing the number of items requested with the number of items actually read.

Notes

Note that when you specify a value for the **offset** argument, constants of the appropriate type should be used. In Fortran, constants of type INTEGER(KIND=8) should be used, for example, 45_8.

Passing MPI_STATUS_IGNORE for the **status** argument is not supported in this release.

If an error is raised, the number of bytes contained in the **status** argument is meaningless.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Returning Errors (MPI Error Class):

Permission denied (MPI_ERR_ACCESS)
The file was opened in write-only mode.

Invalid file handle (MPI_ERR_FILE)
fh is not a valid file handle.

Invalid count (MPI_ERR_COUNT)
count is not an invalid count.

MPI_DATATYPE_NULL not valid (MPI_ERR_TYPE)
datatype has already been freed.

Undefined datatype (MPI_ERR_TYPE)
datatype is not a defined datatype.

Invalid datatype (MPI_ERR_TYPE)
datatype can be neither MPI_LB nor MPI_UB.

Uncommitted datatype (MPI_ERR_TYPE)
datatype must be committed.

Unsupported operation on sequential access file (MPI_ERR_UNSUPPORTED_OPERATION)
MPI_MODE_SEQUENTIAL was set when the file was opened.

Invalid offset (MPI_ERR_ARG)
offset is and invalid offset.

Internal read failed (MPI_ERR_IO)
An internal **read** operation failed.

Internal lseek failed (MPI_ERR_IO)
An internal **lseek** operation failed.

Related Information

MPI_FILE_READ_AT_ALL
MPI_FILE_IREAD_AT

MPI_FILE_READ_AT_ALL, MPI_File_read_at_all

Purpose

A collective version of MPI_FILE_READ_AT.

C Synopsis

```
#include <mpi.h>
int MPI_File_read_at_all (MPI_File fh,MPI_Offset offset,void *buf,
    int count,MPI_Datatype datatype,MPI_Status *status);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_FILE_READ_AT_ALL(INTEGER FH,INTEGER(KIND=MPI_OFFSET_KIND) OFFSET,
    CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,
    INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
```

Parameters

fh	is the file handle (handle)(IN).
offset	is the file offset (long long) (IN).
buf	is the initial address of the buffer (choice) (OUT).
count	is the number of elements in buffer (integer) (IN).
datatype	is the datatype of each buffer element (handle) (IN).
status	is the status object (Status) (OUT).
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_FILE_READ_AT_ALL is the collective version of the routine MPI_FILE_READ_AT. It has the exact semantics as its counterpart. The number of bytes actually read by the calling task is returned in **status**. The call returns when the data requested by the calling task is available in **buf**. The call does not wait for accesses from other tasks associated with the file handle **fh** to have data available in their buffers.

Notes

Note that when you specify a value for the **offset** argument, constants of the appropriate type should be used. In Fortran, constants of type INTEGER(KIND=8) should be used, for example, 45_8.

Passing MPI_STATUS_IGNORE for the **status** argument is not supported in this release.

If an error is raised, the number of bytes contained in **status** is meaningless.

For additional information, see MPI_FILE_READ_AT.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Returning Errors (MPI Error Class):

Permission denied (MPI_ERR_ACCESS)
The file was opened in write-only mode.

Invalid count (MPI_ERR_COUNT)
count is an invalid count.

Invalid file handle (MPI_ERR_FILE)
fh is not a valid file handle.

MPI_DATATYPE_NULL not valid (MPI_ERR_TYPE)
datatype has already been freed.

Undefined datatype (MPI_ERR_TYPE)
datatype is not a defined datatype.

Invalid datatype (MPI_ERR_TYPE)
datatype can be neither MPI_LB nor MPI_UB.

Uncommitted datatype (MPI_ERR_TYPE)
datatype must be committed.

Unsupported operation on sequential access file (MPI_ERR_UNSUPPORTED_OPERATION)
MPI_MODE_SEQUENTIAL was set when the file was opened.

Invalid offset (MPI_ERR_ARG)
offset is an invalid offset.

Internal read failed (MPI_ERR_IO)
An internal **read** operation failed.

Internal lseek failed (MPI_ERR_IO)
An internal **lseek** operation failed.

Related Information

MPI_FILE_READ_AT
MPI_FILE_IREAD_AT

MPI_FILE_SET_ERRHANDLER, MPI_File_set_errhandler

Purpose

Associates a new error handler to a file.

C Synopsis

```
#include <mpi.h>
int MPI_File_set_errhandler (MPI_File fh,
    MPI_Errhandler errhandler);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_FILE_SET_ERRHANDLER(INTEGER FH,INTEGER ERRHANLDER,
    INTEGER IERROR)
```

Parameters

fh	is the valid file handle (handle) (IN)
errhandler	is the new error handler for the opened file (handle) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_FILE_SET_ERRHANDLER associates a new error handler to a file. If **fh** is equal to MPI_FILE_NULL, then MPI_FILE_SET_ERRHANDLER defines the new default file error handler on the calling task to be error handler **errhandler**. If **fh** is a valid file handle, then this routine associates the error handler **errhandler** with the file referred to by **fh**.

Notes

The error **Invalid error handler** is raised if **errhandler** was created with any error handler create routine other than MPI_FILE_CREATE_ERRHANDLER. You can associate the predefined error handlers, MPI_ERRORS_ARE_FATAL and MPI_ERRORS_RETURN, as well as the implementation-specific MPE_ERRORS_WARN, with file handles.

Errors

Fatal Errors:

MPI not initialized	
MPI already finalized	
Invalid file handle	**fh** must be a valid file handle or MPI_FILE_NULL.
Invalid error handler	**errhandler** must be a valid error handler.

Related Information

MPI_FILE_CREATE_ERRHANDLER
MPI_FILE_GET_ERRHANDLER
MPI_ERRHANDLER_FREE

MPI_FILE_SET_INFO, MPI_File_set_info

Purpose

Specifies new hints for an open file.

C Synopsis

```
#include <mpi.h>
int MPI_File_set_info (MPI_File fh,MPI_Info info);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_FILE_SET_INFO(INTEGER FH,INTEGER INFO,INTEGER IERROR)
```

Parameters

fh	is the file handle (handle) (INOUT)
info	is the **info** object (handle) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_FILE_SET_INFO sets any hints that the **info** object contains for **fh**. In this release, file hints are not supported, so all **info** objects will be empty. However, you are free to associate new hints with an open file. They will just be ignored by MPI.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Returning Errors (MPI Error Class):

Invalid file handle (MPI_ERR_FILE)
fh is not a valid file handle.

Invalid info (MPI_ERR_INFO)
info is not a valid **info** object.

Related Information

MPI_FILE_GET_INFO
MPI_FILE_OPEN
MPI_FILE_SET_VIEW

MPI_FILE_SET_SIZE, MPI_File_set_size

Purpose

Expands or truncates an open file.

C Synopsis

```
#include <mpi.h>
int MPI_File_set_size (MPI_File fh,MPI_Offset size);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_FILE_SET_SIZE (INTEGER FH,INTEGER(KIND=MPI_OFFSET_KIND) SIZE,
    INTEGER IERROR)
```

Parameters

fh	is the file handle (handle) (INOUT)
size	is the requested size of the file after truncation or expansion (long long) (IN).
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_FILE_SET_SIZE is a collective operation that allows you to expand or truncate the open file referred to by **fh**. All participating tasks must specify the same value for **size**. If I/O operations are pending on **fh**, then an error is returned to the participating tasks and the file is not resized.

If **size** is larger than the current file size, the file length is increased to **size** and a read of unwritten data in the extended area returns zeros. However, file blocks are not allocated in the extended area. If **size** is smaller than the current file size, the file is truncated at the position defined by **size**. File blocks located beyond this point are de-allocated.

Notes

Note that when you specify a value for the **size** argument, constants of the appropriate type should be used. In Fortran, constants of type INTEGER(KIND=8) should be used, for example, 45_8.

Parameter consistency checking is only performed if the environment variable MP_EUIDEVELOP is set to **yes**. If this variable is set and the sizes specified are not identical, the error **Inconsistent file sizes** will be raised on some tasks, and the error **Consistency error occurred on another task** will be raised on the other tasks.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Returning Errors (MPI Error Class):

Permission denied (MPI_ERR_ACCESS)
The file was opened in read-only mode.

Unsupported operation on sequential access file (MPI_ERR_UNSUPPORTED_OPERATION)
MPI_MODE_SEQUENTIAL was set when the file was opened.

Pending I/O operations (MPI_ERR_OTHER)
There are pending I/O operations.

Locally detected error occurred on another task (MPI_ERR_ARG)
Local parameter check failed on other task(s).

Invalid file handle (MPI_ERR_FILE)
fh is not a valid file handle.

Invalid file size (MPI_ERR_ARG)
Local **size** is negative

Inconsistent file sizes (MPI_ERR_NOT_SAME)
Local **size** is not consistent with the file size of other task(s)

Consistency error occurred on another task (MPI_ERR_ARG)
Consistency check failed on other task(s).

Internal ftruncate failed (MPI_ERR_IO)
An internal **ftruncate** operation on the file failed.

Related Information

MPI_FILE_GET_SIZE

MPI_FILE_SET_VIEW, MPI_File_set_view

Purpose

Associates a new view with the open file.

C Synopsis

```
#include <mpi.h>
int MPI_File_set_view (MPI_File fh,MPI_Offset disp,
    MPI_Datatype etype,MPI_Datatype filetype,
    char *datarep,MPI_Info info);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_FILE_SET_VIEW (INTEGER FH,INTEGER(KIND=MPI_OFFSET_KIND) DISP,
    INTEGER ETYPE,INTEGER FILETYPE,CHARACTER DATAREP(*),INTEGER INFO,
    INTEGER IERROR)
```

Parameters

fh	is the file handle (handle) (IN).
disp	is the displacement (long long) (IN).
etype	is the elementary datatype (handle) (IN).
filetype	is the filetype (handle) (IN).
datarep	is the data representation (string) (IN).
info	is the **info** object (handle) (IN).
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_FILE_SET_VIEW is a collective operation and associates a new view defined by **disp**, **etype**, **filetype**, and **datarep** with the open file referred to by **fh**. All participating tasks must specify the same values for **datarep** and the same extents for **etype**.

There are no further restrictions on **etype** and **filetype** except those referred to in the MPI-2 standard. No checking is performed on the validity of these datatypes. If I/O operations are pending on **fh**, an error is returned to the participating tasks and the new view is not associated with the file. The only data representation currently supported is *native*. Since in this release file hints are not supported, the **info** argument will be ignored, after its validity is checked.

Notes

Note that when you specify a value for the **disp** argument, constants of the appropriate type should be used. In Fortran, constants of type INTEGER(KIND=8) should be used, for example, 45_8.

It is expected that a call to MPI_FILE_SET_VIEW will immediately follow MPI_FILE_OPEN in many instances.

Parameter consistency checking is only performed if the environment variable MP_EUIDEVELOP is set to **yes**. If this variable is set and the extents of the elementary datatypes specified are not identical, the error **Inconsistent elementary datatypes** will be raised on some tasks and the error **Consistency error occurred on another task** will be raised on the other tasks.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Returning Errors (MPI Error Class):

Invalid displacement (MPI_ERR_ARG)
Invalid displacement.

Invalid file handle (MPI_ERR_FILE)
fh is not a valid file handle.

MPI_DATATYPE_NULL not valid (MPI_ERR_TYPE)
Either **etype** or **filetype** has already been freed.

Undefined datatype (MPI_ERR_TYPE)
etype or **filetype** is not a defined datatype.

Invalid datatype (MPI_ERR_TYPE)
etype or **filetype** can be neither MPI_LB nor MPI_UB.

Uncommitted datatype (MPI_ERR_TYPE)
Both **etype** or **filetype** must be committed.

Invalid data representation (MPI_ERR_UNSUPPORTED_DATAREP)
datarep is an invalid data representation.

Invalid info (MPI_ERR_INFO)
info is not a valid **info** object.

Pending I/O operations (MPI_ERR_OTHER)
There are pending I/O operations.

Locally detected error occurred on another task (MPI_ERR_ARG)
Local parameter check failed on other task(s).

Inconsistent elementary datatypes (MPI_ERR_NOT_SAME)
Local **etype** extent is not consistent with the elementary datatype extent of other task(s).

Consistency error occurred on another task (MPI_ERR_ARG)
Consistency check failed on other task(s).

Related Information

MPI_FILE_GET_VIEW

MPI_FILE_SYNC, MPI_File_sync

Purpose

Commits file updates of an open file to one or more storage devices.

C Synopsis

```
#include <mpi.h>
int MPI_File_sync (MPI_File fh);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_FILE_SYNC (INTEGER FH,INTEGER IERROR)
```

Parameters

fh	is the file handle (handle) (INOUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_FILE_SYNC is a collective operation. It forces the updates to the file referred to by **fh** to be propagated to the storage device(s) before it returns. If I/O operations are pending on **fh**, an error is returned to the participating tasks and no sync operation is performed on the file.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Returning Errors (MPI Error Class):

Invalid file handle (MPI_ERR_FILE)
fh is not a valid file handle.

Permission denied (MPI_ERR_ACCESS)
The file was opened in read-only mode.

Pending I/O operations (MPI_ERR_OTHER)
There are pending I/O operations.

Locally detected error occurred on another task (MPI_ERR_ARG)
Local parameter check failed on other task(s).

Internal fsync failed (MPI_ERR_IO)
An internal **fsync** operation failed.

Related Information

MPI_FILE_WRITE_AT
MPI_FILE_WRITE_AT_ALL
MPI_FILE_IWRITE_AT

MPI_FILE_WRITE_AT, MPI_File_write_at

Purpose

Writes to a file starting at the position specified by *offset*.

C Synopsis

```
#include <mpi.h>
int MPI_File_write_at (MPI_File fh,MPI_Offset offset,void *buf,
    int count,MPI_Datatype datatype,MPI_Status *status);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_FILE_WRITE_AT(INTEGER FH,INTEGER(KIND_MPI_OFFSET_KIND) OFFSET,
    CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,
    INTEGER STATUS(MPI_STATUS_SIZE),
    INTEGER IERROR)
```

Parameters

fh	is the file handle (handle) (INOUT).
offset	is the file offset (long long) (IN).
buf	is the initial address of buffer (choice) (IN).
count	is the number of elements in buffer (integer) (IN).
datatype	is the datatype of each buffer element (handle) (IN).
status	is the status object (Status) (OUT).
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_FILE_WRITE_AT attempts to write into the file referred to by **fh count** items of type **datatype** out of the buffer **buf**, starting at the offset **offset** and relative to the current view. MPI_FILE_WRITE_AT returns when it is safe to reuse **buf**. **status** contains the number of bytes successfully written and accessor functions MPI_GET_COUNT and MPI_GET_ELEMENTS allows you to extract from **status** the number of items and the number of intrinsic MPI elements successfully written, respectively.

Notes

Note that when you specify a value for the **offset** argument, constants of the appropriate type should be used. In Fortran, constants of type INTEGER(KIND=8) should be used, for example, 45_8.

Passing MPI_STATUS_IGNORE for the **status** argument is not supported in this release.

If an error is raised, the number of bytes contained in **status** is meaningless.

When the call returns, it does not necessarily mean that the write operation has completed. In particular, written data may still be in system buffers and may not have been written to storage device(s) yet. To ensure that written data is committed to the storage device(s), you must use MPI_FILE_SYNC.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Returning Errors (MPI Error Class):

Permission denied (MPI_ERR_ACCESS)
The file was opened in read-only mode.

Invalid file handle (MPI_ERR_FILE)
fh is not a valid file handle.

Invalid count (MPI_ERR_COUNT)
count is not a valid count.

MPI_DATATYPE_NULL not valid (MPI_ERR_TYPE)
datatype has already been freed.

Undefined datatype (MPI_ERR_TYPE)
datatype is not a defined datatype.

Invalid datatype (MPI_ERR_TYPE)
datatype can be neither MPI_LB nor MPI_UB.

Uncommitted datatype (MPI_ERR_TYPE)
datatype must be committed.

Unsupported operation on sequential access file (MPI_ERR_UNSUPPORTED_OPERATION)
MPI_MODE_SEQUENTIAL was set when the file was opened.

Invalid offset(MPI_ERR_ARG)
offset is an invalid offset.

Not enough space in file system (MPI_ERR_NO_SPACE)
The file system on which the file resides is full.

File too big (MPI_ERR_IO) The file has reached the maximum size allowed.

Internal write failed (MPI_ERR_IO)
An internal **write** operation failed.

Internal lseek failed (MPI_ERR_IO)
An internal **lseek** operation failed.

Related Information

MPI_FILE_WRITE_AT_ALL
MPI_FILE_IWRITE
MPI_FILE_SYNC

MPI_FILE_WRITE_AT_ALL, MPI_File_write_at_all

Purpose

A collective version of MPI_FILE_WRITE_AT.

C Synopsis

```
#include <mpi.h>
int MPI_File_write_at_all (MPI_File fh,MPI_Offset offset,void *buf,
    int count,MPI_Datatype datatype,MPI_Status *status);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_FILE_WRITE_AT_ALL (INTEGER FH,
    INTEGER (KIND=MPI_OFFSET_KIND) OFFSET,
    CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,
    INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
```

Parameters

fh	is the file handle (handle)(INOUT).
offset	is the file offset (long long) (IN).
buf	is the initial address of buffer (choice) (IN).
count	is the number of elements in buffer (integer) (IN).
datatype	is the datatype of each buffer element (handle) (IN).
status	is the status object (Status) (OUT).
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_FILE_WRITE_AT_ALL is the collective version of MPI_FILE_WRITE_AT. In **status** is stored the number of bytes actually written by the calling task. The call returns when the calling task can safely reuse **buf**. It does not wait until the storing buffers in other participating tasks can safely be re-used.

Notes

Note that when you specify a value for the **offset** argument, constants of the appropriate type should be used. In Fortran, constants of type INTEGER(KIND=8) should be used, for example, 45_8.

Passing MPI_STATUS_IGNORE for the **status** argument is not supported in this release.

If an error is raised, the number of bytes contained in **status** is meaningless.

When the call returns, it does not necessarily mean that the write operation has completed. In particular, written data may still be in system buffers and may not have been written to storage device(s) yet. To ensure that written data is committed to the storage device(s), you must use MPI_FILE_SYNC.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Returning Errors (MPI Error Class):

Permission denied (MPI_ERR_ACCESS)
The file was opened in read-only mode.

Invalid count (MPI_ERR_COUNT)
count is not a valid count.

Invalid file handle (MPI_ERR_FILE)
fh is not a valid file handle.

MPI_DATATYPE_NULL not valid (MPI_ERR_TYPE)
datatype has already been freed.

Undefined datatype (MPI_ERR_TYPE)
datatype is not a defined datatype.

Invalid datatype (MPI_ERR_TYPE)
datatype can be neither MPI_LB nor MPI_UB.

Uncommitted datatype (MPI_ERR_TYPE)
datatype must be committed.

Unsupported operation on sequential access file (MPI_ERR_UNSUPPORTED_OPERATION)
MPI_MODE_SEQUENTIAL was set when the file was opened.

Invalid offset (MPI_ERR_ARG)
offset is an invalid offset.

Not enough space in file system (MPI_ERR_NO_SPACE)
The file system on which the file resides is full.

File too big (MPI_ERR_IO) The file has reached the maximum size allowed.

Internal write failed (MPI_ERR_IO)
An internal **write** operation failed.

Internal lseek failed (MPI_ERR_IO)
An internal **lseek** operation failed.

Related Information

MPI_FILE_WRITE_AT
MPI_FILE_IWRITE_AT
MPI_FILE_SYNC

MPI_FINALIZE, MPI_Finalize

Purpose

Terminates all MPI processing.

C Synopsis

```
#include <mpi.h>
int MPI_Finalize(void);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_FINALIZE(INTEGER IERROR)
```

Parameters

IERROR is the Fortran return code. It is always the last argument.

Description

Make sure this routine is the last MPI call. Any MPI calls made after MPI_FINALIZE raise an error. You must be sure that all pending communications involving a task have completed before the task calls MPI_FINALIZE. You must also be sure that all files opened by the task have been closed before the task calls MPI_FINALIZE.

Although MPI_FINALIZE terminates MPI processing, it does not terminate the task. It is possible to continue with non-MPI processing after calling MPI_FINALIZE, but no other MPI calls (including MPI_INIT) can be made.

In a threaded environment both MPI_INIT and MPI_FINALIZE must be called on the same thread. MPI_FINALIZE closes the communication library and terminates the service threads. It does not affect any threads you created, other than returning an error if one subsequently makes an MPI call. If you had registered a SIGIO handler, it is restored as a signal handler; however, the SIGIO signal is blocked when MPI_FINALIZE returns. If you want to catch SIGIO after MPI_FINALIZE has been called, you should unblock it.

Notes

The MPI standard does not specify the state of MPI tasks after MPI_FINALIZE, therefore, an assumption that all tasks continue may not be portable. If MPI_BUFFER_ATTACH has been used and MPI_BUFFER_DETACH has been not called, there will be an implicit MPI_BUFFER_DETACH within MPI_FINALIZE. See MPI_BUFFER_DETACH.

Errors

MPI already finalized

MPI not initialized

Related Information

MPI_ABORT
| MPI_BUFFER_DETACH
MPI_INIT

MPI_GATHER, MPI_Gather

Purpose

Collects individual messages from each task in **comm** at the **root** task.

C Synopsis

```
#include <mpi.h>
int MPI_Gather(void* sendbuf,int sendcount,MPI_Datatype sendtype,
    void* recvbuf,int recvcount,MPI_Datatype recvtype,int root,
    MPI_Comm comm);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_GATHER(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,
    CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER ROOT,
    INTEGER COMM,INTEGER IERROR)
```

Parameters

sendbuf	is the starting address of the send buffer (choice) (IN)
sendcount	is the number of elements in the send buffer (integer) (IN)
sendtype	is the datatype of the send buffer elements (integer) (IN)
recvbuf	is the address of the receive buffer (choice, significant only at **root**) (OUT)
recvcount	is the number of elements for any single receive (integer, significant only at **root**) (IN)
recvtype	is the datatype of the receive buffer elements (handle, significant only at **root**) (IN)
root	is the rank of the receiving task (integer) (IN)
comm	is the communicator (handle) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine collects individual messages from each task in **comm** at the **root** task and stores them in rank order.

The type signature of **sendcount**, **sendtype** on task **i** must be equal to the type signature of **recvcount**, **recvtype** at the root. This means the amount of data sent must be equal to the amount of data received, pairwise between each task and the root. Distinct type maps between sender and receiver are allowed.

The following is information regarding MPI_GATHER arguments and tasks:

- On the task **root**, all arguments to the function are significant.
- On other tasks, only the arguments **sendbuf, sendcount, sendtype, root,** and **comm** are significant.

- The argument **root** must be the same on all tasks.

Note that the argument **revcount** at the root indicates the number of items it receives from each task. It is not the total number of items received.

A call where the specification of counts and types causes any location on the root to be written more than once is erroneous.

When you use this routine in a threaded application, make sure all collective operations on a particular communicator occur in the same order at each task. See Appendix G, "Programming Considerations for User Applications in POE" on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid communicator	
Invalid count(s)	**count** < 0
Invalid datatype(s)	
Type not committed	
Invalid root	**root** < 0 or **root** >= groupsize
Unequal message lengths	
MPI not initialized	
MPI already finalized	

Develop mode error if:

Inconsistent root

Inconsistent message lengths

Related Information

MPE_IGATHER
MPI_SCATTER
MPI_GATHER
MPI_ALLGATHER

MPI_GATHERV, MPI_Gatherv

Purpose

Collects individual messages from each task in **comm** at the **root** task. Messages can have different sizes and displacements.

C Synopsis

```
#include <mpi.h>
int MPI_Gatherv(void* sendbuf,int sendcount,MPI_Datatype sendtype,
    void* recvbuf,int recvcounts,int *displs,MPI_Datatype recvtype,
    int root,MPI_Comm comm);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_GATHERV(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,
    CHOICE RECVBUF,INTEGER RECVCOUNTS(*),INTEGER DISPLS(*),
    INTEGER RECVTYPE,INTEGER ROOT,INTEGER COMM,INTEGER IERROR)
```

Parameters

sendbuf	is the starting address of the send buffer (choice) (IN)
sendcount	is the number of elements in the send buffer (integer) (IN)
sendtype	is the datatype of the send buffer elements (handle) (IN)
recvbuf	is the address of the receive buffer (choice, significant only at **root**) (OUT)
recvcounts	integer array (of length group size) that contains the number of elements received from each task (significant only at **root**) (IN)
displs	integer array (of length group size). Entry **i** specifies the displacement relative to **recvbuf** at which to place the incoming data from task **i** (significant only at **root**) (IN)
recvtype	is the datatype of the receive buffer elements (handle, significant only at **root**) (IN)
root	is the rank of the receiving task (integer) (IN)
comm	is the communicator (handle) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine collects individual messages from each task in **comm** at the **root** task and stores them in rank order. With **recvcounts** as an array, messages can have varying sizes, and **displs** allows you the flexibility of where the data is placed on the root.

The type signature of **sendcount**, **sendtype** on task **i** must be equal to the type signature of **recvcounts[i]**, **recvtype** at the root. This means the amount of data sent must be equal to the amount of data received, pairwise between each task and the root. Distinct type maps between sender and receiver are allowed.

The following is information regarding MPI_GATHERV arguments and tasks:

- On the task **root**, all arguments to the function are significant.
- On other tasks, only the arguments. **sendbuf, sendcount, sendtype, root,** and **comm** are significant.
- The argument **root** must be the same on all tasks.

A call where the specification of sizes, types and displacements causes any location on the root to be written more than once is erroneous.

Notes

Displacements are expressed as elements of type **recvtype**, not as bytes.

When you use this routine in a threaded application, make sure all collective operations on a particular communicator occur in the same order at each task. See Appendix G, "Programming Considerations for User Applications in POE" on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid communicator	
Invalid communicator type	must be intracommunicator
Invalid count(s)	**count** < 0
Invalid datatype(s)	
Type not committed	
Invalid root	**root** < 0 or **root** >= groupsize
A send and receive have unequal message lengths	
MPI not initialized	
MPI already finalized	

Develop mode error if:

Inconsistent root

Related Information

MPE_IGATHER
MPI_GATHER

MPI_GET_COUNT, MPI_Get_count

Purpose

Returns the number of elements in a message.

C Synopsis

```
#include <mpi.h>
int MPI_Get_count(MPI_Status *status,MPI_Datatype datatype,
    int *count);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_GET_COUNT(INTEGER STATUS(MPI_STATUS_SIZE),INTEGER DATATYPE,
    INTEGER COUNT,INTEGER IERROR)
```

Parameters

status	is a status object (status) (IN). Note that in Fortran a single status object is an array of integers.
datatype	is the datatype of each message element (handle) (IN)
count	is the number of elements (integer) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This subroutine returns the number of elements in a message. The **datatype** argument and the argument provided by the call that set the **status** variable should match.

When one of the MPI wait or test calls returns **status** for a non-blocking operation request and the corresponding blocking operation does not provide a **status** argument, the **status** from this wait/test does not contain meaningful source, tag or message size information.

Errors

Invalid datatype

Type not committed

MPI not initialized

MPI already finalized

Related Information

MPI_IRECV
MPI_WAIT
MPI_RECV
MPI_PROBE

MPI_GET_ELEMENTS, MPI_Get_elements

Purpose

Returns the number of basic elements in a message.

C Synopsis

```
#include <mpi.h>
int MPI_Get_elements(MPI_Status *status,MPI_Datatype datatype,
    int *count);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_GET_ELEMENTS(INTEGER STATUS(MPI_STATUS_SIZE),INTEGER DATATYPE,
    INTEGER COUNT,INTEGER IERROR)
```

Parameters

status	is a status of object (status) (IN). Note that in Fortran a single status object is an array of integers.
datatype	is the datatype used by the operation (handle) (IN)
count	is an integer specifying the number of basic elements (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine returns the number of type map elements in a message. When the number of bytes does not align with the type signature, MPI_GET_ELEMENTS returns MPI_UNDEFINED. For example, given type signature (int, short, int, short) a 10 byte message would return 3 while an 8 byte message would return MPI_UNDEFINED.

When one of the MPI wait or test calls returns **status** for a nonblocking operation request and the corresponding blocking operation does not provide a **status** argument, the **status** from this wait/test does not contain meaningful source, tag or message size information.

Errors

Invalid datatype

Type is not committed

MPI not initialized

MPI already finalized

Related Information

MPI_GET_COUNT

MPI_GET_PROCESSOR_NAME, MPI_Get_processor_name

Purpose

Returns the name of the local processor.

C Synopsis

```
#include <mpi.h>
int MPI_Get_processor_name(char *name,int *resultlen);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_GET_PROCESSOR_NAME(CHARACTER NAME(*),INTEGER RESULTLEN,
      INTEGER IERROR)
```

Parameters

name	is a unique specifier for the actual node (OUT)
resultlen	specifies the printable character length of the result returned in **name** (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine returns the name of the local processor at the time of the call. The name is a character string from which it is possible to identify a specific piece of hardware. **name** represents storage that is at least MPI_MAX_PROCESSOR_NAME characters long and MPI_GET_PROCESSOR_NAME can write up to this many characters in **name**.

The actual number of characters written is returned in **resultlen**. The returned **name** is a null terminated C string with the terminating byte not counted in **resultlen**.

Errors

MPI not initialized

MPI already finalized

MPI_GET_VERSION, MPI_Get_version

Purpose

Returns the version of the MPI standard supported in this release.

C Synopsis

```
#include <mpi.h>
int MPI_Get_version(int *version,int *subversion);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_GET_VERSION(INTEGER VERSION, INTEGER SUBVERSION, INTEGER IERROR)
```

Parameters

version	MPI standard version number (integer) (OUT)
subversion	MPI standard subversion number (integer) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine is used to determine the version of the MPI standard supported by the MPI implementation.

There are also new symbolic constants, MPI_VERSION and MPI_SUBVERSION, provided in mpi.h and mpif.h that provide similar compile-time information.

MPI_GET_VERSION can be called before MPI_INIT.

MPI_GRAPH_CREATE, MPI_Graph_create

Purpose

Creates a new communicator containing graph topology information.

C Synopsis

```
#include <mpi.h>
MPI_Graph_create(MPI_Comm comm_old,int nnodes, int *index,
    int *edges,int reorder,MPI_Comm *comm_graph);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_GRAPH_CREATE(INTEGER COMM_OLD,INTEGER NNODES,INTEGER INDEX(*),
    INTEGER EDGES(*),INTEGER REORDER,INTEGER COMM_GRAPH,
    INTEGER IERROR)
```

Parameters

comm_old	is the input communicator (handle) (IN)
nnodes	is an integer specifying the number of nodes in the graph (IN)
index	is an array of integers describing node degrees (IN)
edges	is an array of integers describing graph edges (IN)
reorder	if true, ranking may be reordered (logical) (IN)
comm_graph	is the communicator with the graph topology added (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine creates a new communicator containing graph topology information provided by **nnodes, index, edges,** and **reorder**. MPI_GRAPH_CREATE returns the handle for this new communicator in **comm_graph**.

If there are more tasks in **comm_old** then **nnodes**, some tasks are returned **comm_graph** as MPI_COMM_NULL.

Notes

The reorder argument is currently ignored.

The following is an example showing how to define the arguments **nnodes**, **index**, and **edges**. Assume there are four tasks (0, 1, 2, 3) with the following adjacency matrix:

Task	**Neighbors**
0	1, 3
1	0
2	3
3	0, 2

Then the input arguments are:

Argument	**Input**
nnodes	4
index	2, 3, 4, 6
edges	1, 3, 0, 3, 0, 2

Thus, in C, **index[0]** is the degree of node zero, and **index[i]–index[i–1]** is the degree of node **i**, **i**=**1**, **...**, **nnodes–1**. The list of neighbors of node zero is stored in **edges[j]**, for 0 ≥ **j** ≥ **index[0]–1** and the list of neighbors of node **i**, **i** > **0**, is stored in **edges[j]**, **index[i–1]** ≥ **j** ≥ **index[i]–1**.

In Fortran, **index(1)** is the degree of node zero, and **index(i+1)– index(i)** is the degree of node **i**, **i**=**1**, **...**, **nnodes–1**. The list of neighbors of node zero is stored in **edges(j)**, for **1** ≥ **j** ≥ **index(1)** and the list of neighbors of node **i**, **i** > 0, is stored in **edges(j)**, **index(i)+1** ≥ **j** ≥ **index(i+1)**.

Observe that because node 0 indicates node 1 is a neighbor, that node 1 must indicate that node 0 is its' neighbor. For any edge A→B the edge B→A must also be specified.

Errors

MPI not initialized	
MPI already finalized	
Invalid communicator	
Invalid communicator type	must be intracommunicator
Invalid nnodes	**nnodes**<0 or **nnodes** > groupsize
Invalid node degree	(**index[i]**–**index[i–1]**) < 0
Invalid neighbor	**edges[i]** < 0 or **edges[i]**>=**nnodes**
Asymmetric graph	
Conflicting collective operations on communicator	

Related Information

MPI_CART_CREATE

MPI_GRAPH_GET, MPI_Graph_get

Purpose

Retrieves graph topology information from a communicator.

C Synopsis

```
#include <mpi.h>
MPI_Graph_get(MPI_Comm comm,int maxindex,int maxedges,
      int *index,int *edges);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_GRAPH_GET(INTEGER COMM,INTEGER MAXINDEX,INTEGER MAXEDGES,
      INTEGER INDEX(*),INTEGER EDGES(*),INTEGER IERROR)
```

Parameters

comm	is a communicator with graph topology (handle) (IN)
maxindex	is an integer specifying the length of **index** in the calling program (IN)
maxedges	is an integer specifying the length of **edges** in the calling program (IN)
index	is an array of integers containing node degrees (OUT)
edges	is an array of integers containing node neighbors (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine retrieves the **index** and **edges** graph topology information associated with a communicator.

Errors

MPI not initialized

MPI already finalized

Invalid communicator

No topology

Invalid topology type	topology type must be graph
Invalid array size	**maxindex** < 0 or **maxedges** < 0

Related Information

MPI_GRAPHDIMS_GET
MPI_GRAPH_CREATE

MPI_GRAPH_MAP, MPI_Graph_map

Purpose

Computes placement of tasks on the physical machine.

C Synopsis

```
#include <mpi.h>
MPI_Graph_map(MPI_Comm comm,int nnodes,int *index,int *edges,int *newrank);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_GRAPH_MAP(INTEGER COMM,INTEGER NNODES,INTEGER INDEX(*),
              INTEGER EDGES(*),INTEGER NEWRANK,INTEGER IERROR)
```

Parameters

comm	is the input communicator (handle) (IN)
nnodes	is the number of graph nodes (integer) (IN)
index	is an integer array specifying node degrees (IN)
edges	is an integer array specifying node adjacency (IN)
newrank	is the reordered rank,or MPI_Undefined if the calling task does not belong to the graph (integer) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_GRAPH_MAP allows MPI to compute an optimal placement for the calling task on the physical machine by reordering the tasks in **comm**.

Notes

MPI_CART_MAP returns **newrank** as the original rank of the calling task if it belongs to the grid or MPI_UNDEFINED if it does not. Currently, no reordering is done by this function.

Errors

Invalid communicator	
Invalid communicator type	must be intracommunicator
Invalid nnodes	**nnodes** <0 or **nnodes** > groupsize
Invalid node degree	**index[i]** < 0
Invalid neighbors	**edges[i]** < 0 or **edges[i]** >= **nnodes**
MPI not initialized	
MPI already finalized	

Related Information

MPI_GRAPH_CREATE
MPI_CART_MAP

MPI_GRAPH_NEIGHBORS, MPI_Graph_neighbors

Purpose

Returns the neighbors of the given task.

C Synopsis

```
#include <mpi.h>
MPI_Graph_neighbors(MPI_Comm comm,int rank,int maxneighbors,int *neighbors);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_GRAPH_NEIGHBORS(MPI_COMM COMM,INTEGER RANK,INTEGER MAXNEIGHBORS,
      INTEGER NNEIGHBORS(*),INTEGER IERROR)
```

Parameters

comm	is a communicator with graph topology (handle) (IN)
rank	is the rank of a task within group of **comm** (integer) (IN)
maxneighbors	is the size of array **neighbors** (integer) (IN)
neighbors	is the ranks of tasks that are neighbors of the specified task (array of integer) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine retrieves the adjacency information for a particular task.

Errors

Invalid array size	**maxneighbors** < 0
Invalid rank	**rank** < 0 or **rank** > groupsize
MPI not initialized	
MPI already finalized	
Invalid communicator	
No topology	
Invalid topology type	no graph topology associate with communicator

Related Information

MPI_GRAPH_NEIGHBORS_COUNT
MPI_GRAPH_CREATE

MPI_GRAPH_NEIGHBORS_COUNT, MPI_Graph_neighbors_count

Purpose

Returns the number of neighbors of the given task.

C Synopsis

```
#include <mpi.h>
MPI_Graph_neighbors_count(MPI_Comm comm,int rank,
int *neighbors);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_GRAPH_NEIGHBORS_COUNT(INTEGER COMM,INTEGER RANK,
INTEGER NEIGHBORS(*),INTEGER IERROR)
```

Parameters

comm	is a communicator with graph topology (handle) (IN)
rank	is the rank of a task within **comm** (integer) (IN)
neighbors	is the number of neighbors of the specified task (integer) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine returns the number of neighbors of the given task.

Errors

Invalid rank **rank** < 0 or **rank** > = groupsize

MPI not initialized

MPI already finalized

Invalid communicator

No graph topology associated with communicator

Invalid topology type

Related Information

MPI_GRAPH_NEIGHBORS
MPI_GRAPH_CREATE

MPI_GRAPHDIMS_GET, MPI_Graphdims_get

Purpose

Retrieves graph topology information from a communicator.

C Synopsis

```
#include <mpi.h>
MPI_Graphdims_get(MPI_Comm comm,int *nnodes,int *nedges);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_GRAPHDIMS_GET(INTEGER COMM,INTEGER NNDODES,INTEGER NEDGES,
      INTEGER IERROR)
```

Parameters

comm	is a communicator with graph topology (handle) (IN)
nnodes	is an integer specifying the number of nodes in the graph. The number of nodes and the number of tasks in the group are equal. (OUT)
nedges	is an integer specifying the number of edges in the graph. (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine retrieves the number of nodes and the number of edges in the graph topology associated with a communicator.

Errors

MPI not initialized

MPI already finalized

Invalid communicator

No topology

Invalid topology type	topology type must be graph

Related Information

MPI_GRAPH_GET
MPI_GRAPH_CREATE

MPI_GROUP_COMPARE, MPI_Group_compare

Purpose

Compares the contents of two task groups.

C Synopsis

```
#include <mpi.h>
int MPI_Group_compare(MPI_Group group1,MPI_Group group2,
    int *result);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_GROUP_COMPARE(INTEGER GROUP1,INTEGER GROUP2,INTEGER RESULT,
    INTEGER IERROR)
```

Parameters

group1	is the first group (handle) (IN)
group2	is the second group (handle) (IN)
result	is the result (integer) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine compares the contents of two task groups and returns one of the following:

MPI_IDENT	both groups have the exact group members and group order
MPI_SIMILAR	group members are the same but group order is different
MPI_UNEQUAL	group size and/or members are different

Errors

Invalid group(s)

MPI not initialized

MPI already finalized

Related Information

MPI_COMM_COMPARE

MPI_GROUP_DIFFERENCE, MPI_Group_difference

Purpose

Creates a new group that is the difference of two existing groups.

C Synopsis

```
#include <mpi.h>
int MPI_Group_difference(MPI_Group group1,MPI_Group group2,
    MPI_Group *newgroup);
```

Fortran Synopsis

include 'mpif.h'

```
MPI_GROUP_DIFFERENCE(INTEGER GROUP1,INTEGER GROUP2,
      INTEGER NEWGROUP,INTEGER IERROR)
```

Parameters

group1	is the first group (handle) (IN)
group2	is the second group (handle) (IN)
newgroup	is the difference group (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine creates a new group that is the difference of two existing groups. The new group consists of all elements of the first group (**group1**) that are not in the second group (**group2**), and is ordered as in the first group.

Errors

Invalid group(s)

MPI not initialized

MPI already finalized

Related Information

MPI_GROUP_UNION
MPI_GROUP_INTERSECTION

MPI_GROUP_EXCL, MPI_Group_excl

Purpose

Creates a new group by excluding selected tasks of an existing group.

C Synopsis

```
#include <mpi.h>
int MPI_Group_excl(MPI_Group group,int n,int *ranks,
    MPI_Group *newgroup);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_GROUP_EXCL(INTEGER GROUP,INTEGER N,INTEGER RANKS(*),
      INTEGER NEWGROUP,INTEGER IERROR)
```

Parameters

group	is the group (handle) (IN)
n	is the number of elements in array **ranks** (integer) (IN)
ranks	is the array of integer ranks in **group** not to appear in **newgroup** (IN)
newgroup	is the new group derived from above preserving the order defined by **group** (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine removes selected tasks from an existing group to create a new group.

MPI_GROUP_EXCL creates a group of tasks **newgroup** obtained by deleting from **group** tasks with ranks **ranks[0],... ranks[n-1]**. The ordering of tasks in **newgroup** is identical to the ordering in **group**. Each of the **n** elements of **ranks** must be a valid rank in **group** and all elements must be distinct. If **n**= 0, then **newgroup** is identical to **group**.

Errors

Invalid group	
Invalid size	n <0 or **n** > groupsize
Invalid rank(s)	**ranks**[i] < 0 or **ranks**[i] > = groupsize
Duplicate rank(s)	
MPI not initialized	
MPI already finalized	

Related Information

MPI_GROUP_INCL
MPI_GROUP_RANGE_EXCL
MPI_GROUP_RANGE_INCL

MPI_GROUP_FREE, MPI_Group_free

Purpose

Marks a group for deallocation.

C Synopsis

```
#include <mpi.h>
int MPI_Group_free(MPI_Group *group);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_GROUP_FREE(INTEGER GROUP,INTEGER IERROR)
```

Parameters

group	is the group (handle) (INOUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_GROUP_FREE sets the handle **group** to MPI_GROUP_NULL and marks the group object for deallocation. Actual deallocation occurs only after all operations involving **group** are completed. Any active operation using **group** completes normally but no new calls with meaningful references to the freed group are possible.

Errors

Invalid group

MPI not initialized

MPI already finalized

MPI_GROUP_INCL, MPI_Group_incl

Purpose

Creates a new group consisting of selected tasks from an existing group.

C Synopsis

```
#include <mpi.h>
int MPI_Group_incl(MPI_Group group,int n,int *ranks,
    MPI_Group *newgroup);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_GROUP_INCL(INTEGER GROUP,INTEGER N,INTEGER RANKS(*),
      INTEGER NEWGROUP,INTEGER IERROR)
```

Parameters

group	is the group (handle) (IN)
n	is the number of elements in array **ranks** and the size of **newgroup**(integer) (IN)
ranks	is the ranks of tasks in **group** to appear in **newgroup** (array of integers) (IN)
newgroup	is the new group derived from above in the order defined by **ranks** (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine creates a new group consisting of selected tasks from an existing group.

MPI_GROUP_INCL creates a group **newgroup** consisting of **n** tasks in **group** with ranks **rank[0], ..., rank[n-1]**. The task with rank **i** in **newgroup** is the task with rank **ranks[i]** in **group**.

Each of the **n** elements of **ranks** must be a valid rank in **group** and all elements must be distinct. If **n** = 0, then **newgroup** is MPI_GROUP_EMPTY. This function can be used to reorder the elements of a group.

Errors

Invalid group	
Invalid size	n <0 or **n** > groupsize
Invalid rank(s)	**ranks**[i] < 0 or **ranks**[i] >= groupsize
Duplicate rank(s)	
MPI not initialized	
MPI already finalized	

Related Information

MPI_GROUP_EXCL
MPI_GROUP_RANGE_INCL
MPI_GROUP_RANGE_EXCL

MPI_GROUP_INTERSECTION, MPI_Group_intersection

Purpose

Creates a new group that is the intersection of two existing groups.

C Synopsis

```
#include <mpi.h>
int MPI_Group_intersection(MPI_Group group1,MPI_Group group2,
    MPI_Group *newgroup);
```

Fortran Synopsis

include 'mpif.h'

```
MPI_GROUP_INTERSECTION(INTEGER GROUP1,INTEGER GROUP2,
      INTEGER NEWGROUP,INTEGER IERROR)
```

Parameters

group1	is the first group (handle) (IN)
group2	is the second group (handle) (IN)
newgroup	is the intersection group (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine creates a new group that is the intersection of two existing groups. The new group consists of all elements of the first group (**group1**) that are also part of the second group (**group2**), and is ordered as in the first group.

Errors

Invalid group(s)

MPI not initialized

MPI already finalized

Related Information

MPI_GROUP_UNION
MPI_GROUP_DIFFERENCE

MPI_GROUP_RANGE_EXCL, MPI_Group_range_excl

Purpose

Creates a new group by removing selected ranges of tasks from an existing group.

C Synopsis

```
#include <mpi.h>
int MPI_Group_range_excl(MPI_Group group,int n,
    int ranges[][3],MPI_Group *newgroup);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_GROUP_RANGE_EXCL(INTEGER GROUP,INTEGER N,INTEGER RANGES(3,*),
    INTEGER NEWGROUP,INTEGER IERROR)
```

Parameters

group	is the group (handle) (IN)
n	is the number of triplets in array ranges (integer) (IN)
ranges	is an array of integer triplets of the form (first rank, last rank, stride) specifying the ranks in **group** of tasks that are to be excluded from the output group **newgroup**. (IN)
newgroup	is the new group derived from above that preserves the order in **group** (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine creates a new group by removing selected ranges of tasks from an existing group. Each computed rank must be a valid rank in **group** and all computed ranks must be distinct.

The function of this routine is equivalent to expanding the array **ranges** to an array of the excluded ranks and passing the resulting array of ranks and other arguments to MPI_GROUP_EXCL. A call to MPI_GROUP_EXCL is equivalent to a call to MPI_GROUP_RANGE_EXCL with each rank **i** in **ranks** replaced by the triplet (i,i,1) in the argument **ranges**.

Errors

Invalid group	
Invalid size	**n** < 0 or **n** > groupsize
Invalid rank(s)	a computed rank < 0 or >= groupsize
Duplicate rank(s)	
Invalid stride(s)	**stride[i]** = **0**
Too many ranks	Number of ranks > groupsize
MPI not initialized	

MPI already finalized

Related Information

MPI_GROUP_RANGE_INCL
MPI_GROUP_EXCL
MPI_GROUP_INCL

MPI_GROUP_RANGE_INCL, MPI_Group_range_incl

Purpose

Creates a new group consisting of selected ranges of tasks from an existing group.

C Synopsis

```
#include <mpi.h>
int MPI_Group_range_incl(MPI_Group group,int n,
    int ranges[][3],MPI_Group *newgroup);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_GROUP_RANGE_INCL(INTEGER GROUP,INTEGER N,INTEGER RANGES(3,*),
    INTEGER NEWGROUP,INTEGER IERROR)
```

Parameters

group	is the group (handle) (IN)
n	is the number of triplets in array **ranges** (integer) (IN)
ranges	is a one-dimensional array of integer triplets of the form (first rank, last rank, stride) indicating ranks in **group** of tasks to be included in **newgroup** (IN)
newgroup	is the new group derived from above in the order defined by **ranges** (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine creates a new group consisting of selected ranges of tasks from an existing group. The function of this routine is equivalent to expanding the array of ranges to an array of the included ranks and passing the resulting array of ranks and other arguments to MPI_GROUP_INCL. A call to MPI_GROUP_INCL is equivalent to a call to MPI_GROUP_RANGE_INCL with each rank **i** in **ranks** replaced by the triplet (i,i,1) in the argument **ranges**.

Errors

Invalid group	
Invalid size	**n** <0 or **n** > groupsize
Invalid rank(s)	a computed rank < 0 or >= groupsize
Duplicate rank(s)	
Invalid stride(s)	**stride[i]** = 0
Too many ranks	**nranks** > groupsize
MPI not initialized	
MPI already finalized	

Related Information

MPI_GROUP_RANGE_EXCL
MPI_GROUP_INCL
MPI_GROUP_EXCL

MPI_GROUP_RANK, MPI_Group_rank

Purpose

Returns the rank of the local task with respect to **group**.

C Synopsis

```
#include <mpi.h>
int MPI_Group_rank(MPI_Group group,int *rank);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_GROUP_RANK(INTEGER GROUP,INTEGER RANK,INTEGER IERROR)
```

Parameters

group	is the group (handle) (IN)
rank	is an integer that specifies the rank of the calling task in group or MPI_UNDEFINED if the task is not a member. (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine returns the rank of the local task with respect to **group**. This local operation does not require any intertask communication.

Errors

Invalid group

MPI not initialized

MPI already finalized

Related Information

MPI_COMM_RANK

MPI_GROUP_SIZE, MPI_Group_size

Purpose

Returns the number of tasks in a group.

C Synopsis

```
#include <mpi.h>
int MPI_Group_size(MPI_Group group,int *size);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_GROUP_SIZE(INTEGER GROUP,INTEGER SIZE,INTEGER IERROR)
```

Parameters

group	is the group (handle) (IN)
size	is the number of tasks in the group (integer) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine returns the number of tasks in a group. This is a local operation and does not require any intertask communication.

Errors

Invalid group

MPI not initialized

MPI already finalized

Related Information

MPI_COMM_SIZE

MPI_GROUP_TRANSLATE_RANKS, MPI_Group_translate_ranks

Purpose

Converts task ranks of one group into ranks of another group.

C Synopsis

```
#include <mpi.h>
int MPI_Group_translate_ranks(MPI_Group group1,int n,
    int *ranks1,MPI_Group group2,int *ranks2);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_GROUP_TRANSLATE_RANKS(INTEGER GROUP1, INTEGER N,
    INTEGER RANKS1(*),INTEGER GROUP2,INTEGER RANKS2(*),INTEGER IERROR)
```

Parameters

group1	is group1 (handle) (IN)
n	is an integer that specifies the number of ranks in **ranks1** and **ranks2** arrays (IN)
ranks1	is an array of zero or more valid ranks in **group1** (IN)
group2	is group2 (handle) (IN)
ranks2	is an array of corresponding ranks in **group2**. If the task of **ranks1(i)** is not a member of **group2**, **ranks2(i)** returns MPI_UNDEFINED. (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This subroutine converts task ranks of one group into ranks of another group. For example, if you know the ranks of tasks in one group, you can use this function to find the ranks of tasks in another group.

Errors

Invalid group(s)	
Invalid rank count	**n** < 0
Invalid rank	**ranks1[i]** < 0 or **ranks1[i]** > &equals size of **group1**
MPI not initialized	
MPI already finalized	

Related Information

MPI_COMM_COMPARE

MPI_GROUP_UNION, MPI_Group_union

Purpose

Creates a new group that is the union of two existing groups.

C Synopsis

```
#include <mpi.h>
int MPI_Group_union(MPI_Group group1,MPI_Group group2,
    MPI_Group *newgroup);
```

Fortran Synopsis

include 'mpif.h'

```
MPI_GROUP_UNION(INTEGER GROUP1,INTEGER GROUP2,INTEGER NEWGROUP,
      INTEGER IERROR)
```

Parameters

group1	is the first group (handle) (IN)
group2	is the second group (handle) (IN)
newgroup	is the union group (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine creates a new group that is the union of two existing groups. The new group consists of the elements of the first group (**group1**) followed by all the elements of the second group (**group2**) not in the first group.

Errors

Invalid group(s)

MPI not initialized

MPI already finalized

Related Information

MPI_GROUP_INTERSECTION
MPI_GROUP_DIFFERENCE

MPI_IBSEND, MPI_Ibsend

Purpose

Performs a nonblocking buffered mode send operation.

C Synopsis

```
#include <mpi.h>
int MPI_Ibsend(void* buf,int count,MPI_Datatype datatype,
    int dest,int tag,MPI_Comm comm,MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_IBSEND(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,
    INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Parameters

buf	is the initial address of the send buffer (choice) (IN)
count	is the number of elements in the send buffer (integer) (IN)
datatype	is the datatype of each send buffer element (handle) (IN)
dest	is the rank of the destination task in **comm** (integer) (IN)
tag	is the message tag (integer) (IN)
comm	is the communicator (handle) (IN)
request	is the communication request (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_IBSEND starts a buffered mode, nonblocking send. The send buffer may not be modified until the request has been completed by MPI_WAIT, MPI_TEST, or one of the other MPI wait or test functions.

Notes

See MPI_BSEND for additional information.

Errors

Invalid count	**count** < 0
Invalid datatype	
Invalid destination	
Type not committed	**dest** < 0 or **dest** > = groupsize
Invalid tag	**tag** < 0
Invalid comm	
MPI not initialized	

MPI already finalized

Develop mode error if:

Illegal buffer update

Related Information

MPI_BSEND
MPI_BSEND_INIT
MPI_WAIT
MPI_BUFFER_ATTACH

MPI_INFO_CREATE, MPI_Info_create

Purpose

Creates a new **info** object.

C Synopsis

```
#include <mpi.h>
int MPI_Info_create (MPI_Info *info);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_INFO_CREATE (INTEGER INFO,INTEGER IERROR)
```

Parameters

info	is the **info** object created (handle)(OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_INFO_CREATE creates a new **info** object and returns a handle to it in the **info** argument.

Because this release does not recognize any key, **info** objects are always empty.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Related Information

MPI_INFO_FREE
MPI_INFO_SET
MPI_INFO_GET
MPI_INFO_GET_NKEYS
MPI_INFO_GET_VALUELEN
MPI_INFO_GET_NTHKEY
MPI_INFO_DELETE
MPI_INFO_DUP

MPI_INFO_DELETE, MPI_Info_delete

Purpose

Deletes a (*key, value*) pair from an **info** object.

C Synopsis

```
#include <mpi.h>
int MPI_Info_delete (MPI_Info info,char *key);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_INFO_DELETE (INTEGER INFO,CHARACTER KEY(*),
    INTEGER IERROR)
```

Parameters

info	is the **info** object (handle)(OUT)
key	is the key of the pair to be deleted (string)(IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_INFO_DELETE deletes a pair (**key, value**) from **info**. If the **key** is not recognized by MPI, it is ignored and the call returns MPI_SUCCESS and has no effect on **info**.

Because this release does not recognize any key, this call always returns MPI_SUCCESS and has no effect on **info**.

Errors

Fatal Errors:

MPI not initialized	
MPI already finalized	
Invalid info	**info** is not a valid **info** object
Invalid info key	**key** must contain less than 128 characters

Related Information

MPI_INFO_CREATE
MPI_INFO_SET
MPI_INFO_GET
MPI_INFO_GET_NKEYS
MPI_INFO_GET_VALUELEN
MPI_INFO_GET_NTHKEY
MPI_INFO_DUP
MPI_INFO_FREE

MPI_INFO_DUP, MPI_Info_dup

Purpose

Duplicates an **info** object.

C Synopsis

```
#include <mpi.h>
int MPI_Info_dup (MPI_Info info,MPI_Info *newinfo);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_INFO_DUP (INTEGER INFO,INTEGER NEWINFO,INTEGER IERROR)
```

Parameters

info	is the **info** object to be duplicated(handle)(IN)
newinfo	is the new **info** object (handle)(OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_INFO_DUP duplicates the **info** object referred to by **info** and returns in **newinfo** a handle to this newly created info object.

Because this release does not recognize any key, the new **info** object is empty.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Invalid info	**info** is not a valid **info** object

Related Information

MPI_INFO_CREATE
MPI_INFO_FREE
MPI_INFO_SET
MPI_INFO_GET
MPI_INFO_GET_NKEYS
MPI_INFO_GET_VALUELEN
MPI_INFO_GET_NTHKEY
MPI_INFO_DELETE

MPI_INFO_FREE, MPI_Info_free

Purpose

Frees the **info** object referred to by the **info** argument and sets it to MPI_INFO_NULL.

C Synopsis

```
#include <mpi.h>
int MPI_Info_free (MPI_Info *info);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_INFO_FREE (INTEGER INFO,INTEGER IERROR)
```

Parameters

info	is the **info** object (handle)(IN/OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_INFO_FREE frees the **info** object referred to by the **info** argument and sets **info** to MPI_INFO_NULL.

Errors

Fatal Errors:

MPI not initialized	
MPI already finalized	
Invalid info	**info** is not a valid **info** object

Related Information

MPI_INFO_CREATE
MPI_INFO_DELETE
MPI_INFO_SET
MPI_INFO_GET
MPI_INFO_GET_NKEYS
MPI_INFO_GET_VALUELEN
MPI_INFO_GET_NTHKEY
MPI_INFO_DUP

MPI_INFO_GET, MPI_Info_get

Purpose

Retrieves the value associated with *key* in an **info** object.

C Synopsis

```
#include <mpi.h>
int MPI_Info_get (MPI_Info info,char *key,int valuelen,
    char *value,int *flag);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_INFO_GET (INTEGER INFO,CHARACTER KEY(*),INTEGER VALUELEN,
    CHARACTER VALUE(*),LOGICAL FLAG,INTEGER IERROR)
```

Parameters

info	is the **info** object (handle)(IN)
key	is the key (string)(IN)
valuelen	is the length of the value argument (integer)(IN)
value	is the value (string)(OUT)
flag	is true if **key** is defined and is false if not (boolean)(OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_INFO_GET retrieves the value associated with **key** in the **info** object referred to by **info**.

Because this release does not recognize any key, **flag** is set to false, **value** remains unchanged, and **valuelen** is ignored.

Notes

In order to determine how much space should be allocated for the **value** argument, call MPI_INFO_GET_VALUELEN first.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Invalid info	**info** is not a valid **info** object
Invalid info key	**key** must contain less than 128 characters

Related Information

MPI_INFO_CREATE
MPI_INFO_FREE
MPI_INFO_SET
MPI_INFO_GET_NKEYS
MPI_INFO_GET_VALUELEN
MPI_INFO_GET_NTHKEY
MPI_INFO_DUP
MPI_INFO_DELETE

MPI_INFO_GET_NKEYS, MPI_Info_get_nkeys

Purpose

Returns the number of keys defined in an **info** object.

C Synopsis

```
#include <mpi.h>
int MPI_Info_get_nkeys (MPI_Info info,int *nkeys);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_INFO_GET_NKEYS (INTEGER INFO,INTEGER NKEYS,INTEGER IERROR)
```

Parameters

info	is the **info** object (handle)(IN)
nkeys	is the number of defined keys (integer)(OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_INFO_GET_NKEYS returns in **nkeys** the number of keys currently defined in the **info** object referred to by **info**.

Because this release does not recognize any key, the number of keys returned is zero.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Invalid info	**info** is not a valid **info** object

Related Information

MPI_INFO_CREATE
MPI_INFO_FREE
MPI_INFO_SET
MPI_INFO_GET
MPI_INFO_GET_VALUELEN
MPI_INFO_GET_NTHKEY
MPI_INFO_DUP
MPI_INFO_DELETE

MPI_INFO_GET_NTHKEY, MPI_Info_get_nthkey

Purpose

Retrieves the *n*th key defined in an **info** object.

C Synopsis

```
#include <mpi.h>
int MPI_Info_get_nthkey (MPI_Info info, int n, char *key);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_INFO_GET_NTHKEY (INTEGER INFO,INTEGER N,CHARACTER KEY(*),
    INTEGER IERROR)
```

Parameters

info	is the **info** object (handle)(IN)
n	is the key number (integer)(IN)
key	is the key (string)(OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_INFO_GET_NTHKEY retrieves the **n**th key defined in the **info** object referred to by **info**. The first key defined has the rank of **0** so **n** must be greater than **– 1** but less than the number of keys returned by MPI_INFO_GET_NKEYS.

Because this release does not recognize any key, this function always raises an error.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Invalid info	**info** is not a valid **info** object
Invalid info key index	**n** must have a value between **0** and **N -1**, where **N** is the number of keys returned by MPI_INFO_GET_NKEYS

Related Information

MPI_INFO_CREATE
MPI_INFO_FREE
MPI_INFO_SET
MPI_INFO_GET
MPI_INFO_GET_VALUELEN
MPI_INFO_GET_NKEYS
MPI_INFO_DUP

| MPI_INFO_DELETE

MPI_INFO_GET_VALUELEN, MPI_Info_get_valuelen

Purpose

Retrieves the length of the value associated with a *key* of an **info** object.

C Synopsis

```
#include <mpi.h>
int MPI_Info_get_valuelen (MPI_Info info,char *key,int *valuelen,
    int *flag);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_INFO_GET_VALUELEN (INTEGER INFO,CHARACTER KEY(*),INTEGER VALUELEN,
    LOGICAL FLAG,INTEGER IERROR)
```

Parameters

info	is the **info** object (handle)(IN)
key	is the key (string)(IN)
valuelen	is the length of the value associated with **key** (integer)(OUT)
flag	is true if **key** is defined and is false if not (boolean)(OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_INFO_GET_VALUELEN retrieves the length of the value associated with the **key** in the **info** object referred to by **info**.

Because this release does not recognize any key, **flag** is set to false and **valuelen** remains unchanged.

Notes

Use this routine prior to calling MPI_INFO_GET to determine how much space must be allocated for the value parameter of MPI_INFO_GET.

Errors

Fatal Errors:

MPI not initialized	
MPI already finalized	
Invalid info	**info** is not a valid **info** object
Invalid info key	**key** must contain less than 128 characters

| ## Related Information

| MPI_INFO_CREATE
| MPI_INFO_FREE
| MPI_INFO_SET
| MPI_INFO_GET
| MPI_INFO_GET_NKEYS
| MPI_INFO_GET_NTHKEY
| MPI_INFO_DUP
| MPI_INFO_DELETE

MPI_INFO_SET, MPI_Info_set

Purpose

Adds a pair (*key*, *value*) to an **info** object.

C Synopsis

```
#include <mpi.h>
int MPI_Info_set(MPI_Info info,char *key,char *value);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_INFO_SET (INTEGER INFO,CHARACTER KEY(*),CHARACTER VALUE(*),
    INTEGER IERROR)
```

Parameters

info	is the **info** object (handle)(INOUT)
key	is the key (string)(IN)
value	is the value (string)(IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_INFO_SET adds a recognized (**key**, **value**) pair to the **info** object referred to by **info**. When MPI_INFO_SET is called with a key which is not recognized, it behaves as a no-op.

Because this release does not recognize any key, the **info** object remains unchanged.

Errors

Fatal Errors:

MPI not initialized

MPI already finalized

Invalid info	**info** is not a valid **info** object
Invalid info key	**key** must contain less than 128 characters
Invalid info value	**value** must contain less than 1024 characters

Related Information

MPI_INFO_CREATE
MPI_INFO_FREE
MPI_INFO_GET
MPI_INFO_GET_VALUELEN
MPI_INFO_GET_NKEYS
MPI_INFO_GET_NTHKEY
MPI_INFO_DUP

| MPI_INFO_DELETE

MPI_INIT, MPI_Init

Purpose

Initializes MPI.

C Synopsis

```
#include <mpi.h>
int MPI_Init(int *argc,char ***argv);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_INIT(INTEGER IERROR)
```

Parameters

IERROR is the Fortran return code. It is always the last argument.

Description

This routine initializes MPI. All MPI programs must call this routine before any other MPI routine (with the exception of MPI_INITIALIZED). More than one call to MPI_INIT by any task is erroneous.

Notes

argc and **argv** are the arguments passed to **main**. The IBM MPI implementation of the MPI Standard does not examine or modify these arguments when passed to MPI_INIT.

In a threaded environment, MPI_INIT needs to be called once per task and not once per thread. You don't need to call it on the main thread but both MPI_INIT and MPI_FINALIZE must be called on the same thread.

MPI_INIT opens a local socket and binds it to a port, sends that information to POE, receives a list of destination addresses and ports, opens a socket to send to each one, verifies that communication can be established, and distributes MPI internal state to each task.

In the signal-handling library, this work is done in the initialization stub added by POE, so that the library is open when your main program is called. MPI_INIT sets a flag saying that you called it.

In the threaded library, the work of MPI_INIT is done when the function is called.
| The local socket is not open when your main program starts. This may affect the numbering of file descriptors, the use of the environment strings, and the treatment of stdin (the MP_HOLD_STDIN variable). If an existing non-threaded program is
| relinked using the threaded library, the code prior to calling MPI_INIT should be examined with these thoughts in mind.

Also for the threaded library, if you had registered a function as an AIX signal handler for the SIGIO signal at the time that MPI_INIT was called, that function will be added to the interrupt service thread and be processed as a thread function rather than as a signal handler. You'll need to set the environment variable

MP_CSS_INTERRUPT=YES to get arriving packets to invoke the interrupt service thread.

Errors

MPI already initialized

MPI already finalized

Related Information

MPI_INITIALIZED
MPI_FINALIZE

MPI_INITIALIZED, MPI_Initialized

Purpose

Determines whether MPI is initialized.

C Synopsis

```
#include <mpi.h>
int MPI_Initialized(int *flag);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_INITIALIZED(INTEGER FLAG,INTEGER IERROR)
```

Parameters

flag	is true if MPI_INIT was called; otherwise is false.
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine determines if MPI is initialized. This and MPI_GET_VERSION are the only MPI calls that can be made before MPI_INIT is called.

Notes

Because it is erroneous to call MPI_INIT more than once per task, use MPI_INITIALIZED if there is doubt as to the state of MPI.

Related Information

MPI_INIT

MPI_INTERCOMM_CREATE, MPI_Intercomm_create

Purpose

Creates an intercommunicator from two intracommunicators.

C Synopsis

```
#include <mpi.h>
int MPI_Intercomm_create(MPI_Comm local_comm,int local_leader,
    MPI_Comm peer_comm,int remote_leader,int tag,MPI_Comm *newintercom);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_INTERCOMM_CREATE(INTEGER LOCAL_COMM,INTEGER LOCAL_LEADER,
    INTEGER PEER_COMM,INTEGER REMOTE_LEADER,INTEGER TAG,
    INTEGER NEWINTERCOM,INTEGER IERROR)
```

Parameters

local_comm	is the local intracommunicator (handle) (IN)
local_leader	is an integer specifying the rank of local group leader in **local_comm** (IN)
peer_comm	is the "peer" intracommunicator (significant only at the **local_leader**) (handle) (IN)
remote_leader	is the rank of remote group leader in **peer_comm** (significant only at the **local_leader**) (integer) (IN)
tag	"safe" tag (integer) (IN)
newintercom	is the new intercommunicator (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine creates an intercommunicator from two intracommunicators and is collective over the union of the local and the remote groups. Tasks should provide identical **local_comm** and **local_leader** arguments within each group. Wildcards are not permitted for **remote_leader**, **local_leader**, and **tag**.

MPI_INTERCOMM_CREATE uses point-to-point communication with communicator **peer_comm** and tag **tag** between the leaders. Make sure that there are no pending communications on **peer_comm** that could interfere with this communication. It is recommended that you use a dedicated peer communicator, such as a duplicate of MPI_COMM_WORLD, to avoid trouble with peer communicators.

Errors

Conflicting collective operations on communicator	
Invalid communicator(s)	
Invalid communicator type(s)	must be intracommunicator(s)
Invalid rank(s)	**rank** < 0 or **rank** > = groupsize
Invalid tag	**tag** < 0
MPI not initialized	
MPI already finalized	

Related Information

MPI_COMM_DUP
MPI_INTERCOMM_MERGE

MPI_INTERCOMM_MERGE, MPI_Intercomm_merge

Purpose

Creates an intracommunicator by merging the local and the remote groups of an intercommunicator.

C Synopsis

```
#include <mpi.h>
int MPI_Intercomm_merge(MPI_Comm intercomm,int high,
    MPI_Comm *newintracom);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_INTERCOMM_MERGE(INTEGER INTERCOMM,INTEGER HIGH,
      INTEGER NEWINTRACOMM,INTEGER IERROR)
```

Parameters

intercomm	is the intercommunicator (handle) (IN)
high	(logical) (IN)
newintracomm	is the new intracommunicator (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine creates an intracommunicator from the union of two groups associated with **intercomm**. Tasks should provide the same **high** value within each of the two groups. If tasks in one group provide the value **high** = **false** and tasks in the other group provide the value **high** = **true**, then the union orders the "low" group before the "high" group. If all tasks provided the same **high** argument, then the order of the union is arbitrary.

This call is blocking and collective within the union of the two groups.

Errors

Invalid communicator

Invalid communicator type must be intercommunicator

Inconsistent high within group

MPI not initialized

MPI already finalized

Related Information

MPI_INTERCOMM_CREATE

MPI_IPROBE, MPI_Iprobe

Purpose

Checks to see if a message matching *source*, *tag*, and *comm* has arrived.

C Synopsis

```
#include <mpi.h>
int MPI_Iprobe(int source,int tag,MPI_Comm comm,int *flag,
      MPI_Status *status);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_IPROBE(INTEGER SOURCE,INTEGER TAG,INTEGER COMM,INTEGER FLAG,
      INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
```

Parameters

source	is a source rank or MPI_ANY_SOURCE (integer) (IN)
tag	is a tag value or MPI_ANY_TAG (integer) (IN)
comm	is a communicator (handle) (IN)
flag	(logical) (OUT)
status	is a status object (status) (OUT). Note that in Fortran a single status object is an array of integers.
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine allows you to check for incoming messages without actually receiving them.

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

When MPI_IPROBE returns **flag** = **true**, the content of the status object can be accessed to find the source, tag and length of the probed message.

A subsequent receive executed with the same **comm**, and the source and tag returned in **status** by MPI_IPROBE receives the message that was matched by the probe, if no other intervening receive occurs after the initial probe.

source can be MPI_ANY_SOURCE and **tag** can be MPI_ANY_TAG. This allows you to probe messages from any source and/or with any tag, but you must provide a specific communicator with **comm**.

When a message is not received immediately after it is probed, the same message can be probed for several times before it is received.

Notes

In a threaded environment, MPI_PROBE or MPI_IPROBE followed by MPI_RECV, based on the information from the probe, may not be a thread-safe operation. You must ensure that no other thread received the detected message.

An MPI_IPROBE cannot prevent a message from being cancelled successfully by the sender, making it unavailable for the MPI_RECV. Structure your program so this will not occur.

Errors

Invalid source	**source** < 0 or **source** > &equals groupsize
Invalid tag	**tag** < 0
Invalid communicator	
MPI not initialized	
MPI already finalized	

Related Information

MPI_PROBE
MPI_RECV

MPI_IRECV, MPI_Irecv

Purpose

Performs a nonblocking receive operation.

C Synopsis

```
#include <mpi.h>
int MPI_Irecv(void* buf,int count,MPI_Datatype datatype,
    int source,int tag,MPI_Comm comm,MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_IRECV(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER SOURCE,
      INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Parameters

buf	is the initial address of the receive buffer (choice) (OUT)
count	is the number of elements in the receive buffer (integer) (IN)
datatype	is the datatype of each receive buffer element (handle) (IN)
source	is the rank of source or MPI_ANY_SOURCE (integer) (IN)
tag	is the message tag or MPI_ANY_TAG (integer) (IN)
comm	is the communicator (handle) (IN)
request	is the communication request (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine starts a nonblocking receive and returns a handle to a request object. You can later use the **request** to query the status of the communication or wait for it to complete.

A nonblocking receive call means the system may start writing data into the receive buffer. Once the nonblocking receive operation is called, do not access any part of the receive buffer until the receive is complete.

Notes

The message received must be less than or equal to the length of the receive buffer. If all incoming messages do not fit without truncation, an overflow error occurs. If a message arrives that is shorter than the receive buffer, then only those locations corresponding to the actual message are changed. If an overflow occurs, it is flagged at the MPI_WAIT or MPI_TEST. See MPI_RECV for additional information.

Errors

Invalid count	**count** < 0
Invalid datatype	
Type not committed	
Invalid source	**source** < 0 or **source** > = groupsize
Invalid tag	**tag** < 0
Invalid comm	
MPI not initialized	
MPI already finalized	

Related Information

MPI_RECV
MPI_RECV_INIT
MPI_WAIT

MPI_IRSEND, MPI_Irsend

Purpose

Performs a nonblocking ready mode send operation.

C Synopsis

```
#include <mpi.h>
int MPI_Irsend(void* buf,int count,MPI_Datatype datatype,
    int dest,int tag,MPI_Comm comm,MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_IRSEND(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,
      INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Parameters

buf	is the initial address of the send buffer (choice) (IN)
count	is the number of elements in the send buffer (integer) (IN)
datatype	is the datatype of each send buffer element (handle) (IN)
dest	is the rank of the destination task in **comm** (integer) (IN)
tag	is the message tag (integer) (IN)
comm	is the communicator (handle) (IN)
request	is the communication request (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_IRSEND starts a ready mode, nonblocking send. The send buffer may not be modified until the request has been completed by MPI_WAIT, MPI_TEST, or one of the other MPI wait or test functions.

Notes

See MPI_RSEND for additional information.

Errors

Invalid count	**count** < 0
Invalid datatype	
Type not committed	
Invalid destination	**dest** < 0 or **dest** > = groupsize
Invalid tag	**tag** < 0
Invalid comm	
No receive posted	error flagged at destination

MPI not initialized

MPI already finalized

Develop mode error if:

Illegal buffer update

Related Information

MPI_RSEND
MPI_RSEND_INIT
MPI_WAIT

MPI_ISEND, MPI_Isend

Purpose

Performs a nonblocking standard mode send operation.

C Synopsis

```
#include <mpi.h>
int MPI_Isend(void* buf,int count,MPI_Datatype datatype,
    int dest,int tag,MPI_Comm comm,MPI_request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_ISEND(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,
    INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Parameters

buf	is the initial address of the send buffer (choice) (IN)
count	is the number of elements in the send buffer (integer) (IN)
datatype	is the datatype of each send buffer element (handle) (IN)
dest	is the rank of the destination task in **comm** (integer) (IN)
tag	is the message tag (integer) (IN)
comm	is the communicator (handle) (IN)
request	is the communication request (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine starts a nonblocking standard mode send. The send buffer may not be modified until the request has been completed by MPI_WAIT, MPI_TEST, or one of the other MPI wait or test functions.

Notes

See MPI_SEND for additional information.

Errors

Invalid count	**count** < 0
Invalid datatype	
Type not committed	
Invalid destination	**dest** < 0 or **dest** > = groupsize
Invalid tag	**tag** < 0
Invalid comm	
MPI not initialized	

MPI already finalized

Develop mode error if:

Illegal buffer update

Related Information

MPI_SEND
MPI_SEND_INIT
MPI_WAIT

MPI_ISSEND, MPI_Issend

Purpose

Performs a nonblocking synchronous mode send operation.

C Synopsis

```
#include <mpi.h>
int MPI_Issend(void* buf,int count,MPI_Datatype datatype,
    int dest,int tag,MPI_Comm comm,MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_ISSEND(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,
    INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Parameters

buf	is the initial address of the send buffer (choice) (IN)
count	is the number of elements in the send buffer (integer) (IN)
datatype	is the datatype of each send buffer element (handle) (IN)
dest	is the rank of the destination task in **comm** (integer) (IN)
tag	is the message tag (integer) (IN)
comm	is the communicator (handle) (IN)
request	is the communication request (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_ISSEND starts a synchronous mode, nonblocking send. The send buffer may not be modified until the request has been completed by MPI_WAIT, MPI_TEST, or one of the other MPI wait or test functions.

Notes

See MPI_SSEND for additional information.

Errors

Invalid count	**count** < 0
Invalid datatype	
Type not committed	
Invalid destination	**dest** < 0 or **dest** > = groupsize
Invalid tag	**tag** < 0
Invalid comm	
MPI not initialized	

MPI already finalized

Develop mode error if:

Illegal buffer update

Related Information

MPI_SSEND
MPI_SSEND_INIT
MPI_WAIT

MPI_KEYVAL_CREATE, MPI_Keyval_create

Purpose

Generates a new attribute key.

C Synopsis

```
#include <mpi.h>
int MPI_Keyval_create(MPI_Copy_function *copy_fn,
    MPI_Delete_function *delete_fn,int *keyval,
    void* extra_state);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_KEYVAL_CREATE(EXTERNAL COPY_FN,EXTERNAL DELETE_FN,
      INTEGER KEYVAL,INTEGER EXTRA_STATE,INTEGER IERROR)
```

Parameters

copy_fn	is the copy callback function for keyval (IN)
delete_fn	is the delete callback function for keyval (IN)
keyval	is an integer specifying the key value for future access (OUT)
extra_state	is the extra state for callback functions (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine generates a new attribute key. Keys are locally unique in a task, opaque to the user, and are explicitly stored in integers. Once allocated, **keyval** can be used to associate attributes and access them on any locally defined communicator. **copy_fn** is invoked when a communicator is duplicated by MPI_COMM_DUP. It should be of type MPI_COPY_FUNCTION, which is defined as follows:

In C:

```
typedef int MPI_Copy_function (MPI_Comm oldcomm,int keyval,
        void *extra_state,void *attribute_val_in,
        void *attribute_val_out,int *flag);
```

In Fortran:

|
```
SUBROUTINE COPY_FUNCTION(INTEGER OLDCOMM,INTEGER KEYVAL,
        INTEGER EXTRA_STATE,INTEGER ATTRIBUTE_VAL_IN,
        INTEGER ATTRIBUTE_VAL_OUT,LOGICAL FLAG,INTEGER IERROR)
```

You can use the predefined functions MPI_NULL_COPY_FN and MPI_DUP_FN to never copy or to always copy, respectively.

delete_fn is invoked when a communicator is deleted by MPI_COMM_FREE or when a call is made to MPI_ATTR_DELETE. A call to MPI_ATTR_PUT that overlays a previously put attribute also causes **delete_fn** to be called. It should be defined as follows:

In C:

```
typedef int MPI_Delete_function (MPI_Comm comm,int keyval,
        void *attribute_val, void *extra_state);
```

In Fortran:

```
SUBROUTINE DELETE_FUNCTION(INTEGER COMM,INTEGER KEYVAL,
        INTEGER ATTRIBUTE_VAL,INTEGER EXTRA_STATE,
        INTEGER IERROR)
```

You can use the predefined function MPI_NULL_DELETE_FN if no special handling of attribute deletions is required.

In Fortran, the value of **extra_state** is recorded by MPI_KEYVAL_CREATE and the callback functions should not attempt to modify this value.

The MPI standard requires that when **copy_fn** or **delete_fn** gives a return code other than MPI_SUCCESS, the MPI routine in which this occurs must fail. The standard does not suggest that the **copy_fn** or **delete_fn** return code be used as the MPI routine's return value. The standard does require that an MPI return code be in the range between MPI_SUCCESS and MPI_ERR_LASTCODE. It places no range limits on **copy_fn** or **delete_fn** return codes. For this reason, we provide a specific error code for a **copy_fn** failure and another for a **delete_fn** failure. These error codes can be found in error class MPI_ERR_OTHER. The **copy_fn** or the **delete_fn** return code is not preserved.

Errors

MPI not initialized

MPI already finalized

Related Information

MPI_ATTR_PUT
MPI_ATTR_DELETE
MPI_COMM_DUP
MPI_COMM_FREE

MPI_KEYVAL_FREE, MPI_Keyval_free

Purpose

Marks an attribute key for deallocation.

C Synopsis

```
#include <mpi.h>
int MPI_Keyval_free(int *keyval);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_KEYVAL_FREE(INTEGER KEYVAL,INTEGER IERROR)
```

Parameters

keyval	attribute key (integer) (INOUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine sets **keyval** to MPI_KEYVAL_INVALID and marks the attribute key for deallocation. You can free an attribute key that is in use because the actual deallocation occurs only when all active references to it are complete. These references, however, need to be explicitly freed. Use calls to MPI_ATTR_DELETE to free one attribute instance. To free all attribute instances associated with a communicator, use MPI_COMM_FREE.

Errors

Invalid attribute key	attribute key is undefined
Predefined attribute key	attribute key is predefined
MPI not initialized	
MPI already finalized	

Related Information

MPI_ATTR_DELETE
MPI_COMM_FREE

MPI_OP_CREATE, MPI_Op_create

Purpose

Binds a user-defined reduction operation to an **op** handle.

C Synopsis

```
#include <mpi.h>
int MPI_Op_create(MPI_User_function *function,int commute,
      MPI_Op *op);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_OP_CREATE(EXTERNAL FUNCTION,INTEGER COMMUTE,INTEGER OP,
      INTEGER IERROR)
```

Parameters

function	is the user-defined reduction function (function) (IN)
commute	is **true** if commutative; otherwise it's **false** (IN)
op	is the reduction operation (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine binds a user-defined reduction operation to an **op** handle which you can then use in MPI_REDUCE, MPI_ALLREDUCE, MPI_REDUCE_SCATTER and MPI_SCAN and their nonblocking equivalents.

The user-defined operation is assumed to be associative. If **commute** = **true**, then the operation must be both commutative and associative. If **commute** = **false**, then the order of the operation is fixed. The order is defined in ascending, task rank order and begins with task zero.

function is user-defined function. It must have the following four arguments: **invec**, **inoutvec**, **len**, and **datatype**.

The following is the ANSI-C prototype for the function:

```
typedef void MPI_User_function(void *invec, void *inoutvec,
    int *len, MPI_Datatype *datatype);
```

The following is the Fortran declaration for the function:

```
SUBROUTINE USER_FUNCTION(INVEC(*), INOUTVEC(*), LEN, TYPE)
<type> INVEC(LEN), INOUTVEC(LEN)
 INTEGER LEN, TYPE
```

Notes

See Appendix D, “Reduction Operations” on page 355 for information about reduction functions.

Errors

Null function

MPI not initialized

MPI already finalized

Related Information

MPI_OP_FREE
MPI_REDUCE
MPI_ALLREDUCE
MPI_REDUCE_SCATTER
MPI_SCAN

MPI_OP_FREE, MPI_Op_free

Purpose

Marks a user-defined reduction operation for deallocation.

C Synopsis

```
#include <mpi.h>
int MPI_Op_free(MPI_Op *op);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_OP_FREE(INTEGER OP,INTEGER IERROR)
```

Parameters

op	is the reduction operation (handle) (INOUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This function marks a reduction operation for deallocation, and set **op** to MPI_OP_NULL. Actual deallocation occurs when the operation's reference count is zero.

Errors

Invalid operation

Predefined operation

MPI not initialized

MPI already finalized

Related Information

MPI_OP_CREATE

MPI_PACK, MPI_Pack

Purpose

Packs the message in the specified send buffer into the specified buffer space.

C Synopsis

```
#include <mpi.h>
int MPI_Pack(void* inbuf,int incount,MPI_Datatype datatype,
    void *outbuf,int outsize,int *position,MPI_Comm comm);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_PACK(CHOICE INBUF,INTEGER INCOUNT,INTEGER DATATYPE,
    CHOICE OUTBUF,INTEGER OUTSIZE,INTEGER POSITION,INTEGER COMM
    INTEGER IERROR)
```

Parameters

inbuf	is the input buffer start (choice) (IN)
incount	is an integer specifying the number of input data items (IN)
datatype	is the datatype of each input data item (handle) (IN)
outbuf	is the output buffer start (choice) (OUT)
outsize	is an integer specifying the output buffer size in bytes (OUT)
position	is the current position in the output buffer counted in bytes (integer) (INOUT)
comm	is the communicator for sending the packed message (handle) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine packs the message specified by **inbuf**, **incount**, and **datatype** into the buffer space specified by **outbuf** and **outsize**. The input buffer is any communication buffer allowed in MPI_SEND. The output buffer is any contiguous storage space containing **outsize** bytes and starting at the address **outbuf**.

The input value of **position** is the beginning offset in the output buffer that will be used for packing. The output value of **position** is the offset in the output buffer following the locations occupied by the packed message. **comm** is the communicator that will be used for sending the packed message.

Errors

Invalid incount	**incount** < 0
Invalid datatype	
Type not committed	

Invalid communicator

Outbuf too small

MPI not initialized

MPI already finalized

Related Information

MPI_UNPACK
MPI_PACK_SIZE

MPI_PACK_SIZE, MPI_Pack_size

Purpose

Returns the number of bytes required to hold the data.

C Synopsis

```
#include <mpi.h>
int MPI_Pack_size(int incount,MPI_Datatype datatype,
   MPI_Comm comm, int *size);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_PACK_SIZE(INTEGER INCOUNT,INTEGER DATATYPE,INTEGER COMM,
INTEGER SIZE,INTEGER IERROR)
```

Parameters

incount	is an integer specifying the count argument to a packing call (IN)
datatype	is the datatype argument to a packing call (handle) (IN)
comm	is the communicator to a packing call (handle) (IN)
size	size of packed message in bytes (integer) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine returns the number of bytes required to pack **incount** replications of the datatype. You can use MPI_PACK_SIZE to determine the size required for a packing buffer or to track space needed for buffered sends.

Errors

Invalid datatype

Type is not committed

MPI not initialized

MPI already finalized

Invalid communicator

Invalid incount	**incount** < 0

Related Information

MPI_PACK

MPI_PCONTROL, MPI_Pcontrol

Purpose

Provides profiler control.

C Synopsis

```
#include <mpi.h>
int MPI_Pcontrol(const int level, ...);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_PCONTROL(INTEGER LEVEL, ...)
```

Parameters

level	is the profiling level (IN) The proper values for **level** and the meanings of those values are determined by the profiler being used.
...	0 or more parameters
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_PCONTROL is a placeholder to allow applications to run with or without an independent profiling package without modification. MPI implementations do not use this routine and do not have any control of the implementation of the profiling code.

Calls to this routine allow a profiling package to be controlled from MPI programs. The nature of control and the arguments required are determined by the profiling package. The MPI library routine by this name returns to the caller without any action.

Notes

For each additional call level introduced by the profiling code, the global variable VT_instaddr_depth needs to be incremented so the Visualization Tool Tracing Subsystem(VT) can record where the application called the MPI message passing library routine. The VT_instaddr_depth variable is defined in /usr/lpp/ppe.vt/include/VT_mpi.h.

Errors

MPI does not report any errors for MPI_PCONTROL.

MPI_PROBE, MPI_Probe

Purpose

Waits until a message matching *source*, *tag*, and *comm* arrives.

C Synopsis

```
#include <mpi.h>
int MPI_Probe(int source,int tag,MPI_Comm comm,MPI_Status *status);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_PROBE(INTEGER SOURCE,INTEGER TAG,INTEGER COMM,
    INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
```

Parameters

source	is a source rank or MPI_ANY_SOURCE (integer) (IN)
tag	is a source tag or MPI_ANY_TAG (integer) (IN)
comm	is a communicator (handle) (IN)
status	is a status object (status) (OUT). Note that in Fortran a single status object is an array of integers.
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_PROBE behaves like MPI_IPROBE. It allows you to check for an incoming message without actually receiving it. MPI_PROBE is different in that it is a blocking call that returns only after a matching message has been found.

Notes

In a threaded environment, MPI_PROBE or MPI_IPROBE followed by MPI_RECV, based on the information from the probe, may not be a thread-safe operation. You must ensure that no other thread received the detected message.

An MPI_IPROBE cannot prevent a message from being cancelled successfully by the sender, making it unavailable for the MPI_RECV. Structure your program so this will not occur.

Errors

Invalid source	**source** < 0 or **source** > = groupsize
Invalid tag	**tag** < 0
Invalid communicator	
MPI not initialized	
MPI already finalized	

Related Information

MPI_IPROBE
MPI_RECV

MPI_RECV, MPI_Recv

Purpose

Performs a blocking receive operation.

C Synopsis

```
#include <mpi.h>
int MPI_Recv(void* buf,int count,MPI_Datatype datatype,
    int source,int tag,MPI_Comm comm,MPI_Status *status);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_RECV(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER SOURCE,
      INTEGER TAG,INTEGER COMM,INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
```

Parameters

buf	is the initial address of the receive buffer (choice) (OUT)
count	is the number of elements to be received (integer) (IN)
datatype	is the datatype of each receive buffer element (handle) (IN)
source	is the rank of the source task in **comm** or MPI_ANY_SOURCE (integer) (IN)
tag	is the message tag or MPI_ANY_TAG (integer) (IN)
comm	is the communicator (handle) (IN)
status	is the status object (status) (OUT). Note that in Fortran a single status object is an array of integers.
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_RECV is a blocking receive. The receive buffer is storage containing room for **count** consecutive elements of the type specified by **datatype**, starting at address **buf**.

The message received must be less than or equal to the length of the receive buffer. If all incoming messages do not fit without truncation, an overflow error occurs. If a message arrives that is shorter than the receive buffer, then only those locations corresponding to the actual message are changed.

Errors

Invalid count	**count** < 0
Invalid datatype	
Type not committed	
Invalid source	**source** < 0 or **source** > = groupsize
Invalid tag	**tag** < 0

Invalid comm

Truncation occurred

MPI not initialized

MPI already finalized

Related Information

MPI_IRECV
MPI_SENDRECV
MPI_SEND

MPI_RECV_INIT, MPI_Recv_init

Purpose

Creates a persistent receive request.

C Synopsis

```
#include <mpi.h>
int MPI_Recv_init(void* buf,int count,MPI_Datatype datatype,
    int source,int tag,MPI_Comm comm,MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_RECV_INIT(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,
      INTEGER SOURCE,INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Parameters

buf	is the initial address of the receive buffer (choice) (OUT)
count	is the number of elements to be received (integer) (IN)
datatype	is the type of each element (handle) (IN)
source	is the rank of source or MPI_ANY_SOURCE (integer) (IN)
tag	is the tag or MPI_ANY_TAG (integer) (IN)
comm	is the communicator (handle) (IN)
request	is the communication request (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine creates a persistent communication request for a receive operation. The argument **buf** is marked as OUT because the user gives permission to write to the receive buffer by passing the argument to MPI_RECV_INIT.

A persistent communication request is inactive after it is created. No active communication is attached to the request.

A send or receive communication using a persistent request is initiated by the function MPI_START.

Notes

See MPI_RECV for additional information.

Errors

Invalid count	**count** < 0
Invalid datatype	
Type not committed	
Invalid source	**source** < 0 or **source** > = groupsize
Invalid tag	**tag** < 0
Invalid comm	
MPI not initialized	
MPI already finalized	

Related Information

MPI_START
MPI_IRECV

MPI_REDUCE, MPI_Reduce

Purpose

Applies a reduction operation to the vector **sendbuf** over the set of tasks specified by **comm** and places the result in **recvbuf** on **root**.

C Synopsis

```
#include <mpi.h>
int MPI_Reduce(void* sendbuf,void* recvbuf,int count,
    MPI_Datatype datatype,MPI_Op op,int root,MPI_Comm comm);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_REDUCE(CHOICE SENDBUF,CHOICE RECVBUF,INTEGER COUNT,
    INTEGER DATATYPE,INTEGER OP,INTEGER ROOT,INTEGER COMM,
    INTEGER IERROR)
```

Parameters

sendbuf	is the address of the send buffer (choice) (IN)
recvbuf	is the address of the receive buffer (choice, significant only at root) (OUT)
count	is the number of elements in the send buffer (integer) (IN)
datatype	is the datatype of elements of the send buffer (handle) (IN)
op	is the reduction operation (handle) (IN)
root	is the rank of the root task (integer) (IN)
comm	is the communicator (handle) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine applies a reduction operation to the vector **sendbuf** over the set of tasks specified by **comm** and places the result in **recvbuf** on **root**.

Both the input and output buffers have the same number of elements with the same type. The arguments **sendbuf**, **count**, and **datatype** define the send or input buffer and **recvbuf**, **count** and **datatype** define the output buffer. MPI_REDUCE is called by all group members using the same arguments for **count**, **datatype**, **op**, and **root**. If a sequence of elements is provided to a task, then the reduce operation is executed element-wise on each entry of the sequence. Here's an example. If the operation is MPI_MAX and the send buffer contains two elements that are floating point numbers (**count** = 2 and **datatype** = MPI_FLOAT), then **recvbuf**(**1**) = global max(**sendbuf**(**1**)) and **recvbuf** (**2**) = global max(**sendbuf**(**2**)).

Users may define their own operations or use the predefined operations provided by MPI. User defined operations can be overloaded to operate on several datatypes, either basic or derived. A list of the MPI predefined operations is in this manual. Refer to Appendix D, "Reduction Operations" on page 355.

The argument **datatype** of MPI_REDUCE must be compatible with **op**. For a list of predefined operations refer to Appendix I, “Predefined Datatypes” on page 435.

When you use this routine in a threaded application, make sure all collective operations on a particular communicator occur in the same order at each task. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Notes

See Appendix D, “Reduction Operations” on page 355.

Errors

Invalid count	**count** < 0
Invalid datatype	
Type not committed	
Invalid op	
Invalid root	**root** < 0 or **root** > = groupsize
Invalid communicator	
Invalid communicator type	must be intracommunicator
Unequal message lengths	
MPI not initialized	
MPI already finalized	

Develop mode error if:

Inconsistent op

Inconsistent datatype

Inconsistent root

Inconsistent message length

Related Information

MPE_IREDUCE
MPI_ALLREDUCE
MPI_REDUCE_SCATTER
MPI_SCAN
MPI_OP_CREATE

MPI_REDUCE_SCATTER, MPI_Reduce_scatter

Purpose

Applies a reduction operation to the vector **sendbuf** over the set of tasks specified by **comm** and scatters the result according to the values in **recvcounts**.

C Synopsis

```
#include <mpi.h>
int MPI_Reduce_scatter(void* sendbuf,void* recvbuf,int *recvcounts,
    MPI_Datatype datatype,MPI_Op op,MPI_Comm comm);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_REDUCE_SCATTER(CHOICE SENDBUF,CHOICE RECVBUF,
    INTEGER RECVCOUNTS(*),INTEGER DATATYPE,INTEGER OP,
    INTEGER COMM,INTEGER IERROR)
```

Parameters

sendbuf	is the starting address of the send buffer (choice) (IN)
recvbuf	is the starting address of the receive buffer (choice) (OUT)
recvcounts	integer array specifying the number of elements in result distributed to each task. Must be identical on all calling tasks. (IN)
datatype	is the datatype of elements in the input buffer (handle) (IN)
op	is the reduction operation (handle) (IN)
comm	is the communicator (handle) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_REDUCE_SCATTER first performs an element-wise reduction on vector of **count** = Σ **i recvcounts[i]** elements in the send buffer defined by **sendbuf**, **count** and **datatype**. Next, the resulting vector is split into **n** disjoint segments, where **n** is the number of members in the group. Segment **i** contains **recvcounts[i]** elements. The **i**th segment is sent to task **i** and stored in the receive buffer defined by **recvbuf**, **recvcounts[i]** and **datatype**.

Notes

MPI_REDUCE_SCATTER is functionally equivalent to MPI_REDUCE with **count** equal to the sum of **recvcounts[i]** followed by MPI_SCATTERV with **sendcounts** equal to **recvcounts**. When you use this routine in a threaded application, make sure all collective operations on a particular communicator occur in the same order at each task. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid recvcounts	**recvcounts[i]** < 0
Invalid datatype	
Type not committed	
Invalid op	
Invalid communicator	
Invalid communicator type	must be intracommunicator
Unequal message lengths	
MPI not initialized	
MPI already finalized	

Develop mode error if:

Inconsistent op

Inconsistent datatype

Related Information

MPE_IREDUCE_SCATTER
MPI_REDUCE
MPI_OP_CREATE

MPI_REQUEST_FREE, MPI_Request_free

Purpose

Marks a request for deallocation.

C Synopsis

```
#include <mpi.h>
int MPI_Request_free(int MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_REQUEST_FREE(INTEGER REQUEST,INTEGER IERROR)
```

Parameters

request	is a communication request (handle) (INOUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine marks a request object for deallocation and sets **request** to MPI_REQUEST_NULL. An ongoing communication associated with the request is allowed to complete before deallocation occurs.

Notes

This function marks a communication request as **free**. Actual deallocation occurs when the **request** is complete. Active receive requests and collective communication requests cannot be freed.

Errors

Invalid request

Attempt to free receive request

Attempt to free CCL request

MPI not initialized

MPI already finalized

Related Information

MPI_WAIT

MPI_RSEND, MPI_Rsend

Purpose

Performs a blocking ready mode send operation.

C Synopsis

```
#include <mpi.h>
int MPI_Rsend(void* buf,int count,MPI_Datatype datatype,
    int dest,int tag,MPI_Comm comm);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_RSEND(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,
      INTEGER TAG,INTEGER COMM,INTEGER IERROR)
```

Parameters

buf	is the initial address of the send buffer (choice) (IN)
count	is the number of elements in the send buffer (integer) (IN)
datatype	is the datatype of each send buffer element (handle) (IN)
dest	is the rank of destination (integer) (IN)
tag	is the message tag (integer) (IN)
comm	is the communicator (handle) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine is a blocking ready mode send. It can be started only when a matching receive is posted. If a matching receive is not posted, the operation is erroneous and its outcome is undefined.

The completion of MPI_RSEND indicates that the send buffer can be reused.

Notes

A ready send for which no receive exists produces an asynchronous error at the destination. The error is not detected at the MPI_RSEND and it returns MPI_SUCCESS.

Errors

Invalid count	**count** < 0
Invalid datatype	
Type not committed	
Invalid destination	**dest** < 0 or **dest** > = groupsize
Invalid tag	**tag** < 0
Invalid comm	

No receive posted	error flagged at destination
MPI not initialized	
MPI already finalized	

Related Information

MPI_IRSEND
MPI_SEND

MPI_RSEND_INIT, MPI_Rsend_init

Purpose

Creates a persistent ready mode send request.

C Synopsis

```
#include <mpi.h>
int MPI_Rsend_init(void* buf,int count,MPI_Datatype datatype,
      int dest,int tag,MPI_Comm comm,MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_RSEND_INIT(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,
      INTEGER DEST,INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Parameters

buf	is the initial address of the send buffer (choice) (IN)
count	is the number of elements to be sent (integer) (IN)
datatype	is the type of each element (handle) (IN)
dest	is the rank of the destination task (integer) (IN)
tag	is the message tag (integer) (IN)
comm	is the communicator (handle) (IN)
request	is the communication request (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_RSEND_INIT creates a persistent communication object for a ready mode send operation. MPI_START or MPI_STARTALL is used to activate the send.

Notes

See MPI_RSEND for additional information.

Errors

Invalid count	**count** < 0
Invalid datatype	
Type not committed	
Invalid destination	**dest** < 0 or **dest** > = groupsize
Invalid tag	**tag** < 0
Invalid comm	
MPI not initialized	
MPI already finalized	

Related Information

MPI_START
MPI_IRSEND

MPI_SCAN, MPI_Scan

Purpose

Performs a parallel prefix reduction on data distributed across a group.

C Synopsis

```
#include <mpi.h>
int MPI_Scan(void* sendbuf,void* recvbuf,int count,
    MPI_Datatype datatype,MPI_Op op,MPI_Comm comm);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_SCAN(CHOICE SENDBUF,CHOICE RECVBUF,INTEGER COUNT,
    INTEGER DATATYPE,INTEGER OP,INTEGER COMM,INTEGER IERROR)
```

Parameters

sendbuf	is the starting address of the send buffer (choice) (IN)
recvbuf	is the starting address of the receive buffer (choice) (OUT)
count	is the number of elements in **sendbuf** (integer) (IN)
datatype	is the datatype of elements in **sendbuf** (handle) (IN)
op	is the reduction operation (handle) (IN)
comm	is the communicator (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_SCAN is used to perform a prefix reduction on data distributed across the group. The operation returns, in the receive buffer of the task with rank **i**, the reduction of the values in the send buffers of tasks with ranks 0, ..., **i** (inclusive). The type of operations supported, their semantics, and the restrictions on send and receive buffers are the same as for MPI_REDUCE.

When you use this routine in a threaded application, make sure all collective operations on a particular communicator occur in the same order at each task. See Appendix G, "Programming Considerations for User Applications in POE" on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid count	**count** < 0
Invalid datatype	
Type not committed	
Invalid op	
Invalid communicator	

Invalid communicator type must be intracommunicator

Unequal message lengths

MPI not initialized

MPI already finalized

Develop mode error if:

Inconsistent op

Inconsistent datatype

Inconsistent message length

Related Information

MPE_ISCAN
MPI_REDUCE
MPI_OP_CREATE

MPI_SCATTER, MPI_Scatter

Purpose

Distributes individual messages from **root** to each task in **comm**.

C Synopsis

```
#include <mpi.h>
int MPI_Scatter(void* sendbuf,int sendcount,MPI_Datatype sendtype,
    void* recvbuf,int recvcount,MPI_Datatype recvtype,int root,
    MPI_Comm comm);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_SCATTER(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,
    CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER ROOT,
    INTEGER COMM,INTEGER IERROR)
```

Parameters

sendbuf	is the address of the send buffer (choice, significant only at **root**) (IN)
sendcount	is the number of elements to be sent to each task (integer, significant only at **root**) (IN)
sendtype	is the datatype of the send buffer elements (handle, significant only at **root**) (IN)
recvbuf	is the address of the receive buffer (choice) (OUT)
recvcount	is the number of elements in the receive buffer (integer) (IN)
recvtype	is the datatype of the receive buffer elements (handle) (IN)
root	is the rank of the sending task (integer) (IN)
comm	is the communicator (handle) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_SCATTER distributes individual messages from **root** to each task in **comm**. This routine is the inverse operation to MPI_GATHER.

The type signature associated with **sendcount**, **sendtype** at the root must be equal to the type signature associated with **recvcount**, **recvtype** at all tasks. (Type maps can be different.) This means the amount of data sent must be equal to the amount of data received, pairwise between each task and the root. Distinct type maps between sender and receiver are allowed.

The following is information regarding MPI_SCATTER arguments and tasks:

- On the task **root**, all arguments to the function are significant.

- On other tasks, only the arguments **recvbuf, recvcount, recvtype, root,** and **comm** are significant.
- The argument **root** must be the same on all tasks.

A call where the specification of counts and types causes any location on the root to be read more than once is erroneous.

When you use this routine in a threaded application, make sure all collective operations on a particular communicator occur in the same order at each task. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid communicator	
Invalid communicator type	must be intracommunicator
Invalid count(s)	**count** < 0
Invalid datatype(s)	
Type not committed	
Invalid root	(**root** < 0 or **root** >= groupsize)
Unequal message lengths	
MPI not initialized	
MPI already finalized	

Develop mode error if:

Inconsistent root

Inconsistent message length

Related Information

MPE_ISCATTER
MPI_SCATTER
MPI_GATHER

MPI_SCATTERV, MPI_Scatterv

Purpose

Distributes individual messages from **root** to each task in **comm**. Messages can have different sizes and displacements.

C Synopsis

```
#include <mpi.h>
int MPI_Scatterv(void* sendbuf,int *sendcounts,
    int *displs,MPI_Datatype sendtype,void* recvbuf,
    int recvcount,MPI_Datatype recvtype,int root,
    MPI_Comm comm);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_SCATTERV(CHOICE SENDBUF,INTEGER SENDCOUNTS(*),INTEGER DISPLS(*),
    INTEGER SENDTYPE,CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,
    INTEGER ROOT,INTEGER COMM,INTEGER IERROR)
```

Parameters

sendbuf	is the address of the send buffer (choice, significant only at **root**) (IN)
sendcounts	integer array (of length group size) that contains the number of elements to send to each task (significant only at **root**) (IN)
displs	integer array (of length group size). Entry **i** specifies the displacement relative to **sendbuf** from which to send the outgoing data to task **i** (significant only at **root**) (IN)
sendtype	is the datatype of the send buffer elements (handle, significant only at **root**) (IN)
recvbuf	is the address of the receive buffer (choice) (OUT)
recvcount	is the number of elements in the receive buffer (integer) (IN)
recvtype	is the datatype of the receive buffer elements (handle) (IN)
root	is the rank of the sending task (integer) (IN)
comm	is the communicator (handle) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine distributes individual messages from **root** to each task in **comm**. Messages can have different sizes and displacements.

With **sendcounts** as an array, messages can have varying sizes of data that can be sent to each task. **displs** allows you the flexibility of where the data can be taken from on the **root**.

The type signature of **sendcount[i]**, **sendtype** at the **root** must be equal to the type signature of **recvcount**, **recvtype** at task **i**. (The type maps can be different.) This means the amount of data sent must be equal to the amount of data received, pairwise between each task and the **root**. Distinct type maps between sender and receiver are allowed.

The following is information regarding MPI_SCATTERV arguments and tasks:

- On the task **root**, all arguments to the function are significant.
- On other tasks, only the arguments **recvbuf, recvcount, recvtype, root,** and **comm** are significant.
- The argument **root** must be the same on all tasks.

A call where the specification of sizes, types and displacements causes any location on the root to be read more than once is erroneous.

When you use this routine in a threaded application, make sure all collective operations on a particular communicator occur in the same order at each task. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid communicator	
Invalid communicator type	must be intracommunicator
Invalid count(s)	**count** < 0
Invalid datatype(s)	
Type not committed	
Invalid root	**root** < 0 or **root** >= groupsize
Unequal message lengths	
MPI not initialized	
MPI already finalized	

Develop mode error if:

Inconsistent root

Related Information

MPI_SCATTER
MPI_GATHER

MPI_SEND, MPI_Send

Purpose

Performs a blocking standard mode send operation.

C Synopsis

```
#include <mpi.h>
int MPI_Send(void* buf,int count,MPI_Datatype datatype,
    int dest,int tag,MPI_Comm comm);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_SEND(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,
      INTEGER TAG,INTEGER COMM,INTEGER IERROR)
```

Parameters

buf	is the initial address of the send buffer (choice) (IN)
count	is the number of elements in the send buffer (non-negative integer) (IN)
datatype	is the datatype of each send buffer element (handle) (IN)
dest	is the rank of the destination task in **comm**(integer) (IN)
tag	is the message tag (integer) (IN)
comm	is the communicator (handle) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine is a blocking standard mode send. MPI_SEND causes **count** elements of type **datatype** to be sent from **buf** to the task specified by **dest**. **dest** is a task rank which can be any value from 0 to n–1, where n is the number of tasks in **comm**.

Errors

Invalid count	**count** < 0
Invalid datatype	
Type not committed	
Invalid destination	**dest** < 0 or **dest** > = groupsize
Invalid tag	**tag** < 0
Invalid comm	
MPI not initialized	
MPI already finalized	

Related Information

MPI_ISEND
MPI_BSEND
MPI_SSEND
MPI_RSEND
MPI_SENDRECV

MPI_SEND_INIT, MPI_Send_init

Purpose

Creates a persistent standard mode send request.

C Synopsis

```
#include <mpi.h>
int MPI_Send_init(void* buf,int count,MPI_Datatype datatype,
    int dest,int tag,MPI_Comm comm,MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_SEND_INIT(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,
      INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Parameters

buf	is the initial address of the send buffer (choice) (IN)
count	is the number of elements to be sent (integer) (IN)
datatype	is the type of each element (handle) (IN)
dest	is the rank of the destination task (integer) (IN)
tag	is the message tag (integer) (IN)
comm	is the communicator (handle) (IN)
request	is the communication request (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine creates a persistent communication request for a standard mode send operation, and binds to it all arguments of a send operation. MPI_START or MPI_STARTALL is used to activate the send.

Notes

See MPI_SEND for additional information.

Errors

Invalid count	**count** < 0
Invalid datatype	
Type not committed	
Invalid destination	**dest** < 0 or **dest** > = groupsize
Invalid tag	**tag** < 0
Invalid comm	
MPI not initialized	

MPI already finalized

Related Information

MPI_START
MPI_ISEND

MPI_SENDRECV, MPI_Sendrecv

Purpose

Performs a blocking send and receive operation.

C Synopsis

```
#include <mpi.h>
int MPI_Sendrecv(void* sendbuf,int sendcount,MPI_Datatype sendtype,
     int dest,int sendtag,void *recvbuf,int recvcount,MPI_Datatype recvtype,
     int source,int recvtag,MPI_Comm comm,MPI_Status *status);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_SENDRECV(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,
      INTEGER DEST,INTEGER SENDTAG,CHOICE RECVBUF,INTEGER RECVCOUNT,
      INTEGER RECVTYPE,INTEGER SOURCE,INTEGER RECVTAG,INTEGER COMM,
      INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
```

Parameters

sendbuf	is the initial address of the send buffer (choice) (IN)
sendcount	is the number of elements to be sent (integer) (IN)
sendtype	is the type of elements in the send buffer (handle) (IN)
dest	is the rank of the destination task (integer) (IN)
sendtag	is the send tag (integer) (IN)
recvbuf	is the initial address of the receive buffer (choice) (OUT)
recvcount	is the number of elements to be received (integer) (IN)
recvtype	is the type of elements in the receive buffer (handle) (IN)
source	is the rank of the source task or MPI_ANY_SOURCE (integer) (IN)
recvtag	is the receive tag or MPI_ANY_TAG (integer) (IN)
comm	is the communicator (handle) (IN)
status	is the status object (status) (OUT). Note that in Fortran a single status object is an array of integers.
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine is a blocking send and receive operation. Send and receive use the same communicator but can use different tags. The send and the receive buffers must be disjoint and can have different lengths and datatypes.

Errors

Invalid count(s)	**count** < 0
Invalid datatype(s)	
Type not committed	
Invalid destination	**dest** < 0 or **dest** > = groupsize
Invalid source	**source** < 0 or **source** > = groupsize
Invalid communicator	
Invalid tag(s)	**tag** < 0
MPI not initialized	
MPI already finalized	

Related Information

MPI_SENDRECV_REPLACE
MPI_SEND
MPI_RECV

MPI_SENDRECV_REPLACE, MPI_Sendrecv_replace

Purpose

Performs a blocking send and receive operation using a common buffer.

C Synopsis

```
#include <mpi.h>
int MPI_Sendrecv_replace(void* buf,int count,MPI_Datatype datatype,
     int dest,int sendtag,int source,int recvtag,MPI_Comm comm,
     MPI_Status *status);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_SENDRECV_REPLACE(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,
      INTEGER DEST,INTEGER SENDTAG,INTEGER SOURCE,INTEGER RECVTAG,
      INTEGER COMM,INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
```

Parameters

buf	is the initial address of the send and receive buffer (choice) (INOUT)
count	is the number of elements to be sent and received (integer) (IN)
datatype	is the type of elements in the send and receive buffer (handle) (IN)
dest	is the rank of the destination task (integer) (IN)
sendtag	is the send message tag (integer) (IN)
source	is the rank of the source task or MPI_ANY_SOURCE (integer) (IN)
recvtag	is the receive message tag or MPI_ANY_TAGE (integer) (IN)
comm	is the communicator (handle) (IN)
status	is the status object (status) (OUT). Note that in Fortran a single status object is an array of integers.
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine is a blocking send and receive operation using a common buffer. Send and receive use the same buffer so the message sent is replaced with the message received.

Errors

Invalid count	**count** < 0
Invalid datatype	
Type not committed	

Invalid destination	**dest** < 0 or **dest** > = groupsize
Invalid source	**source** < 0 or **source** > = groupsize
Invalid communicator	
Invalid tag(s)	**tag** < 0
Out of memory	
MPI not initialized	
MPI already finalized	

Related Information

MPI_SENDRECV

MPI_SSEND, MPI_Ssend

Purpose

Performs a blocking synchronous mode send operation.

C Synopsis

```
#include <mpi.h>
int MPI_Ssend(void* buf,int count,MPI_Datatype datatype,
    int dest,int tag,MPI_Comm comm);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_SSEND(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,
    INTEGER TAG,INTEGER COMM,INTEGER IERROR)
```

Parameters

buf	is the initial address of the send buffer (choice) (IN)
count	is the number of elements in the send buffer (integer) (IN)
datatype	is the datatype of each send buffer element (handle) (IN)
dest	is the rank of the destination task (integer) (IN)
tag	is the message tag (integer) (IN)
comm	is the communicator (handle) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine is a blocking synchronous mode send. This a non-local operation. It can be started whether or not a matching receive was posted. However, the send will complete only when a matching receive is posted and the receive operation has started to receive the message sent by MPI_SSEND.

The completion of MPI_SSEND indicates that the send buffer is freed and also that the receiver has started executing the matching receive. If both sends and receives are blocking operations, the synchronous mode provides synchronous communication.

Errors

Invalid count	**count** < 0
Invalid datatype	
Type not committed	
Invalid destination	**dest** < 0 or **dest** > = groupsize
Invalid tag	**tag** < 0
Invalid comm	
MPI not initialized	

MPI already finalized

Related Information

MPI_ISSEND
MPI_SEND

MPI_SSEND_INIT, MPI_Ssend_init

Purpose

Creates a persistent synchronous mode send request.

C Synopsis

```
#include <mpi.h>
int MPI_Ssend_init(void* buf,int count,MPI_Datatype datatype,
    int dest,int tag,MPI_Comm comm,MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_SSEND_INIT(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST

      INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Parameters

buf	is the initial address of the send buffer (choice) (IN)
count	is the number of elements to be sent (integer) (IN)
datatype	is the type of each element (handle) (IN)
dest	is the rank of the destination task (integer) (IN)
tag	is the message tag (integer) (IN)
comm	is the communicator (handle) (IN)
request	is the communication request (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine creates a persistent communication object for a synchronous mode send operation. MPI_START or MPI_STARTALL can be used to activate the send.

Notes

See MPI_SSEND for additional information.

Errors

Invalid count	**count** < 0
Invalid datatype	
Type not committed	
Invalid destination	**dest** < 0 or **dest** > = groupsize
Invalid tag	**tag** < 0
Invalid comm	
MPI not initialized	
MPI already finalized	

Related Information

MPI_START
MPI_ISSEND

MPI_START, MPI_Start

Purpose

Activates a persistent request operation.

C Synopsis

```
#include <mpi.h>
int MPI_Start(MPI_Request *request);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_START(INTEGER REQUEST,INTEGER IERROR)
```

Parameters

request	is a communication request (handle) (INOUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_START activates a persistent request operation. **request** is a handle returned by MPI_RECV_INIT, MPI_RSEND_INIT, MPI_SSEND_INIT, MPI_BSEND_INIT or MPI_SEND_INIT. Once the call is made, do not access the communication buffer until the operation completes.

If the request is for a send with ready mode, then a matching receive must be posted before the call is made. If the request is for a buffered send, adequate buffer space must be available.

Errors

Invalid request

Request not persistent

Request already active

Insufficient buffer space only if buffered send

MPI not initialized

MPI already finalized

Related Information

MPI_STARTALL
MPI_SEND_INIT
MPI_BSEND_INIT
MPI_RSEND_INIT
MPI_SSEND_INIT
MPI_RECV_INIT

MPI_STARTALL, MPI_Startall

Purpose

Activates a collection of persistent request operations.

C Synopsis

```
#include <mpi.h>
int MPI_Startall(int count,MPI_request *array_of_requests);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_STARTALL(INTEGER COUNT,INTEGER ARRAY_OF_REQUESTS(*),INTEGER IERROR)
```

Parameters

count	is the list length (integer) (IN)
array_of_requests	is the array of requests (array of handle) (INOUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_STARTALL starts all communications associated with request operations in **array_of_requests**.

A communication started with MPI_STARTALL is completed by a call to one of the MPI wait or test operations. The request becomes inactive after successful completion but is not deallocated and can be reactivated by an MPI_STARTALL. If a request is for a send with ready mode, then a matching receive must be posted before the call. If a request is for a buffered send, adequate buffer space must be available.

Errors

Invalid count

Invalid request array

Request(s) invalid

Request(s) not persistent

Request(s) active

Insufficient buffer space only if a buffered send

MPI not initialized

MPI already finalized

Related Information

MPI_START

MPI_TEST, MPI_Test

Purpose

Checks to see if a nonblocking request has completed.

C Synopsis

```
#include <mpi.h>
int MPI_Test(MPI_Request *request,int *flag,MPI_Status *status);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_TEST(INTEGER REQUEST,INTEGER FLAG,INTEGER STATUS(MPI_STATUS_SIZE),
    INTEGER IERROR)
```

Parameters

request	is the operation request (handle) (INOUT)
flag	*true* if operation completed (logical) (OUT)
status	status object (status) (OUT). Note that in Fortran a single status object is an array of integers.
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_TEST returns **flag** = **true** if the operation identified by **request** is complete. The status object is set to contain information on the completed operation. The request object is deallocated and the **request** handle is set to MPI_REQUEST_NULL. Otherwise, **flag** = **false** and the status object is undefined. MPI_TEST is a local operation. The status object can be queried for information about the operation. (See MPI_WAIT.)

You can call MPI_TEST with a null or inactive **request** argument. The operation returns **flag** = **true** and empty status.

The error field of MPI_Status is never modified. The success or failure is indicated by the return code only.

When one of the MPI wait or test calls returns **status** for a nonblocking operation request and the corresponding blocking operation does not provide a **status** argument, the **status** from this wait/test does not contain meaningful source, tag or message size information.

When you use this routine in a threaded application, make sure the request is tested on only one thread. The request does not have to be tested on the thread that created the request. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid request handle

Truncation occurred

MPI not initialized

MPI already finalized

Develop mode error if:

Illegal buffer update

Related Information

MPI_TESTALL
MPI_TESTSOME
MPI_TESTANY
MPI_WAIT

MPI_TEST_CANCELLED, MPI_Test_cancelled

Purpose

Tests whether a nonblocking operation was cancelled.

C Synopsis

```
#include <mpi.h>
int MPI_Test_cancelled(MPI_Status * status,int *flag);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_TEST_CANCELLED(INTEGER STATUS(MPI_STATUS_SIZE),INTEGER FLAG,
    INTEGER IERROR)
```

Parameters

status	is a status object (status) (IN). Note that in Fortran a single status object is an array of integers.
flag	**true** if the operation was cancelled (logical) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_TEST_CANCELLED returns **flag** = **true** if the communication associated with the status object was cancelled successfully. In this case, all other fields of **status** (such as count or tag) are undefined. Otherwise, **flag** = **false** is returned. If a receive operation might be cancelled, you should call MPI_TEST_CANCELLED first to check if the operation was cancelled, before checking on the other fields of the return status.

Notes

In this release, nonblocking I/O operations are never cancelled successfully.

Errors

MPI not initialized

MPI already finalized

Related Information

MPI_CANCEL

MPI_TESTALL, MPI_Testall

Purpose

Tests a collection of nonblocking operations for completion.

C Synopsis

```
#include <mpi.h>
int MPI_Testall(int count,MPI_Request *array_of_requests,
    int *flag,MPI_Status *array_of_statuses);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_TESTALL(INTEGER COUNT,INTEGER ARRAY_OF_REQUESTS(*),INTEGER FLAG,
      INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*),INTEGER IERROR)
```

Parameters

count	is the number of requests to test (integer) (IN)
array_of_requests	is an array of requests of length **count** (array of handles) (INOUT)
flag	(logical) (OUT)
array_of_statuses	is an array of status of length **count** objects (array of status) (OUT). Note that in Fortran a status object is itself an array.
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine tests a collection of nonblocking operations for completion. **flag** = **true** is returned if all operations associated with active handles in the array completed, or when no handle in the list is active.

Each status entry of an active handle request is set to the status of the corresponding operation. A request allocated by a nonblocking operation call is deallocated and the handle is set to MPI_REQUEST_NULL.

Each status entry of a null or inactive handle is set to **empty**. If one or more requests have not completed, **flag** = **false** is returned. No request is modified and the values of the status entries are undefined.

The error fields are never modified unless the function gives a return code of MPI_ERR_IN_STATUS. In which case, the error field of every MPI_Status is modified to reflect the result of the corresponding request.

When one of the MPI wait or test calls returns **status** for a nonblocking operation request and the corresponding blocking operation does not provide a **status** argument, the **status** from this wait/test does not contain meaningful source, tag or message size information.

When you use this routine in a threaded application, make sure the request is tested on only one thread. The request does not have to be tested on the thread that created it. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid count	**count** <0
Invalid request array	
Invalid request(s)	
Truncation occurred	
MPI not initialized	
MPI already finalized	

Related Information

MPI_TEST
MPI_WAITALL

MPI_TESTANY, MPI_Testany

Purpose

Tests for the completion of any nonblocking operation.

C Synopsis

```
#include <mpi.h>
int MPI_Testany(int count,MPI_Request *array_of_requests,
    int *index,int *flag,MPI_Status *status);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_TESTANY(INTEGER COUNT,INTEGER ARRAY_OF_REQUESTS(*),INTEGER INDEX,
     INTEGER FLAG,INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
```

Parameters

count	is the list length (integer) (IN)
array_of_requests	is the array of request (array of handles) (INOUT)
index	is the index of the operation that completed, or MPI_UNDEFINED is no operation completed (OUT)
flag	**true** if one of the operations is complete (logical) (OUT)
status	status object (status) (OUT). Note that in Fortran a single status object is an array of integers.
IERROR	is the Fortran return code. It is always the last argument.

Description

If one of the operations has completed, MPI_TESTANY returns **flag** = **true** and returns in **index** the index of this request in the array, and returns in **status** the status of the operation. If the request was allocated by a nonblocking operation, the request is deallocated and the handle is set to MPI_REQUEST_NULL.

If none of the operations has completed, it returns **flag** = **false** and returns a value of MPI_UNDEFINED in **index**, and **status** is undefined. The array can contain null or inactive handles. When the array contains no active handles, then the call returns immediately with **flag** = **true**, **index** = MPI_UNDEFINED, and empty **status**.

MPI_TESTANY(**count, array_of_requests, index, flag, status**) has the same effect as the execution of MPI_TEST(**array_of_requests[i], flag, status**), for **i** = 0, 1, ..., **count-1**, in some arbitrary order, until one call returns **flag** = **true**, or all fail.

The error fields are never modified unless the function gives a return code of MPI_ERR_IN_STATUS. In which case, the error field of every MPI_Status is modified to reflect the result of the corresponding request.

When one of the MPI wait or test calls returns **status** for a nonblocking operation request and the corresponding blocking operation does not provide a **status** argument, the **status** from this wait/test does not contain meaningful source, tag or message size information.

When you use this routine in a threaded application, make sure the request is tested on only one thread. The request does not have to be tested on the thread that created it. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Notes

The array is indexed from zero in C and from one in Fortran.

Errors

Invalid count	**count** <0
Invalid request array	
Invalid request(s)	
Truncation occurred	
MPI not initialized	
MPI already finalized	

Related Information

MPI_TEST
MPI_WAITANY

MPI_TESTSOME, MPI_Testsome

Purpose

Tests a collection of nonblocking operations for completion.

C Synopsis

```
#include <mpi.h>
int MPI_Testsome(int incount,MPI_Request *array_of_requests,
    int *outcount,int *array_of_indices,
    MPI_Status *array_of_statuses);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_TESTSOME(INTEGER INCOUNT,INTEGER ARRAY_OF_REQUESTS(*),
      INTEGER OUTCOUNT,INTEGER ARRAY_OF_INDICES(*),
      INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*),INTEGER IERROR)
```

Parameters

incount	is the length of **array_of_requests** (integer) (IN)
array_of_requests	is the array of requests (array of handles) (INOUT)
outcount	is the number of completed requests (integer) (OUT)
array_of_indices	is the array of indices of operations that completed (array of integers) (OUT)
array_of_statuses	is the array of status objects for operations that completed (array of status) (OUT). Note that in Fortran a status object is itself an array.
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine tests a collection of nonblocking operations for completion. MPI_TESTSOME behaves like MPI_WAITSOME except that MPI_TESTSOME is a local operation and returns immediately. **outcount** = 0 is returned when no operation has completed.

When a request for a receive repeatedly appears in a list of requests passed to MPI_TESTSOME and a matching send is posted, then the receive eventually succeeds unless the send is satisfied by another receive. This fairness requirement also applies to send requests and to I/O requests.

The error fields are never modified unless the function gives a return code of MPI_ERR_IN_STATUS. In which case, the error field of every MPI_Status is modified to reflect the result of the corresponding request.

When one of the MPI wait or test calls returns **status** for a nonblocking operation request and the corresponding blocking operation does not provide a **status** argument, the **status** from this wait/test does not contain meaningful source, tag or message size information.

When you use this routine in a threaded application, make sure the request is tested on only one thread. The request does not have to be tested on the thread that created it. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid count	**count** < 0
Invalid request array	
Invalid request(s)	
Truncation occurred	
MPI not initialized	
MPI already finalized	

Related Information

MPI_TEST
MPI_TESTSOME

MPI_TOPO_TEST, MPI_Topo_test

Purpose

Returns the type of virtual topology associated with a communicator.

C Synopsis

```
#include <mpi.h>
MPI_Topo_test(MPI_Comm comm,int *status);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_TOPO_TEST(INTEGER COMM,INTEGER STATUS,INTEGER IERROR)
```

Parameters

comm	is the communicator (handle) (IN)
status	is the topology type of communicator **comm** (integer) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine returns the type of virtual topology associated with a communicator. The output of **status** will be as follows:

MPI_GRAPH	graph topology
MPI_CART	cartesian topology
MPI_UNDEFINED	no topology

Errors

MPI not initialized

MPI already finalized

Invalid communicator

Related Information

MPI_CART_CREATE
MPI_GRAPH_CREATE

MPI_TYPE_COMMIT, MPI_Type_commit

Purpose

Makes a datatype ready for use in communication.

C Synopsis

```
#include <mpi.h>
int MPI_Type_commit(MPI_Datatype *datatype);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_TYPE_COMMIT(INTEGER DATATYPE,INTEGER IERROR)
```

Parameters

datatype is the datatype that is to be committed (handle) (INOUT)

IERROR is the Fortran return code. It is always the last argument.

Description

A datatype object must be committed before you can use it in communication. You can use an uncommitted datatype as an argument in datatype constructors.

This routine makes a datatype ready for use in communication. The datatype is the formal description of a communication buffer. It is not the content of the buffer.

Once the datatype is committed it can be repeatedly reused to communicate the changing contents of a buffer or buffers with different starting addresses.

Notes

| Basic datatypes are precommitted. It is not an error to call MPI_TYPE_COMMIT on
| a type that is already committed. Types returned by MPI_TYPE_GET_CONTENTS
| may or may not already be committed.

Errors

Invalid datatype

MPI not initialized

MPI already finalized

Related Information

MPI_TYPE_CONTIGUOUS
| MPI_TYPE_CREATE_DARRAY
| MPI_TYPE_CREATE_SUBARRAY
MPI_TYPE_FREE
| MPI_TYPE_GET_CONTENTS
MPI_TYPE_HINDEXED
MPI_TYPE_HVECTOR
MPI_TYPE_INDEXED
MPI_TYPE_STRUCT

MPI_TYPE_VECTOR

MPI_TYPE_CONTIGUOUS, MPI_Type_contiguous

Purpose

Returns a new datatype that represents the concatenation of *count* instances of *oldtype*.

C Synopsis

```
#include <mpi.h>
int MPI_Type_contiguous(int count,MPI_Datatype oldtype,
    MPI_Datatype *newtype);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_TYPE_CONTIGUOUS(INTEGER COUNT,INTEGER OLDTYPE,INTEGER NEWTYPE,
      INTEGER IERROR)
```

Parameters

count	is the replication **count** (non-negative integer) (IN)
oldtype	is the old datatype (handle) (IN)
newtype	is the new datatype (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine returns a new datatype that represents the concatenation of **count** instances of **oldtype**. MPI_TYPE_CONTIGUOUS allows replication of a datatype into contiguous locations.

Notes

newtype must be committed using MPI_TYPE_COMMIT before being used for communication.

Errors

Invalid count **count** < 0

Undefined oldtype

Oldtype is MPI_LB, MPI_UB, or MPI_PACKED

| **Stride overflow**

| **Extent overflow**

| **Size overflow**

| **Upper or lower bound overflow**

MPI not initialized

MPI already finalized

Related Information

MPI_TYPE_COMMIT
| MPI_TYPE_FREE
| MPI_TYPE_GET_CONTENTS
| MPI_TYPE_GET_ENVELOPE

MPI_TYPE_CREATE_DARRAY, MPI_Type_create_darray

Purpose

Generates the datatypes corresponding to the distribution of an *ndims*–dimensional array of *oldtype* elements onto an *ndims*–dimensional grid of logical tasks.

C Synopsis

```
#include <mpi.h>
int MPI_Type_create_darray (int size,int rank,int ndims,
    int array_of_gsizes[],int array_of_distribs[],
    int array_of_dargs[],int array_of_psizes[],
    int order,MPI_Datatype oldtype,MPI_Datatype *newtype);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_TYPE_CREATE_DARRAY (INTEGER SIZE,INTEGER RANK,INTEGER NDIMS,
    INTEGER ARRAY_OF_GSIZES(*),INTEGER ARRAY_OF_DISTRIBS(*),
    INTEGER ARRAY_OF_DARGS(*),INTEGER ARRAY_OF_PSIZES(*),
    INTEGER ORDER,INTEGER OLDTYPE,INTEGER NEWTYPE,INTEGER IERROR)
```

Parameters

size	is the size of the task group (positive integer)(IN)
rank	is the rank in the task group (nonnegative integer)(IN)
ndims	is the number of array dimensions as well as task grid dimensions (positive integer)(IN)
array_of_gsizes	is the number of elements of type **oldtype** in each dimension of the global array (array of positive integers)(IN)
array_of_distribs	is the distribution of the global array in each dimension (array of state)(IN)
array_of_dargs	is the distribution argument in each dimension of the global array (array of positive integers)(IN)
array_of_psizes	is the size of the logical **grid** of tasks in each dimension (array of positive integers)(IN)
order	is the array storage order **flag** (state)(IN)
oldtype	is the old datatype (handle)(IN)
newtype	is the new datatype (handle)(OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_TYPE_CREATE_DARRAY generates the datatypes corresponding to an HPF-like distribution of an *ndims*–dimensional array of **oldtype** elements onto an *ndims*–dimensional grid of logical tasks. The ordering of tasks in the task grid is assumed to be row-major. See *The High Performance Fortran Handbook* for more information.

Errors

Fatal Errors:

MPI not initialized	
MPI already finalized	
Invalid group size	**size** must be a positive integer
Invalid rank	**rank** must be a nonnegative integer
Invalid dimension count	*ndims* must be a positive integer
Invalid array element	Each element of **array_of_gsizes** and **array_of_psizes** must be a positive integer
Invalid distribution element	Each element of **array_of_distribs** must be either MPI_DISTRIBUTE_BLOCK, MPI_DISTRIBUTE_CYCLIC, or MPI_DISTRIBUTE_NONE
Invalid darg element	Each element of **array_of_dargs** must be a positive integer or equal to MPI_DISTRIBUTE_DFLT_DARG
Invalid order	**order** must either be MPI_ORDER_C or MPI_ORDER_Fortran
MPI_DATATYPE_NULL not valid	**oldtype** cannot be equal to MPI_DATATYPE_NULL
Undefined datatype	**oldtype** is not a defined datatype
Invalid datatype	**oldtype** cannot be MPI_LB, MPI_UB or MPI_PACKED
Invalid grid size	The product of the elements of **array_of_psizes** must be equal to **size**
Invalid block distribution	The condition (**array_of_psizes[i]* array_of_dargs[i])**<**array_of_ gsizes[i]** must be satisfied for all indices **i** between **0** and **ndims-1** for which a block distribution is specified
Invalid psize element	Each element of **array_of_psizes** must be equal to **1** if the same element of **array_of_distribs** has a value of MPI_DISTRIBUTE_NONE
Stride overflow	
Extent overflow	
Size overflow	
Upper or lower bound overflow	

| ## Related Information

| MPI_TYPE_COMMIT
| MPI_TYPE_FREE
| MPI_TYPE_GET_CONTENTS
| MPI_TYPE_GET_ENVELOPE

MPI_TYPE_CREATE_SUBARRAY, MPI_Type_create_subarray

Purpose

Returns a new datatype that represents an *ndims*-dimensional subarray of an *ndims*-dimensional array.

C Synopsis

```
#include <mpi.h>
int MPI_Type_create_subarray (int ndims,int array_of_sizes[],
    int array_of_subsizes[],int array_of_starts[],
    int order,MPI_Datatype oldtype,MPI_Datatype *newtype);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_TYPE_CREATE_SUBARRAY (INTEGER NDIMS,INTEGER ARRAY_OF_SUBSIZES(*),
    INTEGER ARRAY_OF_SIZES(*),INTEGER ARRAY_OF_STARTS(*),
    INTEGER ORDER,INTEGER OLDTYPE,INTEGER NEWTYPE,INTEGER IERROR)
```

Parameters

ndims	is the number of array dimensions(positive integer)(IN)
array_of_sizes	is the number of elements of type **oldtype** in each dimension of the full array (array of positive integers)(IN)
array_of_subsizes	is the number of type **oldtype** in each dimension of the subarray (array of positive integers)(IN)
array_of_starts	is the starting coordinates of the subarray in each dimension (array of nonnegative integers)(IN)
order	is the array storage order **flag**(state)(IN)
oldtype	is the array element datatype (handle)(IN)
newtype	is the new datatype (handle)(OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_TYPE_CREATE_SUBARRAY creates an MPI datatype describing an ndims-dimensional subarray of an ndims-dimensional array. The subarray may be situated anywhere within the full array and may be of any nonzero size up to the size of the larger array as long as it is confined within this array.

This function facilitates creating filetypes to access arrays distributed in blocks among tasks to a single file that contains the full array.

Errors

Fatal Errors:

MPI not initialized	
MPI already finalized	
Invalid dimension count	*ndims* must be a positive integer
Invalid array element	Each element of **array_of_sizes** and **array_of_subsizes** must be a positive integer, and each element of **array_of_starts** must be a nonnegative integer
Invalid order	**order** must be either MPI_ORDER_C or MPI_ORDER_Fortran
MPI_DATATYPE_NULL not valid	**oldtype** cannot be equal to MPI_DATATYPE_NULL
Undefined datatype	**oldtype** is not a defined datatype
Invalid datatype	**oldtype** cannot be MPI_LB, MPI_UB or MPI_PACKED
Invalid subarray size	Each element of **array_of_subsizes** cannot be greater than the same element of **array_of_sizes**
Invalid start element	The subarray must be fully contained within the full array.
Stride overflow	
Extent overflow	
Size overflow	
Upper or lower bound overflow	

Related Information

MPI_TYPE_COMMIT
MPI_TYPE_FREE
MPI_TYPE_GET_CONTENTS
MPI_TYPE_GET_ENVELOPE

MPI_TYPE_EXTENT, MPI_Type_extent

Purpose

Returns the extent of any defined datatype.

C Synopsis

```
#include <mpi.h>
int MPI_Type_extent(MPI_Datatype datatype,int *extent);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_TYPE_EXTENT(INTEGER DATATYPE,INTEGER EXTENT,INTEGER IERROR)
```

Parameters

datatype	is the datatype (handle) (IN)
extent	is the datatype extent (integer) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine returns the extent of a datatype. The extent of a datatype is the span from the first byte to the last byte occupied by entries in this datatype and rounded up to satisfy alignment requirements.

Notes

Rounding for alignment is not done when MPI_UB is used to define the datatype. Types defined with MPI_LB, MP_UB or with any type that itself contains MPI_LB or MPI_UB may return an extent which is not directly related to the layout of data in memory. Refer to MPI_Type_struct for more information on MPI_LB and MPI_UB.

Errors

Invalid datatype

MPI not initialized

MPI already finalized

Related Information

MPI_TYPE_SIZE

MPI_TYPE_FREE, MPI_Type_free

Purpose

Marks a datatype for deallocation.

C Synopsis

```
#include <mpi.h>
int MPI_Type_free(MPI_Datatype *datatype);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_TYPE_FREE(INTEGER DATATYPE,INTEGER IERROR)
```

Parameters

datatype	is the datatype to be freed (handle) (INOUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine marks the datatype object associated with **datatype** for deallocation. It sets **datatype** to MPI_DATATYPE_NULL. All communication currently using this datatype completes normally. Derived datatypes defined from the freed datatype are not affected.

| ### Notes

| MPI_FILE_GET_VIEW and MPI_TYPE_GET_CONTENTS both return new
| references or handles for existing MPI_Datatypes. Each new reference to a derived
| type should be freed after the reference is no longer needed. New references to
| named types must not be freed. You can identify a derived datatype by calling
| MPI_TYPE_GET_ENVELOPE and checking that the combiner is not
| MPI_COMBINER_NAMED. MPI cannot discard a derived MPI_datatype if there are
| any references to it that have not been freed by MPI_TYPE_FREE.

Errors

Invalid datatype

Predefined datatype

Type is already free

MPI not initialized

MPI already finalized

Related Information

MPI_TYPE_COMMIT
| MPI_FILE_GET_VIEW
| MPI_TYPE_GET_CONTENTS
| MPI_TYPE_GET_ENVELOPE

MPI_TYPE_GET_CONTENTS, MPI_Type_get_contents

Purpose

Obtains the arguments used in the creation of the datatype.

C Synopsis

```
#include <mpi.h>
int MPI_Type_get_contents(MPI_Datatype datatype,
    int *max_integers, int *max_addresses, int *max_datatypes,
    int array_of_integers[],
    int array_of_addresses[],
    int array_of_datatypes[]);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_TYPE_GET_CONTENTS(INTEGER DATATYPE, INTEGER MAX_INTEGERS,
    INTEGER MAX_ADDRESSES, INTEGER MAX_DATATYPES,
    INTEGER ARRAY_of_INTEGERS, INTEGER ARRAY_OF_ADDRESSES,
    INTEGER ARRAY_of_DATATYPES, INTEGER IERROR)
```

Parameters

datatype	is the datatype to access (handle) (IN)
max_integers	is the number of elements in array_of_integers (non-negative integer) (IN)
max_addresses	is the number of elements in the array_of_addresses (non-negative integer) (IN)
max_datatypes	is the number of elements in array_of_datatypes (non-negative integer) (IN)
array_of_integers	contains integer arguments used in the constructing datatype (array of integers) (OUT)
array_of_addresses	contains address arguments used in the constructing datatype (array of integers) (OUT)
array_of_datatypes	contains datatype arguments used in the constructing datatype (array of handles) (OUT)

If the combiner is MPI_COMBINER_NAMED, it is erroneous to call MPI_TYPE_GET_CONTENTS.

Table 4 lists the combiners and constructor arguments. The lowercase names of the arguments are shown.

Table 4 (Page 1 of 3). Combiners and Constructor Arguments

Constructor Argument	**C Location**	**Fortran Location**	**ni na nd**
MPI_COMBINER_DUP			

Table 4 (Page 2 of 3). Combiners and Constructor Arguments

Constructor Argument	C Location	Fortran Location	ni na nd
oldtype	d[0]	D(1)	0 0 1
MPI_COMBINER_CONTIGUOUS			

count
oldtype
	i[0]
	d[0]
		I(1)
		D(1)
			1
			0
			1

MPI_COMBINER_VECTOR			

count
blocklength
stride
oldtype
	i[0]
	i[1]
	i[2]
	d[0]
		I(1)
		I(2)
		I(3)
		D(1)
			3
			0
			1

MPI_COMBINER_HVECTOR MPI_COMBINER_HVECTOR_INTEGER			

count
blocklength
stride
oldtype
	i[0]
	i[1]
	a[0]
	d[0]
		I(1)
		I(2)
		A(1)
		D(1)
			2
			1
			1

MPI_COMBINER_INDEXED			

count
array_of_blocklengths
array_of_displacements
oldtype
	i[0]
	i[1] to i[i[0]]
	i[i[0]+1] to i[2*i[0]]
	d[0]
		I(1)
		I(2) to I(I(1)+1)
		I(I(1)+2) to I(2*I(1)+1)
		D(1)
			2*count+1
			0
			1

MPI_COMBINER_HINDEXED MPI_COMBINER_HINDEXED_INTEGER			

count
array_of_blocklengths
array_of_displacements
oldtype
	i[0]
	i[1] to i[i[0]]
	a[0] to a[i[0]-1]
	d[0]
		I(1)
		I(2) to I(I(1)+1)
		A(1) to A(I(1))
		D(1)
			count+1
			count
			1

MPI_COMBINER_INDEXED_BLOCK			

count
blocklength
array_of_displacements
oldtype
	i[0]
	i[1]
	i[2] to i[i[0]+1]
	d[0]
		I(1)
		I(2)
		I(3) to I(I(1)+2)
		D(1)
			count+2
			0
			1

MPI_COMBINER_STRUCT MPI_COMBINER_STRUCT_INTEGER			

count
array_of_blocklengths
array_of_displacements
array_of_types
	i[0]
	i[1] to i[i[0]]
	a[0] to a[i[0]-1]
	d[0] to d[i[0]-1]
		I(1)
		I(2) to I(I(1)+1)
		A(1) to A(I(1))
		D(1)
			count+1
			count
			count

MPI_COMBINER_SUBARRAY			

Table 4 (Page 3 of 3). Combiners and Constructor Arguments

Constructor Argument	**C Location**	**Fortran Location**	**ni na nd**

ndims
array_of_sizes
array_of_subsizes
array_of_starts
order
oldtype
	i[0]
	i[1] to i[i[0]]
	i[i[0]+1] to i[2*i[0]]
	i[2*i[0]+1] to i[3*i[0]]
	d[0]
		I(1)
		I(2) to I(I(1)+1)
		I(I(1)+2) to I(2*I(1)+1)
		I(2*I(1)+2) to I(3*I(1)+1)
		I(3*I(1)+2)
		D(1)
			3*ndims+2
			0
			1

MPI_COMBINER_DARRAY			

size
rank
ndims
array_of_gsizes
array_of_distribs
array_of_dargs
array_of_psizes
order
oldtype
	i[0]
	i[1]
	i[2]
	i[3] to i[i[2]+2]
	i[i[2]+3] to i[2*i[2]+2]
	i[2*i[2]+3] to i[3*i[2]+2]
	i[3*i[2]+3] to i[4*i[2]+2]
	i[4*i[2]+3]
	d[0]
		I(1)
		I(2)
		I(3)
		I(4) to I(I(3)+3)
		I(I(3)+4) to I(2*I(3)+3)
		I(2*I(3)+4) to I(3*I(3)+3)
		I(3*I(3)+4) to I(4*I(3)+3)
		I(4*I(3)+4)
		D(1)
			4*ndims+4
			0
			1

MPI_COMBINER_F90_REAL MPI_COMBINER_F90_COMPLEX			

p
r
	i[0]
	i[1]
		I(1)
		I(2)
			2
			0
			0

MPI_COMBINER_F90_INTEGER			
r	i[0]	I(1)	1 0 0
MPI_COMBINER_RESIZED			

lb
extent
oldtype
	a[0]
	a[1]
	d[0]
		A(1)
		A(2)
		D(1)
			0
			2
			1

Description

MPI_TYPE_GET_CONTENTS identifies the combiner and returns the arguments that were used with this combiner to create the datatype of interest. A call to MPI_TYPE_GET_CONTENTS is normally preceded by a call to MPI_TYPE_GET_ENVELOPE to discover whether the type of interest is one that can be decoded and if so, how large the output arrays must be. An MPI_COMBINER_NAMED datatype is a predefined type that may not be decoded. The datatype handles returned in array_of_datatypes can include both named and derived types. The derived types may or may not already be committed. Each entry in array_of_datatypes is a separate datatype handle that must eventually be freed if it represents a derived type.

Notes

An MPI type constructor, such as MPI_TYPE_CONTIGUOUS, creates a datatype object within MPI and gives a handle for that object to the caller. This handle represents one reference to the object. In this implementation of MPI, the MPI datatypes obtained with calls to MPI_TYPE_GET_CONTENTS are new handles for the existing datatype objects. The number of handles (references) given to the user is tracked by a reference counter in the object. MPI cannot discard a datatype object unless MPI_TYPE_FREE has been called on every handle the user has obtained.

The use of reference-counted objects is encouraged, but not mandated, by the MPI standard. Another MPI implementation may create new objects instead. The user should be aware of a side effect of the reference count approach. Suppose mytype was created by a call to MPI_TYPE_VECTOR and used so that a later call to MPI_TYPE_GET_CONTENTS returns its handle in hertype. Because both handles identify the same datatype object, attribute changes made with either handle are changes in the single object. That object will exist at least until MPI_TYPE_FREE has been called on both mytype and hertype. Freeing either handle alone will leave the object intact and the other handle will remain valid.

Errors

Invalid datatype

Predefined datatype

Maximum array size is not big enough

MPI already finalized

MPI not initialized

Related Information

MPI_TYPE_COMMIT
MPI_TYPE_FREE
MPI_TYPE_GET_ENVELOPE

MPI_TYPE_GET_ENVELOPE, MPI_Type_get_envelope

Purpose

Determines the constructor that was used to create the datatype and the amount of data that will be returned by a call to MPI_TYPE_GET_CONTENTS for the same datatype.

C Synopsis

```
#include <mpi.h>
int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers,
    int *num_addresses, int *num_datatypes, int *combiner);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_TYPE_GET_ENVELOPE(INTEGER DATATYPE, INTEGER NUM_INTEGERS,
    INTEGER NUM_ADDRESSES, INTEGER NUM_DATATYPES, INTEGER COMBINER,
    INTEGER IERROR)
```

Parameters

datatype	is the datatype to access (handle) (IN)
num_integers	is the number of input integers used in the call constructing combiner (non-negative integer) (OUT)
num_addresses	is the number of input addresses used in the call constructing combiner (non-negative integer) (OUT)
num_datatypes	is the number of input datatypes used in the call constructing combiner (non-negative integer) (OUT)
combiner	is the combiner (state) (OUT)

Table 5 lists the combiners and the calls associated with them.

Table 5 (Page 1 of 2). Combiners and Calls

Combiner	**What It Represents**
MPI_COMBINER_NAMED	A named, predefined datatype
MPI_COMBINER_DUP	MPI_TYPE_DUP
MPI_COMBINER_CONTIGUOUS	MPI_TYPE_CONTIGUOUS
MPI_COMBINER_VECTOR	MPI_TYPE_VECTOR
MPI_COMBINER_HVECTOR	MPI_TYPE_HVECTOR from C and in some cases Fortran or MPI_TYPE_CREATE_HVECTOR
MPI_COMBINER_HVECTOR_INTEGER	MPI_TYPE_HVECTOR from Fortran
MPI_COMBINER_INDEXED	MPI_TYPE_INDEXED
MPI_COMBINER_HINDEXED	MPI_TYPE_HINDEXED from C and in some cases Fortran or MPI_TYPE_CREATE_HINDEXED
MPI_COMBINER_HINDEXED_INTEGER	MPI_TYPE_HINDEXED from Fortran

Table 5 (Page 2 of 2). Combiners and Calls

Combiner	**What It Represents**
MPI_COMBINER_INDEXED_BLOCK	MPI_TYPE_CREATE_INDEXED_BLOCK
MPI_COMBINER_STRUCT	MPI_TYPE_STRUCT from C and in some cases Fortran or MPI_TYPE_CREATE_STRUCT
MPI_COMBINER_STRUCT_INTEGER	MPI_TYPE_STRUCT from Fortran
MPI_COMBINER_SUBARRAY	MPI_TYPE_CREATE_SUBARRAY
MPI_COMBINER_DARRAY	MPI_TYPE_CREATE_DARRAY
MPI_COMBINER_F90_REAL	MPI_TYPE_CREATE_F90_REAL
MPI_COMBINER_F90_COMPLEX	MPI_TYPE_CREATE_F90_COMPLEX
MPI_COMBINER_F90_INTEGER	MPI_TYPE_CREATE_F90_INTEGER
MPI_COMBINER_RESIZED	MPI_TYPE_CREATE_RESIZED

Description

MPI_TYPE_GET_ENVELOPE provides information about an unknown datatype which will allow it to be decoded if appropriate. This includes identifying the combiner used to create the unknown type and the sizes that the arrays must be if MPI_TYPE_GET_CONTENTS is to be called. MPI_TYPE_GET_ENVELOPE is also used to determine whether a datatype handle returned by MPI_TYPE_GET_CONTENTS or MPI_FILE_GET_VIEW is for a predefined, named datatype. When the combiner is MPI_COMBINER_NAMED, it is an error to call MPI_TYPE_GET_CONTENTS or MPI_TYPE_FREE with the datatype.

Errors

Invalid datatype

MPI already finalized

MPI not initialized

Related Information

MPI_TYPE_FREE
MPI_TYPE_GET_CONTENTS

MPI_TYPE_HINDEXED, MPI_Type_hindexed

Purpose

Returns a new datatype that represents *count* blocks. Each block is defined by an entry in *array_of_blocklengths* and *array_of_displacements*. Displacements are expressed in bytes.

C Synopsis

```
#include <mpi.h>
int MPI_Type_hindexed(int count,int *array_of_blocklengths,
    MPI_Aint *array_of_displacements,MPI_Datatype oldtype,
    MPI_Datatype *newtype);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_TYPE_HINDEXED(INTEGER COUNT,INTEGER ARRAY_OF_BLOCKLENGTHS(*),
    INTEGER ARRAY_OF DISPLACEMENTS(*),INTEGER OLDTYPE,INTEGER NEWTYPE,
    INTEGER IERROR)
```

Parameters

count	is the number of blocks and the number of entries in **array_of_displacements** and **array_of_blocklengths** (non-negative integer) (IN)
array_of_blocklengths	is the number of instances of **oldtype** for each block (array of non-negative integers) (IN)
array_of_displacements	is a byte displacement for each block (array of integer) (IN)
oldtype	is the old datatype (handle) (IN)
newtype	is the new datatype (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine returns a new datatype that represents **count** blocks. Each is defined by an entry in **array_of_blocklengths** and **array_of_displacements**. Displacements are expressed in bytes rather than in multiples of the **oldtype** extent as in MPI_TYPE_INDEXED.

Notes

newtype must be committed using MPI_TYPE_COMMIT before being used for communication.

Errors

Invalid count	**count** < 0
Invalid blocklength	**blocklength** [i] < 0
Undefined oldtype	
Oldtype is MPI_LB, MPI_UB or MPI_PACKED	
MPI not initialized	
MPI already finalized	

Related Information

MPI_TYPE_COMMIT
| MPI_TYPE_FREE
| MPI_TYPE_GET_CONTENTS
| MPI_TYPE_GET_ENVELOPE
MPI_TYPE_INDEXED

MPI_TYPE_HVECTOR, MPI_Type_hvector

Purpose

Returns a new datatype that represents equally-spaced blocks. The spacing between the start of each block is given in bytes.

C Synopsis

```
#include <mpi.h>
int MPI_Type_hvector(int count,int blocklength,MPI_Aint stride,
    MPI_Datatype oldtype,MPI_Datatype *newtype);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_TYPE_HVECTOR(INTEGER COUNT,INTEGER BLOCKLENGTH,INTEGER STRIDE,
    INTEGER OLDTYPE,INTEGER NEWTYPE,INTEGER IERROR)
```

Parameters

count	is the number of blocks (non-negative integer) (IN)
blocklength	is the number of **oldtype** instances in each block (non-negative integer) (IN)
stride	is an integer specifying the number of bytes between start of each block. (IN)
oldtype	is the old datatype (handle) (IN)
newtype	is the new datatype (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine returns a new datatype that represents **count** equally spaced blocks. Each block is a concatenation of **blocklength** instances of **oldtype**. The origins of the blocks are spaced **stride** units apart where the counting unit is one byte.

Notes

newtype must be committed using MPI_TYPE_COMMIT before being used for communication.

Errors

Invalid count	**count** < 0
Invalid blocklength	**blocklength** < 0
Undefined oldtype	
Oldtype is MPI_LB, MPI_UB or MPI_PACKED	

MPI not initialized

MPI already finalized

Related Information

MPI_TYPE_COMMIT
| MPI_TYPE_FREE
| MPI_TYPE_GET_CONTENTS
| MPI_TYPE_GET_ENVELOPE
MPI_TYPE_VECTOR

MPI_TYPE_INDEXED, MPI_Type_indexed

Purpose

Returns a new datatype that represents *count* blocks. Each block is defined by an entry in *array_of_blocklengths* and *array_of_displacements*. Displacements are expressed in units of extent(*oldtype*).

C Synopsis

```
#include <mpi.h>
int MPI_Type_indexed(int count,int *array_of_blocklengths,
    int *array_of_displacements,MPI_Datatype oldtype,
    MPI_datatype *newtype);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_TYPE_INDEXED(INTEGER COUNT,INTEGER ARRAY_OF_BLOCKLENGTHS(*),
    INTEGER ARRAY_OF DISPLACEMENTS(*),INTEGER OLDTYPE,INTEGER NEWTYPE,
    INTEGER IERROR)
```

Parameters

count	is the number of blocks and the number of entries in **array_of_displacements** and **array_of_blocklengths** (non-negative integer) (IN)
array_of_blocklengths	is the number of instances of **oldtype** in each block (array of non-negative integers) (IN)
array_of_displacements	is the displacement of each block in units of extent(**oldtype**) (array of integer) (IN)
oldtype	is the old datatype (handle) (IN)
newtype	is the new datatype (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine returns a new datatype that represents **count** blocks. Each is defined by an entry in **array_of_blocklengths** and **array_of_displacements**. Displacements are expressed in units of extent(**oldtype**).

Notes

newtype must be committed using MPI_TYPE_COMMIT before being used for communication.

Errors

Invalid count	**count** < 0
Invalid count	**blocklength** [i] < 0
Undefined oldtype	
Oldtype is MPI_LB, MPI_UB or MPI_PACKED	
MPI not initialized	
MPI already finalized	

Related Information

MPI_TYPE_COMMIT
| MPI_TYPE_FREE
| MPI_TYPE_GET_CONTENTS
| MPI_TYPE_GET_ENVELOPE
MPI_TYPE_HINDEXED

MPI_TYPE_LB, MPI_Type_lb

Purpose

Returns the lower bound of a datatype.

C Synopsis

```
#include <mpi.h>
int MPI_Type_lb(MPI_Datatype datatype,int *displacement);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_TYPE_LB(INTEGER DATATYPE,INTEGER DISPLACEMENT,INTEGER IERROR)
```

Parameters

datatype	is the datatype (handle) (IN)
displacement	is the displacement of lower bound from the origin in bytes (integer) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine returns the lower bound of a specific datatype.

Normally the lower bound is the offset of the lowest address byte in the datatype. Datatype constructors with explicit MPI_LB and vector constructors with negative stride can produce lb < 0. Lower bound cannot be greater than upper bound. For a type with MPI_LB in its ancestry, the value returned by MPI_TYPE_LB may not be related to the displacement of the lowest address byte. Refer to MPI_TYPE_STRUCT for more information on MPI_LB and MPI_UB.

Errors

Invalid datatype

MPI not initialized

MPI already finalized

Related Information

MPI_TYPE_UB
MPI_TYPE_STRUCT

MPI_TYPE_SIZE, MPI_Type_size

Purpose

Returns the number of bytes represented by any defined datatype.

C Synopsis

```
#include <mpi.h>
int MPI_Type_size(MPI_Datatype datatype,int *size);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_TYPE_SIZE(INTEGER DATATYPE,INTEGER SIZE,INTEGER IERROR)
```

Parameters

datatype	is the datatype (handle) (IN)
size	is the datatype size (integer) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine returns the total number of bytes in the type signature associated with **datatype**. Entries with multiple occurrences in the datatype are counted.

Errors

Invalid datatype

MPI not initialized

MPI already finalized

Related Information

MPI_TYPE_EXTENT

MPI_TYPE_STRUCT, MPI_Type_struct

Purpose

Returns a new datatype that represents *count* blocks. Each is defined by an entry in *array_of_blocklengths*, *array_of_displacements* and *array_of_types*. Displacements are expressed in bytes.

C Synopsis

```
#include <mpi.h>
int MPI_Type_struct(int count,int *array_of_blocklengths,
    MPI_Aint *array_of_displacements,MPI_Datatype *array_of_types,
    MPI_datatype *newtype);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_TYPE_STRUCT(INTEGER COUNT,INTEGER ARRAY_OF_BLOCKLENGTHS(*),
    INTEGER ARRAY_OF DISPLACEMENTS(*),INTEGER ARRAY_OF_TYPES(*),
    INTEGER NEWTYPE,INTEGER IERROR)
```

Parameters

count	is an integer specifying the number of blocks. It is also the number of entries in arrays **array_of_types**, **array_of_displacements** and **array_of_blocklengths**. (IN)
array_of_blocklengths	is the number of elements in each block (array of integer). That is, **array_of_blocklengths(i)** specifies the number of instances of type **array_of_types(i)**in block(i). (IN)
array_of_displacements	is the byte displacement of each block (array of integer) (IN)
array_of_types	is the datatype comprising each block. That is, block(i) is made of a concatenation of type **array_of_types(i)**. (array of handles to datatype objects) (IN)
newtype	is the new datatype (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine returns a new datatype that represents **count** blocks. Each is defined by an entry in **array_of_blocklengths**, **array_of_displacements** and **array_of_types**. Displacements are expressed in bytes.

MPI_TYPE_STRUCT is the most general type constructor. It allows each block to consist of replications of different datatypes. This is the only constructor which allows MPI pseudo types MPI_LB and MPI_UB. Without these pseudo types, the extent of a datatype is the range from the first byte to the last byte rounded up as needed to meet boundary requirements. For example, if a type is made of an integer followed by 2 characters, it will still have an extent of 8 because it is padded to meet the boundary constraints of an int. This is intended to match the behavior of a compiler defining an array of such structures.

Because there may be cases in which this default behavior is not correct, MPI provides a means to set explicit upper and lower bounds which may not be directly related to the lowest and highest displacement datatype. When the pseudo type MPI_UB is used, the upper bound will be the value specified as the displacement of the MPI_UB block. No rounding for alignment is done. MPI_LB can be used to set an explicit lower bound but its use does not suppress rounding. When MPI_UB is not used, the upper bound of the datatype is adjusted to make the extent a multiple of the type's most boundary constrained component.

The marker placed by a MPI_LB or MPI_UB is 'sticky'. For example, assume type A is defined with a MPI_UB at 100. Type B is defined with a type A at 0 and a MPI_UB at 50. In effect, type B has received a MPI_UB at 50 and an inherited MPI_UB at 100. Because the inherited MPI_UB is higher, it is kept in the type B definition and the MPI_UB explicitly placed at 50 is discarded.

Notes

newtype must be committed using MPI_TYPE_COMMIT before being used for communication.

Errors

Invalid count	**count** < 0
Invalid blocklength	**blocklength**[i] < 0
Undefined oldtype in array_of_types	
MPI not initialized	
MPI already finalized	

Related Information

MPI_TYPE_COMMIT
| MPI_TYPE_FREE
| MPI_TYPE_GET_CONTENTS
| MPI_TYPE_GET_ENVELOPE

MPI_TYPE_UB, MPI_Type_ub

Purpose

Returns the upper bound of a datatype.

C Synopsis

```
#include <mpi.h>
int MPI_Type_ub(MPI_Datatype datatype,int *displacement);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_TYPE_UB(INTEGER DATATYPE,INTEGER DISPLACEMENT,
INTEGER IERROR)
```

Parameters

datatype	is the datatype (handle) (IN)
displacement	is the displacement of upper bound from origin in bytes (integer) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine returns the upper bound of a specific datatype.

The upper bound is the displacement you use in locating the origin byte of the next instance of **datatype** for operations which use count and datatype. In the normal case, ub represents the displacement of the highest address byte of the datatype + e (where e >= 0 and results in (ub − lb) being a multiple of the boundary requirement for the most boundary constrained type in the datatype). If MPI_UB is used in a type constructor, no alignment adjustment is done so ub is exactly as you set it.

For a type with MPI_UB in its ancestry, the value returned by MPI_TYPE_UB may not be related to the displacement of the highest address byte (with rounding). Refer to MPI_TYPE_STRUCT for more informatin on MPI_LB and MPI_UB.

Errors

Invalid datatype

MPI not initialized

MPI already finalized

Related Information

MPI_TYPE_LB
| MPI_TYPE_STRUCT

MPI_TYPE_VECTOR, MPI_Type_vector

Purpose

Returns a new datatype that represents equally spaced blocks. The spacing between the start of each block is given in units of extent (*oldtype*).

C Synopsis

```
#include <mpi.h>
int MPI_Type_vector(int count,int blocklength,int stride,
    MPI_Datatype oldtype,MPI_Datatype *newtype);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_TYPE_VECTOR(INTEGER COUNT,INTEGER BLOCKLENGTH,
    INTEGER STRIDE,INTEGER OLDTYPE,INTEGER NEWTYPE,INTEGER IERROR)
```

Parameters

count	is the number of blocks (non-negative integer) (IN)
blocklength	is the number of **oldtype** instances in each block (non-negative integer) (IN)
stride	is the number of units between the start of each block (integer) (IN)
oldtype	is the old datatype (handle) (IN)
newtype	is the new datatype (handle) (OUT)
IERROR	is the Fortran return code. It is always the last argument.

Description

This function returns a new datatype that represents **count** equally spaced blocks. Each block is a a concatenation of **blocklength** instances of **oldtype**. The origins of the blocks are spaced **stride** units apart where the counting unit is extent(**oldtype**). That is, from one origin to the next in bytes = **stride** * extent (**oldtype**).

Notes

newtype must be committed using MPI_TYPE_COMMIT before being used for communication.

Errors

Invalid count	**count** < 0
Invalid blocklength	**blocklength** < 0
Undefined oldtype	
Oldtype is MPI_LB, MPI_UB or MPI_PACKED	

MPI not initialized

MPI already finalized

Related Information

| MPI_TYPE_COMMIT
| MPI_TYPE_FREE
| MPI_TYPE_GET_CONTENTS
| MPI_TYPE_GET_ENVELOPE
MPI_TYPE_HVECTOR

MPI_UNPACK, MPI_Unpack

Purpose

Unpacks the message into the specified receive buffer from the specified packed buffer.

C Synopsis

```
#include <mpi.h>
int MPI_Unpack(void* inbuf,int insize,int *position,
    void *outbuf,int outcount,MPI_Datatype datatype,
    MPI_Comm comm);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_UNPACK(CHOICE INBUF,INTEGER INSIZE,INTEGER POSITION,
     CHOICE OUTBUF,INTEGER OUTCOUNT,INTEGER DATATYPE,INTEGER COMM,
     INTEGER IERROR)
```

Parameters

inbuf	is the input buffer start (choice) (IN)
insize	is an integer specifying the size of input buffer in bytes (IN)
position	is an integer specifying the current packed buffer offset in bytes (INOUT)
outbuf	is the output buffer start (choice) (OUT)
outcount	is an integer specifying the number of instances of **datatype** to be unpacked (IN)
datatype	is the datatype of each output data item (handle) (IN)
comm	is the communicator for the packed message (handle) (IN)
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine unpacks the message specified by **outbuf**, **outcount**, and **datatype** from the buffer space specified by **inbuf** and **insize**. The output buffer is any receive buffer allowed in MPI_RECV. The input buffer is any contiguous storage space containing **insize** bytes and starting at address **inbuf**.

The input value of **position** is the beginning offset in the input buffer for the data to be unpacked. The output value of **position** is the offset in the input buffer following the data already unpacked. That is, the starting point for another call to MPI_UNPACK. **comm** is the communicator that was used to receive the packed message.

Notes

In MPI_UNPACK the **outcount** argument specifies the actual number of items to be unpacked. The size of the corresponding message is the increment in **position**.

Errors

Invalid outcount	**outcount** < 0
Invalid datatype	
Type is not committed	
Invalid communicator	
Inbuf too small	
MPI not initialized	
MPI already finalized	

Related Information

MPI_PACK

MPI_WAIT, MPI_Wait

Purpose

Waits for a nonblocking operation to complete.

C Synopsis

```
#include <mpi.h>
int MPI_Wait(MPI_Request *request,MPI_Status *status);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_WAIT(INTEGER REQUEST,INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
```

Parameters

request	is the request to wait for (handle) (INOUT)
status	is the status object (status) (OUT). Note that in Fortran a single status object is an array of integers.
IERROR	is the Fortran return code. It is always the last argument.

Description

MPI_WAIT returns after the operation identified by **request** completes. If the object associated with **request** was created by a nonblocking operation, the object is deallocated and **request** is set to MPI_REQUEST_NULL. MPI_WAIT is a non-local operation.

You can call MPI_WAIT with a null or inactive **request** argument. The operation returns immediately. The **status** argument returns **tag** = MPI_ANY_TAG, **source** = MPI_ANY_SOURCE. The **status** argument is also internally configured so that calls to MPI_GET_COUNT and MPI_GET_ELEMENTS return **count** = 0. (This is called an **empty** status.)

Information on the completed operation is found in **status**. You can query the status object for a send or receive operation with a call to MPI_TEST_CANCELLED. For receive operations, you can also retrieve information from **status** with MPI_GET_COUNT and MPI_GET_ELEMENTS. If wildcards were used by the receive for either the source or tag, the actual source and tag can be retrieved by:

```
In C:
    source = status.MPI_SOURCE
    tag = status.MPI_TAG
In Fortran:
    source = status(MPI_SOURCE)
    tag = status(MPI_TAG)
```

The error field of MPI_Status is never modified. The success or failure is indicated by the return code only.

When one of the MPI wait or test calls returns **status** for a nonblocking operation request and the corresponding blocking operation does not provide a **status** argument, the **status** from this wait/test does not contain meaningful source, tag or message size information.

When you use this routine in a threaded application, make sure that the wait for a given request is done on only one thread. The wait does not have to be done on the thread that created the request. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid request handle

Truncation occurred

MPI not initialized

MPI already finalized

Develop mode error if:

Illegal buffer update

Related Information

MPI_WAITALL
MPI_WAITSOME
MPI_WAITANY
MPI_TEST

MPI_WAITALL, MPI_Waitall

Purpose

Waits for a collection of nonblocking operations to complete.

C Synopsis

```
#include <mpi.h>
int MPI_Waitall(int count,MPI_Request *array_of_requests,
    MPI_Status *array_of_statuses);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_WAITALL(INTEGER COUNT,INTEGER ARRAY_OF_ REQUESTS(*),
      INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*),INTEGER IERROR)
```

Parameters

count	is the lists length (integer) (IN)
array_of_requests	is an array of requests of length **count** (array of handles) (INOUT)
array_of_statuses	is an array of status objects of length **count** (array of status) (OUT). Note that in Fortran a status object is itself an array.
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine blocks until all operations associated with active handles in the list complete, and returns the status of each operation. **array_of_requests** and **array_of statuses** contain **count** entries.

The **i**th entry in **array_of_statuses** is set to the return status of the **i**th operation. Requests created by nonblocking operations are deallocated and the corresponding handles in the array are set to MPI_REQUEST_NULL. If **array_of_requests** contains null or inactive handles, MPI_WAITALL sets the status of each one to **empty**.

MPI_WAITALL(**count, array_of_requests, array_of_statuses**) has the same effect as the execution of MPI_WAIT(**array_of_requests[i], array_of_statuses[i]**) for **i** = 0, 1, ..., **count-1**, in some arbitrary order. MPI_WAITALL with an array of length one is equivalent to MPI_WAIT.

The error fields are never modified unless the function gives a return code of MPI_ERR_IN_STATUS. In which case, the error field of every MPI_Status is modified to reflect the result of the corresponding request.

When you use this routine in a threaded application, make sure that the wait for a given request is done on only one thread. The wait does not have to be done on the thread that created it. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Errors

Invalid count	**count** <0
Invalid request array	
Invalid request(s)	
Truncation occurred	
MPI not initialized	
MPI already finalized	

Related Information

MPI_WAIT
MPI_TESTALL

MPI_WAITANY, MPI_Waitany

Purpose

Waits for any specified nonblocking operation to complete.

C Synopsis

```
#include <mpi.h>
int MPI_Waitany(int count,MPI_Request *array_of_requests,
    int *index,MPI_Status *status);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_WAITANY(INTEGER COUNT,INTEGER ARRAY_OF_REQUESTS(*),INTEGER INDEX,
      INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
```

Parameters

count	is the list length (integer) (IN)
array_of_requests	is the array of requests (array of handles) (INOUT)
index	is the index of the handle for the operation that completed (integer) (OUT)
status	status object (status) (OUT). Note that in Fortran a single status object is an array of integers.
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine blocks until one of the operations associated with the active requests in the array has completed. If more than one operation can complete, one is arbitrarily chosen. MPI_WAITANY returns in **index** the index of that request in the array, and in **status** the status of the completed operation. When the request is allocated by a nonblocking operation, it is deallocated and the request handle is set to MPI_REQUEST_NULL.

The **array_of_requests** list can contain null or inactive handles. When the list has a length of zero or all entries are null or inactive, the call returns immediately with **index** = MPI_UNDEFINED, and an empty status.

MPI_WAITANY(**count, array_of_requests, index, status**) has the same effect as the execution of MPI_WAIT(**array_of_requests[i], status**), where **i** is the value returned by **index**. MPI_WAITANY with an array containing one active entry is equivalent to MPI_WAIT.

The error fields are never modified unless the function gives a return code of MPI_ERR_IN_STATUS. In which case, the error field of every MPI_Status is modified to reflect the result of the corresponding request.

When one of the MPI wait or test calls returns **status** for a nonblocking operation request and the corresponding blocking operation does not provide a **status** argument, the **status** from this wait/test does not contain meaningful source, tag or message size information.

When you use this routine in a threaded application, make sure that the wait for a given request is done on only one thread. The wait does not have to be done on the thread that created it. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Notes

In C, the array is indexed from zero and in Fortran from one.

Errors

Invalid count	**count** < 0
Invalid requests array	
Invalid request(s)	
Truncation occurred	
MPI not initialized	
MPI already finalized	

Related Information

MPI_WAIT
MPI_TESTANY

MPI_WAITSOME, MPI_Waitsome

Purpose

Waits for at least one of a list of nonblocking operations to complete.

C Synopsis

```
#include <mpi.h>
int MPI_Waitsome(int incount,MPI_Request *array_of_requests,
    int *outcount,int *array_of_indices,MPI_Status *array_of_statuses);
```

Fortran Synopsis

```
include 'mpif.h'
MPI_WAITSOME(INTEGER INCOUNT,INTEGER ARRAY_OF_REQUESTS,INTEGER OUTCOUNT,
      INTEGER ARRAY_OF_INDICES(*),INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*),
      INTEGER IERROR)
```

Parameters

incount	is the length of **array_of_requests**, **array_of_indices**, and **array_of_statuses** (integer) (IN)
array_of_requests	is an array of requests (array of handles) (INOUT)
outcount	is the number of completed requests (integer) (OUT)
array_of_indices	is the array of indices of operations that completed (array of integers) (OUT)
array_of_statuses	is the array of status objects for operations that completed (array of status) (OUT). Note that in Fortran a status object is itself an array.
IERROR	is the Fortran return code. It is always the last argument.

Description

This routine waits for at least one of a list of nonblocking operations associated with active handles in the list to complete. The number of completed requests from the list of **array_of_requests** is returned in **outcount**. Returns in the first **outcount** locations of the array **array_of_indices** the indices of these operations.

The status for the completed operations is returned in the first **outcount** locations of the array **array_of_statuses**. When a completed request is allocated by a nonblocking operation, it is deallocated and the associated handle is set to MPI_REQUEST_NULL.

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

When a request for a receive repeatedly appears in a list of requests passed to MPI_WAITSOME and a matching send was posted, then the receive eventually succeeds unless the send is satisfied by another receive. This fairness requirement also applies to send requests and to I/O requests.

The error fields are never modified unless the function gives a return code of MPI_ERR_IN_STATUS. In which case, the error field of every MPI_Status is modified to reflect the result of the corresponding request.

When one of the MPI wait or test calls returns **status** for a nonblocking operation request and the corresponding blocking operation does not provide a **status** argument, the **status** from this wait/test does not contain meaningful source, tag or message size information.

When you use this routine in a threaded application, make sure that the wait for a given request is done on only one thread. The wait does not have to be done on the thread that created it. See Appendix G, “Programming Considerations for User Applications in POE” on page 411 for more information on programming with MPI in a threaded environment.

Notes

In C, the index within the array **array_of_requests**, is indexed from zero and from one in Fortran.

Errors

Invalid count	**count** <0
Invalid request(s)	
Invalid index array	
Truncation occurred	
MPI not initialized	
MPI already finalized	

Related Information

MPI_WAIT
MPI_TESTSOME

MPI_WTICK, MPI_Wtick

Purpose

Returns the resolution of MPI_WTIME in seconds.

C Synopsis

```
#include <mpi.h>
double MPI_Wtick(void);
```

Fortran Synopsis

```
include 'mpif.h'
DOUBLE PRECISION MPI_WTICK()
```

Parameters

None.

Description

This routine returns the resolution of MPI_WTIME in seconds, the time in seconds between successive clock ticks.

Errors

MPI not initialized

MPI already finalized

Related Information

MPI_WTIME

MPI_WTIME, MPI_Wtime

Purpose

Returns the current value of *time* as a floating-point value.

C Synopsis

```
#include <mpi.h>
double MPI_Wtime(void);
```

Fortran Synopsis

```
include 'mpif.h'
DOUBLE PRECISION MPI_WTIME()
```

Parameters

None.

Description

This routine returns the current value of **time** as a double precision floating point number of seconds. This value represents elapsed time since some point in the past. This time in the past will not change during the life of the task. You are responsible for converting the number of seconds into other units if you prefer.

Notes

You can use the attribute key MPI_WTIME_IS_GLOBAL to determine if the values returned by MPI_WTIME on different nodes are synchronized. See MPI_ATTR_GET for more information.

| The environment variable MP_CLOCK_SOURCE allows you to control where
| MPI_WTIME gets its time values from. See "Using the SP Switch Clock as a Time
| Source" on page 420 for more information.

Errors

MPI not initialized

MPI already finalized

Related Information

MPI_WTICK
MPI_ATTR_GET

Appendix A. MPI Subroutine Bindings: Quick Reference

The tables in this appendix summarize the C and FORTRAN binding information for all of the subroutines listed in this book.

Note: FORTRAN refers to FORTRAN 77 bindings which are officially supported for MPI. However, FORTRAN 77 bindings can be used by Fortran 90. Fortran 90 and High Performance Fortran (HPF) offer array section and assumed shape arrays as parameters on calls. These are not safe with MPI.

Bindings for Nonblocking Collective Communication

Table 6 lists the C and FORTRAN bindings for nonblocking collective communication routines.

Table 6 (Page 1 of 3). Bindings for Nonblocking Collective Communication

C/FORTRAN Subroutine	**C/FORTRAN Binding**
MPE_Ibarrier	int MPE_Ibarrier(*MPI_Comm comm,MPI_Request *request*);
MPE_IBARRIER	MPE_IBARRIER(*INTEGER COMM,INTEGER REQUEST,INTEGER IERROR*)
MPE_Ibcast	int MPE_Ibcast(*void* buffer,int count,MPI_Datatype datatype,int root,MPI_Comm comm,MPI_Request *request*);
MPE_IBCAST	MPE_IBCAST(*CHOICE BUFFER,INTEGER COUNT,INTEGER DATATYPE,INTEGER ROOT,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR*)
MPE_Igather	int MPE_Igather(*void* sendbuf,int sendcount,MPI_Datatype sendtype,void* recvbuf,int recvcount,MPI_Datatype recvtype,int root, MPI_Comm comm,MPI_Request *request*);
MPE_IGATHER	MPE_IGATHER(*CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER ROOT,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR*)
MPE_Igatherv	int MPE_Igatherv(*void* sendbuf,int sendcount,MPI_Datatype sendtype,void* recvbuf,int *recvcounts,int *displs,MPI_Datatype recvtype,int root,MPI_Comm comm,MPI_Request *request*);
MPE_IGATHERV	MPE_IGATHERV(*CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE, CHOICE RECVBUF,INTEGER RECVCOUNTS(*),INTEGER DISPLS(*),INTEGER RECVTYPE,INTEGER ROOT,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR*)

Table 6 (Page 2 of 3). Bindings for Nonblocking Collective Communication

C/FORTRAN Subroutine	**C/FORTRAN Binding**
MPE_Iscatter	int MPE_Iscatter(*void* sendbuf,int sendcount,MPI_Datatype sendtype,void* recvbuf,int recvcount,MPI_Datatype recvtype,int root,MPI_Comm comm,MPI_Request *request*);
MPE_ISCATTER	MPE_ISCATTER(*CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER ROOT,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR*)
MPE_Iscatterv	int MPE_Iscatterv(*void* sendbuf,int *sendcounts,int *displs,MPI_Datatype sendtype,void* recvbuf,int recvcount,MPI_Datatype recvtype,int root,MPI_Comm comm,MPI_Request *request*);
MPE_ISCATTERV	MPE_ISCATTERV(*CHOICE SENDBUF,INTEGER SENDCOUNTS(*),INTEGER DISPLS(*),INTEGER SENDTYPE,CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER ROOT,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR*)
MPE_Iallgather	int MPE_Iallgather(*void* sendbuf,int sendcount,MPI_Datatype sendtype,void* recvbuf,int recvcount,MPI_Datatype recvtype, MPI_Comm comm,MPI_Request *request*);
MPE_IALLGATHER	MPE_IALLGATHER(*CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE, CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR*)
MPE_Iallgatherv	int MPE_Iallgatherv(*void* sendbuf,int sendcount,MPI_Datatype sendtype,void* recvbuf,int *recvcounts,int *displs,MPI_Datatype recvtype,MPI_Comm comm,MPI_Request *request*);
MPE_IALLGATHERV	MPE_IALLGATHERV(*CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE, CHOICE RECVBUF,INTEGER RECVCOUNTS(*),INTEGER DISPLS(*),INTEGER RECVTYPE,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR*)
MPE_Ialltoall	int MPE_Ialltoall(*void* sendbuf,int sendcount,MPI_Datatype sendtype,void* recvbuf,int recvcount,MPI_Datatype recvtype,MPI_Comm comm,MPI_Request *request*);
MPE_IALLTOALL	MPE_IALLTOALL(*CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR*)
MPE_Ialltoallv	int MPE_Ialltoallv(*void* sendbuf,int *sendcounts,int *sdispls,MPI_Datatype sendtype,void* recvbuf,int *recvcounts,int *rdispls,MPI_Datatype recvtype,MPI_Comm comm,MPI_Request *request*);
MPE_IALLTOALLV	MPE_IALLTOALV(*CHOICE SENDBUF,INTEGER SENDCOUNTS(*),INTEGER SDISPLS(*),INTEGER SENDTYPE,CHOICE RECVBUF,INTEGER RECVCOUNTS(*),INTEGER RDISPLS(*),INTEGER RECVTYPE,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR*)
MPE_Ireduce	int MPE_Ireduce(*void* sendbuf,void* recvbuf,int count,MPI_Datatype datatype,MPI_Op op,int root,MPI_Comm comm,MPI_Request *request*);
MPE_IREDUCE	MPE_IREDUCE(*CHOICE SENDBUF,CHOICE RECVBUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER OP,INTEGER ROOT,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR*)
MPE_Iallreduce	int MPE_Iallreduce(*void* sendbuf,void* recvbuf,int count,MPI_Datatype datatype,MPI_Op op,MPI_Comm comm,MPI_Request *request*);

Table 6 (Page 3 of 3). Bindings for Nonblocking Collective Communication

C/FORTRAN Subroutine	**C/FORTRAN Binding**
MPE_IALLREDUCE	MPE_IALLREDUCE(*CHOICE SENDBUF,CHOICE RECVBUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER OP,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR*)
MPE_Ireduce_scatter	int MPE_Ireduce_scatter(*void* sendbuf,void* recvbuf,int *recvcounts,MPI_Datatype datatype,MPI_Op op,MPI_Comm comm,MPI_Request *request*);
MPE_IREDUCE_SCATTER	MPE_IREDUCE_SCATTER(*CHOICE SENDBUF,CHOICE RECVBUF,INTEGER RECVCOUNTS(*),INTEGER DATATYPE,INTEGER OP,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR*)
MPE_Iscan	int MPE_Iscan(*void* sendbuf,void* recvbuf,int count,MPI_Datatype datatype,MPI_Op op,MPI_Comm comm,MPI_Request *request*);
MPE_ISCAN	MPE_ISCAN(*CHOICE SENDBUF,CHOICE RECVBUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER OP,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR*)

Bindings for Point-to-Point Communication and Derived Datatypes

Table 7 lists the C and FORTRAN bindings for point-to-point communication and derived datatype routines.

Table 7 (Page 1 of 7). Bindings for Point-to-Point Communication and Derived Datatypes

C/FORTRAN Subroutine	**C/FORTRAN Binding**
MPI_Send	int MPI_Send(*void* buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm*);
MPI_SEND	MPI_SEND(*CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG,INTEGER COMM, INTEGER IERROR*)
MPI_Recv	int MPI_Recv(*void* buf,int count,MPI_Datatype datatype,int source,int tag, MPI_Comm comm, MPI_Status *status*);
MPI_RECV	MPI_RECV(*CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER SOURCE, INTEGER TAG,INTEGER COMM,INTEGER STATUS(MPI_STATUS_SIZE),,INTEGER IERROR*)
MPI_Get_count	int MPI_Get_count(*MPI_Status *status, MPI_Datatype datatype, int *count*);
MPI_GET_COUNT	MPI_GET_COUNT(*INTEGER STATUS(MPI_STATUS_SIZE),,INTEGER DATATYPE,INTEGER COUNT, INTEGER IERROR*)
MPI_Bsend	int MPI_Bsend(*void* buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm*);
MPI_BSEND	MPI_BSEND(*CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST, INTEGER TAG,INTEGER COMM,INTEGER IERROR*)
MPI_Ssend	int MPI_Ssend(*void* buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm*);
MPI_SSEND	MPI_SSEND(*CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG,INTEGER COMM,INTEGER IERROR*)

Table 7 (Page 2 of 7). Bindings for Point-to-Point Communication and Derived Datatypes

C/FORTRAN Subroutine	**C/FORTRAN Binding**
MPI_Rsend	int MPI_Rsend(*void* buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm*);
MPI_RSEND	MPI_RSEND(*CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG,INTEGER COMM,INTEGER IERROR*)
MPI_Buffer_attach	int MPI_Buffer_attach(*void* buffer,int size*);
MPI_BUFFER_ATTACH	MPI_BUFFER_ATTACH(*CHOICE BUFFER,INTEGER SIZE,INTEGER IERROR*)
MPI_Buffer_detach	int MPI_Buffer_detach(*void* buffer,int* size*);
MPI_BUFFER_DETACH	MPI_BUFFER_DETACH(*CHOICE BUFFER,INTEGER SIZE,INTEGER IERROR*)
MPI_Isend	int MPI_Isend(*void* buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request*);
MPI_ISEND	MPI_ISEND(*CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR*)
MPI_Ibsend	int MPI_Ibsend(*void* buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request*);
MPI_IBSEND	MPI_IBSEND(*CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR*)
MPI_Issend	int MPI_Issend(*void* buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request*);
MPI_ISSEND	MPI_ISSEND(*CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR*)
MPI_Irsend	int MPI_Irsend(*void* buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request*);
MPI_IRSEND	MPI_IRSEND(*CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR*)
MPI_Irecv	int MPI_Irecv(*void* buf,int count,MPI_Datatype datatype,int source,int tag,MPI_Comm comm,MPI_Request *request*);
MPI_IRECV	MPI_IRECV(*CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER SOURCE,INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR*)
MPI_Wait	int MPI_Wait(*MPI_Request *request,MPI_Status *status*);
MPI_WAIT	MPI_WAIT(*INTEGER REQUEST,INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR*)
MPI_Test	int MPI_Test(*MPI_Request *request,int *flag,MPI_Status *status*);
MPI_TEST	MPI_TEST(*INTEGER REQUEST,INTEGER FLAG,INTEGER STATUS(MPI_STATUS_SIZE), INTEGER IERROR*)
MPI_Request_free	int MPI_Request_free(*MPI_Request *request*);
MPI_REQUEST_FREE	MPI_REQUEST_FREE(*INTEGER REQUEST,INTEGER IERROR*)

Table 7 (Page 3 of 7). Bindings for Point-to-Point Communication and Derived Datatypes

C/FORTRAN Subroutine	**C/FORTRAN Binding**
MPI_Waitany	int MPI_Waitany(*int count,MPI_Request *array_of_requests,int *index,MPI_Status *status*);
MPI_WAITANY	MPI_WAITANY(*INTEGER COUNT,INTEGER ARRAY_OF_REQUESTS(*),INTEGER INDEX, INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR*)
MPI_Testany	int MPI_Testany(*int count, MPI_Request *array_of_requests, int *index, int *flag,MPI_Status *status*);
MPI_TESTANY	MPI_TESTANY(*INTEGER COUNT,INTEGER ARRAY_OF_REQUESTS(*),INTEGER INDEX,INTEGER FLAG,INTEGER STATUS(MPI_STATUS_SIZE), INTEGER IERROR*)
MPI_Waitall	int MPI_Waitall(*int count,MPI_Request *array_of_requests,MPI_Status *array_of_statuses*);
MPI_WAITALL	MPI_WAITALL(*INTEGER COUNT,INTEGER ARRAY_OF_ REQUESTS(*),INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), INTEGER IERROR*)
MPI_Testall	int MPI_Testall(*int count,MPI_Request *array_of_requests,int *flag,MPI_Status *array_of_statuses*);
MPI_TESTALL	MPI_TESTALL(*INTEGER COUNT,INTEGER ARRAY_OF_REQUESTS(*),INTEGER FLAG, INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*),INTEGER IERROR*)
MPI_Waitsome	int MPI_Waitsome(*int incount,MPI_Request *array_of_requests,int *outcount,int *array_of_indices,MPI_Status *array_of_statuses*);
MPI_WAITSOME	MPI_WAITSOME(*INTEGER INCOUNT,INTEGER ARRAY_OF_REQUESTS,INTEGER OUTCOUNT,INTEGER ARRAY_OF_INDICES(*),INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE),*),INTEGER IERROR*)
MPI_Testsome	int MPI_Testsome(*int incount,MPI_Request *array_of_requests,int *outcount,int *array_of_indices,MPI_Status *array_of_statuses*);
MPI_TESTSOME	MPI_TESTSOME(*INTEGER INCOUNT,INTEGER ARRAY_OF_REQUESTS(*),INTEGER OUTCOUNT,INTEGER ARRAY_OF_INDICES(*),INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE),*),INTEGER IERROR*)
MPI_Iprobe	int MPI_Iprobe(*int source,int tag,MPI_Comm comm,int *flag,MPI_Status *status*);
MPI_IPROBE	MPI_IPROBE(*INTEGER SOURCE,INTEGER TAG,INTEGER COMM,INTEGER FLAG,INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR*)
MPI_Probe	int MPI_Probe(*int source,int tag,MPI_Comm comm,MPI_Status *status*);
MPI_PROBE	MPI_PROBE(*INTEGER SOURCE,INTEGER TAG,INTEGER COMM,INTEGER STATUS(MPI_STATUS_SIZE), INTEGER IERROR*)
MPI_Cancel	int MPI_Cancel(*MPI_Request *request*);
MPI_CANCEL	MPI_CANCEL(*INTEGER REQUEST,INTEGER IERROR*)
MPI_Test_cancelled	int MPI_Test_cancelled(*MPI_Status *status,int *flag*);
MPI_TEST_CANCELLED	MPI_TEST_CANCELLED(*INTEGER STATUS(MPI_STATUS_SIZE),INTEGER FLAG,INTEGER IERROR*)

Table 7 (Page 4 of 7). Bindings for Point-to-Point Communication and Derived Datatypes

C/FORTRAN Subroutine	**C/FORTRAN Binding**
MPI_Send_init	int MPI_Send_init(*void* buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request*);
MPI_SEND_INIT	MPI_SEND_INIT(*CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR*)
MPI_Bsend_init	int MPI_Bsend_init(*void* buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request*);
MPI_BSEND_INIT	MPI_SEND_INIT(*CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR*)
MPI_Ssend_init	int MPI_Ssend_init(*void* buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request*);
MPI_SSEND_INIT	MPI_SSEND_INIT(*CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG,INTEGER COMM,INTEGER REQUEST,IERROR*)
MPI_Rsend_init	int MPI_Rsend_init(*void* buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request*);
MPI_RSEND_INIT	MPI_RSEND_INIT(*CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR*)
MPI_Recv_init	int MPI_Recv_init(*void* buf,int count,MPI_Datatype datatype,int source,int tag,MPI_Comm comm,MPI_Request *request*);
MPI_RECV_INIT	MPI_RECV_INIT(*CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER SOURCE,INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR*)
MPI_Start	int MPI_Start(*MPI_Request *request*);
MPI_START	MPI_START(*INTEGER REQUEST,INTEGER IERROR*)
MPI_Startall	int MPI_Startall(*int count,MPI_Request *array_of_requests*);
MPI_STARTALL	MPI_STARTALL(*INTEGER COUNT,INTEGER ARRAY_OF_REQUESTS(*),INTEGER IERROR*)
MPI_Sendrecv	int MPI_Sendrecv(*void *sendbuf,int sendcount,MPI_Datatype sendtype,int dest,int sendtag,void *recvbuf,int recvcount, MPI_Datatype recvtype,int source,int recvtag,MPI_Comm comm,MPI_Status *status*);
MPI_SENDRECV	MPI_SENDRECV(*CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,INTEGER DEST,INTEGER SENDTAG,CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER SOURCE,INTEGER RECVTAG,INTEGER COMM,INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR*)
MPI_Sendrecv_replace	int MPI_Sendrecv_replace(*void* buf,int count,MPI_Datatype datatype,int dest,int sendtag,int source,int recvtag,MPI_Comm comm,MPI_Status *status*);
MPI_SENDRECV_REPLACE	MPI_SENDRECV_REPLACE(*CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER SENDTAG,INTEGER SOURCE,INTEGER RECVTAG,INTEGER COMM,INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR*)
MPI_Type_contiguous	int MPI_Type_contiguous(*int count,MPI_Datatype oldtype,MPI_Datatype *newtype*);

Table 7 (Page 5 of 7). Bindings for Point-to-Point Communication and Derived Datatypes

C/FORTRAN Subroutine	**C/FORTRAN Binding**
MPI_TYPE_CONTIGUOUS	MPI_TYPE_CONTIGUOUS(*INTEGER COUNT,INTEGER OLDTYPE,INTEGER NEWTYPE,INTEGER IERROR*)
MPI_Type_create_darray	int MPI_Type_create_darray (*int size,int rank,int ndims, int array_of_gsizes[],int array_of_distribs[], int array_of_dargs[],int array_of_psizes[], int order,MPI_Datatype oldtype,MPI_Datatype *newtype*);
MPI_TYPE_CREATE_DARRAY	MPI_TYPE_CREATE_DARRAY (*INTEGER SIZE,INTEGER RANK,INTEGER NDIMS, INTEGER ARRAY_OF_GSIZES(*),INTEGER ARRAY_OF_DISTRIBS(*), INTEGER ARRAY_OF_DARGS(*),INTEGER ARRAY_OF_PSIZES(*), INTEGER ORDER,INTEGER OLDTYPE,INTEGER NEWTYPE,INTEGER IERROR*)
MPI_Type_create_subarray	int MPI_Type_create_subarray (*int ndims,int array_of_sizes[], int array_of_subsizes[],int array_of_starts[] , int order,MPI_Datatype oldtype,MPI_Datatype *newtype*);
MPI_TYPE_CREATE_SUBARRAY	MPI_TYPE_CREATE_SUBARRAY (*INTEGER NDIMS,INTEGER ARRAY_OF_SUBSIZES(*), INTEGER ARRAY_OF_SIZES(*),INTEGER ARRAY_OF_STARTS(*), INTEGER ORDER,INTEGER OLDTYPE,INTEGER NEWTYPE,INTEGER IERROR*)
MPI_Type_get_contents	int MPI_Type_get_contents(MPI_Datatype datatype, int *max_integers, int *max_addresses, int *max_datatypes, int array_of_integers[], int array_of_addresses[], int array_of_datatypes[]);
MPI_TYPE_GET_CONTENTS	MPI_TYPE_GET_CONTENTS(*INTEGER DATATYPE, INTEGER MAX_INTEGERS, INTEGER MAX_ADDRESSES, INTEGER MAX_DATATYPES, INTEGER ARRAY_of_INTEGERS, INTEGER ARRAY_OF_ADDRESSES, INTEGER ARRAY_of_DATATYPES, INTEGER IERROR*)
MPI_Type_get_envelope	int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers, int *num_addresses, int *num_datatypes, int *combiner);
MPI_TYPE_GET_ENVELOPE	MPI_TYPE_GET_ENVELOPE(INTEGER DATATYPE, INTEGER NUM_INTEGERS, INTEGER NUM_ADDRESSES, INTEGER NUM_DATATYPES, INTEGER COMBINER, INTEGER IERROR)
MPI_Type_vector	int MPI_Type_vector(*int count,int blocklength,int stride,MPI_Datatype oldtype,MPI_Datatype *newtype*);
MPI_TYPE_VECTOR	MPI_TYPE_VECTOR(*INTEGER COUNT,INTEGER BLOCKLENGTH,INTEGER STRIDE,INTEGER OLDTYPE,INTEGER NEWTYPE,INTEGER IERROR*)
MPI_Type_hvector	int MPI_Type_hvector(*int count,int blocklength,MPI_Aint stride,MPI_Datatype oldtype,MPI_Datatype *newtype*);
MPI_TYPE_HVECTOR	MPI_TYPE_HVECTOR(*INTEGER COUNT,INTEGER BLOCKLENGTH,INTEGER STRIDE,INTEGER OLDTYPE,INTEGER NEWTYPE,INTEGER IERROR*)
MPI_Type_indexed	int MPI_Type_indexed(*int count,int *array_of_blocklengths,int *array_of_displacements,MPI_Datatype oldtype, MPI_Datatype *newtype*);
MPI_TYPE_INDEXED	MPI_TYPE_INDEXED(*INTEGER COUNT, INTEGER ARRAY_OF_BLOCKLENGTHS(*), INTEGER ARRAY_OF DISPLACEMENTS(*),INTEGER OLDTYPE,INTEGER NEWTYPE,INTEGER IERROR*)
MPI_Type_hindexed	int MPI_Type_hindexed(*int count,int *array_of_blocklengths,MPI_Aint *array_of_displacements,MPI_Datatype oldtype, MPI_Datatype *newtype*);

Table 7 (Page 6 of 7). Bindings for Point-to-Point Communication and Derived Datatypes

C/FORTRAN Subroutine	**C/FORTRAN Binding**
MPI_TYPE_HINDEXED	MPI_TYPE_HINDEXED(*INTEGER COUNT,INTEGER ARRAY_OF_BLOCKLENGTHS(*),INTEGER ARRAY_OF DISPLACEMENTS(*),INTEGER OLDTYPE,INTEGER NEWTYPE,INTEGER IERROR*)
MPI_Type_struct	int MPI_Type_struct(*int count,int *array_of_blocklengths, MPI_Aint *array_of_displacements,MPI_Datatype *array_of_types, MPI_Datatype *newtype*);
MPI_TYPE_STRUCT	MPI_TYPE_STRUCT(*INTEGER COUNT,INTEGER ARRAY_OF_BLOCKLENGTHS(*),INTEGER ARRAY_OF DISPLACEMENTS(*),INTEGER ARRAY_OF_TYPES(*),INTEGER NEWTYPE,INTEGER IERROR*)
MPI_Address	int MPI_Address(*void* location,MPI_Aint *address*);
MPI_ADDRESS	MPI_ADDRESS(*CHOICE LOCATION,INTEGER ADDRESS,INTEGER IERROR*)
MPI_Type_extent	int MPI_Type_extent(*MPI_Datatype datatype,int *extent*);
MPI_TYPE_EXTENT	MPI_TYPE_EXTENT(*INTEGER DATATYPE,INTEGER EXTENT,INTEGER IERROR*)
MPI_Type_size	int MPI_Type_size(*MPI_Datatype datatype,int *size*);
MPI_TYPE_SIZE	MPI_TYPE_SIZE(*INTEGER DATATYPE,INTEGER SIZE,INTEGER IERROR*)
MPI_Type_lb	int MPI_Type_lb(*MPI_Datatype datatype,int* displacement*);
MPI_TYPE_LB	MPI_TYPE_LB(*INTEGER DATATYPE,INTEGER DISPLACEMENT,INTEGER IERROR*)
MPI_Type_ub	int MPI_Type_ub(*MPI_Datatype datatype,int* displacement*);
MPI_TYPE_UB	MPI_TYPE_UB(*INTEGER DATATYPE,INTEGER DISPLACEMENT,INTEGER IERROR*)
MPI_Type_commit	int MPI_Type_commit(*MPI_Datatype *datatype*);
MPI_TYPE_COMMIT	MPI_TYPE_COMMIT(*INTEGER DATATYPE,INTEGER IERROR*)
MPI_Type_free	int MPI_Type_free(*MPI_Datatype *datatype*);
MPI_TYPE_FREE	MPI_TYPE_FREE(*INTEGER DATATYPE,INTEGER IERROR*)
MPI_Get_elements	int MPI_Get_elements(*MPI_Status *status,MPI_Datatype datatype,int *count*);
MPI_GET_ELEMENTS	MPI_GET_ELEMENTS(*INTEGER STATUS(MPI_STATUS_SIZE),INTEGER DATATYPE,INTEGER COUNT,INTEGER IERROR*)
MPI_Pack	int MPI_Pack(*void* inbuf,int incount,MPI_Datatype datatype,void *outbuf,int outsize,int *position,MPI_Comm comm*);
MPI_PACK	MPI_PACK(*CHOICE INBUF,INTEGER INCOUNT,INTEGER DATATYPE,CHOICE OUTBUF,INTEGER OUTSIZE,INTEGER POSITION,INTEGER COMM,INTEGER IERROR*)
MPI_Unpack	int MPI_Unpack(*void* inbuf,int insize,int *position,void *outbuf,int outcount,MPI_Datatype datatype,MPI_Comm comm*);
MPI_UNPACK	MPI_UNPACK(*CHOICE INBUF,INTEGER INSIZE,INTEGER POSITION,CHOICE OUTBUF,INTEGER OUTCOUNT,INTEGER DATATYPE,INTEGER COMM, INTEGER IERRROR*)

Table 7 (Page 7 of 7). Bindings for Point-to-Point Communication and Derived Datatypes

C/FORTRAN Subroutine	C/FORTRAN Binding
MPI_Pack_size	int MPI_Pack_size(*int incount,MPI_Datatype datatype,MPI_Comm comm,int *size*);
MPI_PACK_SIZE	MPI_PACK_SIZE(*INTEGER INCOUNT,INTEGER DATATYPE,INTEGER COMM,INTEGER SIZE,INTEGER IERROR*)

Bindings for Collective Communication

Table 8 lists the C and FORTRAN bindings for collective communication routines.

Table 8 (Page 1 of 3). Bindings for Collective Communication

C/FORTRAN Subroutine	C/FORTRAN Binding
MPI_Barrier	int MPI_Barrier(*MPI_Comm comm*);
MPI_BARRIER	MPI_BARRIER(*INTEGER COMM,INTEGER IERROR*)
MPI_Bcast	int MPI_Bcast(*void* buffer,int count,MPI_Datatype datatype,int root,MPI_Comm comm*);
MPI_BCAST	MPI_BCAST(*CHOICE BUFFER,INTEGER COUNT,INTEGER DATATYPE,INTEGER ROOT,INTEGER COMM,INTEGER IERROR*)
MPI_Gather	int MPI_Gather(*void* sendbuf,int sendcount,MPI_Datatype sendtype,void* recvbuf,int recvcount,MPI_Datatype recvtype,int root,MPI_Comm comm*);
MPI_GATHER	MPI_GATHER(*CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER ROOT,INTEGER COMM,INTEGER IERROR*)
MPI_Gatherv	int MPI_Gatherv(*void* sendbuf,int sendcount,MPI_Datatype sendtype,void* recvbuf,int *recvcounts,int *displs,MPI_Datatype recvtype,int root,MPI_Comm comm*);
MPI_GATHERV	MPI_GATHERV(*CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,CHOICE RECVBUF,INTEGER RECVCOUNTS(*),INTEGER DISPLS(*),INTEGER RECVTYPE,INTEGER ROOT,INTEGER COMM,INTEGER IERROR*)
MPI_Scatter	int MPI_Scatter(*void* sendbuf,int sendcount,MPI_Datatype sendtype,void* recvbuf,int recvcount,MPI_Datatype recvtype,int root MPI_Comm comm*);
MPI_SCATTER	MPI_SCATTER(*CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER ROOT,INTEGER COMM,INTEGER IERROR*)
MPI_Scatterv	int MPI_Scatterv(*void* sendbuf,int *sendcounts,int *displs,MPI_Datatype sendtype,void* recvbuf,int recvcount,MPI_Datatype recvtype,int root,MPI_Comm comm*);
MPI_SCATTERV	MPI_SCATTERV(*CHOICE SENDBUF,INTEGER SENDCOUNTS(*),INTEGER DISPLS(*),INTEGER SENDTYPE,CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER ROOT,INTEGER COMM,INTEGER IERROR*)
MPI_Allgather	int MPI_Allgather(*void* sendbuf,int sendcount,MPI_Datatype sendtype,void* recvbuf,int recvcount,MPI_Datatype recvtype, MPI_Comm comm*);

Table 8 (Page 2 of 3). Bindings for Collective Communication

C/FORTRAN Subroutine	**C/FORTRAN Binding**
MPI_ALLGATHER	MPI_ALLGATHER(*CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER COMM,INTEGER IERROR*)
MPI_Allgatherv	int MPI_Allgatherv(*void* sendbuf,int sendcount,MPI_Datatype sendtype,void* recvbuf,int *recvcounts,int *displs, MPI_Datatype recvtype,MPI_Comm comm*);
MPI_ALLGATHERV	MPI_ALLGATHERV(*CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,CHOICE RECVBUF,INTEGER RECVCOUNTS(*),INTEGER DISPLS(*),INTEGER RECVTYPE,INTEGER COMM,INTEGER IERROR*)
MPI_Alltoall	int MPI_Alltoall(*void* sendbuf,int sendcount,MPI_Datatype sendtype,void* recvbuf,int recvcount,MPI_Datatype recvtype, MPI_Comm comm*);
MPI_ALLTOALL	MPI_ALLTOALL(*CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER COMM,INTEGER IERROR*)
MPI_Alltoallv	int MPI_Alltoallv(*void* sendbuf,int *sendcounts,int *sdispls,MPI_Datatype sendtype,void* recvbuf,int *recvcounts,int *rdispls,MPI_Datatype recvtype,MPI_Comm comm*);
MPI_ALLTOALLV	MPI_ALLTOALLV(*CHOICE SENDBUF,INTEGER SENDCOUNTS(*),INTEGER SDISPLS(*),INTEGER SENDTYPE,CHOICE RECVBUF,INTEGER RECVCOUNTS(*),INTEGER RDISPLS(*),INTEGER RECVTYPE,INTEGER COMM,INTEGER IERROR*)
MPI_Reduce	int MPI_Reduce(*void* sendbuf,void* recvbuf,int count,MPI_Datatype datatype,MPI_Op op,int root,MPI_Comm comm*);
MPI_REDUCE	MPI_REDUCE(*CHOICE SENDBUF,CHOICE RECVBUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER OP,INTEGER ROOT,INTEGER COMM,INTEGER IERROR*)
MPI_Op_create	int MPI_Op_create(*MPI_User_function *function, int commute, MPI_Op *op*);
MPI_OP_CREATE	MPI_OP_CREATE(*EXTERNAL FUNCTION,INTEGER COMMUTE,INTEGER OP,INTEGER IERROR*)
MPI_Op_free	int MPI_Op_free(*MPI_Op *op*);
MPI_OP_FREE	MPI_OP_FREE(*INTEGER OP,INTEGER IERROR*)
MPI_Allreduce	int MPI_Allreduce(*void* sendbuf,void* recvbuf,int count,MPI_Datatype datatype,MPI_Op op,MPI_Comm comm*);
MPI_ALLREDUCE	MPI_ALLREDUCE(*CHOICE SENDBUF,CHOICE RECVBUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER OP,INTEGER COMM,INTEGER IERROR*)
MPI_Reduce_scatter	int MPI_Reduce_scatter(*void* sendbuf,void* recvbuf,int *recvcounts,MPI_Datatype datatype,MPI_Op op,MPI_Comm comm*);
MPI_REDUCE_SCATTER	MPI_REDUCE_SCATTER(*CHOICE SENDBUF,CHOICE RECVBUF,INTEGER RECVCOUNTS(*),INTEGER DATATYPE,INTEGER OP,INTEGER COMM,INTEGER IERROR*)
MPI_Scan	int MPI_Scan(*void* sendbuf,void* recvbuf,int count,MPI_Datatype datatype,MPI_Op op,MPI_Comm comm*);

Table 8 (Page 3 of 3). Bindings for Collective Communication

C/FORTRAN Subroutine	**C/FORTRAN Binding**
MPI_SCAN	MPI_SCAN(*CHOICE SENDBUF,CHOICE RECVBUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER OP,INTEGER COMM,INTEGER IERROR*)

Bindings for Groups and Communicators

Table 9 lists the C and FORTRAN bindings for group and communicator routines.

Table 9 (Page 1 of 3). Bindings for Groups and Communicators

C/FORTRAN Subroutine	**C/FORTRAN Binding**
MPI_Group_size	int MPI_Group_size(*MPI_Group group,int *size*);
MPI_GROUP_SIZE	MPI_GROUP_SIZE(*INTEGER GROUP,INTEGER SIZE,INTEGER IERROR*)
MPI_Group_rank	int MPI_Group_rank(*MPI_Group group,int *rank*);
MPI_GROUP_RANK	MPI_GROUP_RANK(*INTEGER GROUP,INTEGER RANK,INTEGER IERROR*)
MPI_Group_translate_ranks	int MPI_Group_translate_ranks (*MPI_Group group1,int n,int *ranks1,MPI_Group group2,int *ranks2*);
MPI_GROUP_TRANSLATE_RANKS	MPI_GROUP_TRANSLATE_RANKS(*INTEGER GROUP1, INTEGER N,INTEGER RANKS1(*),INTEGER GROUP2,INTEGER RANKS2(*),INTEGER IERROR*)
MPI_Group_compare	int MPI_Group_compare(*MPI_Group group1,MPI_Group group2,int *result*);
MPI_GROUP_COMPARE	MPI_GROUP_COMPARE(*INTEGER GROUP1,INTEGER GROUP2,INTEGER RESULT,INTEGER IERROR*)
MPI_Comm_group	int MPI_Comm_group(*MPI_Comm comm,MPI_Group *group*);
MPI_COMM_GROUP	MPI_COMM_GROUP(*INTEGER COMM,INTEGER GROUP,INTEGER IERROR*)
MPI_Group_union	int MPI_Group_union(*MPI_Group group1,MPI_Group group2,MPI_Group *newgroup*);
MPI_GROUP_UNION	MPI_GROUP_UNION(*INTEGER GROUP1,INTEGER GROUP2,INTEGER NEWGROUP,INTEGER IERROR*)
MPI_Group_intersection	int MPI_Group_intersection(*MPI_Group group1,MPI_Group group2,MPI_Group *newgroup*);
MPI_GROUP_INTERSECTION	MPI_GROUP_INTERSECTION(*INTEGER GROUP1,INTEGER GROUP2,INTEGER NEWGROUP,INTEGER IERROR*)
MPI_Group_difference	int MPI_Group_difference(*MPI_Group group1,MPI_Group group2,MPI_Group *newgroup*);
MPI_GROUP_DIFFERENCE	MPI_GROUP_DIFFERENCE(*INTEGER GROUP1,INTEGER GROUP2,INTEGER NEWGROUP,INTEGER IERROR*)
MPI_Group_incl	int MPI_Group_incl(*MPI_Group group,int n,int *ranks,MPI_Group *newgroup*);
MPI_GROUP_INCL	MPI_GROUP_INCL(*INTEGER GROUP,INTEGER N,INTEGER RANKS(*),INTEGER NEWGROUP,INTEGER IERROR*)

Table 9 (Page 2 of 3). Bindings for Groups and Communicators

C/FORTRAN Subroutine	**C/FORTRAN Binding**
MPI_Group_excl	int MPI_Group_excl(*MPI_Group group,int n,int *ranks,MPI_Group *newgroup*);
MPI_GROUP_EXCL	MPI_GROUP_EXCL(*INTEGER GROUP,INTEGER N,INTEGER RANKS(*),INTEGER NEWGROUP,INTEGER IERROR*)
MPI_Group_range_incl	int MPI_Group_range_incl(*MPI_Group group,int n,int ranges[][3],MPI_Group *newgroup*);
MPI_GROUP_RANGE_INCL	MPI_GROUP_RANGE_INCL(*INTEGER GROUP,INTEGER N,INTEGER RANGES(3,*),INTEGER NEWGROUP,INTEGER IERROR*)
MPI_Group_range_excl	int MPI_Group_range_excl(*MPI_Group group,int n,int ranges [][3],MPI_Group *newgroup*);
MPI_GROUP_RANGE_EXCL	MPI_GROUP_RANGE_EXCL(*INTEGER GROUP,INTEGER N,INTEGER RANGES(3,*),INTEGER NEWGROUP,INTEGER IERROR*)
MPI_Group_free	int MPI_Group_free(*MPI_Group *group*);
MPI_GROUP_FREE	MPI_GROUP_FREE(*INTEGER GROUP,INTEGER IERROR*)
MPI_Comm_size	int MPI_Comm_size(*MPI_Comm comm,int *size*);
MPI_COMM_SIZE	MPI_COMM_SIZE(*INTEGER COMM,INTEGER SIZE,INTEGER IERROR*)
MPI_Comm_rank	int MPI_Comm_rank(*MPI_Comm comm,int *rank*);
MPI_COMM_RANK	MPI_COMM_RANK(*INTEGER COMM,INTEGER RANK,INTEGER IERROR*)
MPI_Comm_compare	int MPI_Comm_compare(*MPI_Comm comm1,MPI_Comm comm2,int *result*);
MPI_COMM_COMPARE	MPI_COMM_COMPARE(*INTEGER COMM1,INTEGER COMM2,INTEGER RESULT,INTEGER IERROR*)
MPI_Comm_dup	int MPI_Comm_dup(*MPI_Comm comm,MPI_Comm *newcomm*);
MPI_COMM_DUP	MPI_COMM_DUP(*INTEGER COMM,INTEGER NEWCOMM,INTEGER IERROR*)
MPI_Comm_create	int MPI_Comm_create(*MPI_Comm comm,MPI_Group group,MPI_Comm *newcomm*);
MPI_COMM_CREATE	MPI_COMM_CREATE(*INTEGER COMM,INTEGER GROUP,INTEGER NEWCOMM,INTEGER IERROR*)
MPI_Comm_split	int MPI_Comm_split(*MPI_Comm comm,int color,int key,MPI_Comm *newcomm*);
MPI_COMM_SPLIT	MPI_COMM_SPLIT(*INTEGER COMM,INTEGER COLOR,INTEGER KEY,INTEGER NEWCOMM,INTEGER IERROR*)
MPI_Comm_free	int MPI_Comm_free(*MPI_Comm *comm*);
MPI_COMM_FREE	MPI_COMM_FREE(*INTEGER COMM,INTEGER IERROR*)
MPI_Comm_test_inter	int MPI_Comm_test_inter(*MPI_Comm comm,int *flag*);
MPI_COMM_TEST_INTER	MPI_COMM_TEST_INTER(*INTEGER COMM,LOGICAL FLAG,INTEGER IERROR*)
MPI_Comm_remote_size	int MPI_Comm_remote_size(*MPI_Comm comm,int *size*);
MPI_COMM_REMOTE_SIZE	MPI_COMM_REMOTE_SIZE(*INTEGER COMM,INTEGER SIZE,INTEGER IERROR*)
MPI_Comm_remote_group	int MPI_Comm_remote_group(*MPI_Comm comm,MPI_Group *group*);

Table 9 (Page 3 of 3). Bindings for Groups and Communicators

C/FORTRAN Subroutine	**C/FORTRAN Binding**
MPI_COMM_REMOTE_GROUP	MPI_COMM_REMOTE_GROUP(*INTEGER COMM,INTEGER GROUP,INTEGER IERROR*)
MPI_Intercomm_create	int MPI_Intercomm_create(*MPI_Comm local_comm,int local_leader,MPI_Comm peer_comm,int remote_leader,int tag, MPI_Comm *newintercomm*);
MPI_INTERCOMM_CREATE	MPI_INTERCOMM_CREATE(*INTEGER LOCAL_COMM,INTEGER LOCAL_LEADER,INTEGER PEER_COMM,INTEGER REMOTE_LEADER,INTEGER TAG,INTEGER NEWINTERCOM,INTEGER IERROR*)
MPI_Intercomm_merge	int MPI_Intercomm_merge(*MPI_Comm intercomm,int high,MPI_Comm *newintracomm*);
MPI_INTERCOMM_MERGE	MPI_INTERCOMM_MERGE(*INTEGER INTERCOMM,INTEGER HIGH,INTEGER NEWINTRACOMM,INTEGER IERROR*)
MPI_Keyval_create	int MPI_Keyval_create(*MPI_Copy_function *copy_fn,MPI_Delete_function *delete_fn,int *keyval, void* extra_state*);
MPI_KEYVAL_CREATE	MPI_KEYVAL_CREATE(*EXTERNAL COPY_FN,EXTERNAL DELETE_FN,INTEGER KEYVAL,INTEGER EXTRA_STATE,INTEGER IERROR*)
MPI_Keyval_free	int MPI_Keyval_free(*int *keyval*);
MPI_KEYVAL_FREE	MPI_KEYVAL_FREE(*INTEGER KEYVAL,INTEGER IERROR*)
MPI_Attr_put	int MPI_Attr_put(*MPI_Comm comm,int keyval,void* attribute_val*);
MPI_ATTR_PUT	MPI_ATTR_PUT(*INTEGER COMM,INTEGER KEYVAL,INTEGER ATTRIBUTE_VAL,INTEGER IERROR*)
MPI_Attr_get	int MPI_Attr_get(*MPI_Comm comm,int keyval,void *attribute_val,int *flag*);
MPI_ATTR_GET	MPI_ATTR_GET(*INTEGER COMM,INTEGER KEYVAL,INTEGER ATTRIBUTE_VAL, LOGICAL FLAG,INTEGER IERROR*)
MPI_Attr_delete	int MPI_Attr_delete(*MPI_Comm comm,int keyval*);
MPI_ATTR_DELETE	MPI_ATTR_DELETE(*INTEGER COMM,INTEGER KEYVAL,INTEGER IERROR*)

Bindings for Topologies

Table 10 lists the C and FORTRAN bindings for topology routines.

Table 10 (Page 1 of 3). Bindings for Topologies

C/FORTRAN Subroutine	**C/FORTRAN Binding**
MPI_Cart_create	int MPI_Cart_create(*MPI_Comm comm_old,int ndims,int *dims,int *periods,int reorder,MPI_Comm *comm_cart*);
MPI_CART_CREATE	MPI_CART_CREATE(*INTEGER COMM_OLD,INTEGER NDIMS,INTEGER DIMS(*), INTEGER PERIODS(*),INTEGER REORDER,INTEGER COMM_CART,INTEGER IERROR*)
MPI_Dims_create	int MPI_Dims_create(*int nnodes,int ndims,int *dims*);
MPI_DIMS_CREATE	MPI_DIMS_CREATE(*INTEGER NNODES,INTEGER NDIMS,INTEGER DIMS(*), INTEGER IERROR*)

Table 10 (Page 2 of 3). Bindings for Topologies

C/FORTRAN Subroutine	**C/FORTRAN Binding**
MPI_Graph_create	int MPI_Graph_create(*MPI_Comm comm_old,int nnodes,int *index,int *edges,int reorder,MPI_Comm *comm_graph*);
MPI_GRAPH_CREATE	MPI_GRAPH_CREATE(*INTEGER COMM_OLD,INTEGER NNODES,INTEGER INDEX(*), INTEGER EDGES(*),INTEGER REORDER,INTEGER COMM_GRAPH,INTEGER IERROR*)
MPI_Topo_test	int MPI_Topo_test(*MPI_Comm comm,int *status*);
MPI_TOPO_TEST	MPI_TOPO_TEST(*INTEGER COMM,INTEGER STATUS,INTEGER IERROR*)
MPI_Graphdims_get	int MPI_Graphdims_get(*MPI_Comm comm,int *nnodes,int *nedges*);
MPI_GRAPHDIMS_GET	MPI_GRAPHDIMS_GET(*INTEGER COMM,INTEGER NNDODES,INTEGER NEDGES, INTEGER IERROR*)
MPI_Graph_get	int MPI_Graph_get(*MPI_Comm comm,int maxindex,int maxedges,int *index, int *edges*);
MPI_GRAPH_GET	MPI_GRAPH_GET(*INTEGER COMM,INTEGER MAXINDEX,INTEGER MAXEDGES,INTEGER INDEX(*),INTEGER EDGES(*),INTEGER IERROR*)
MPI_Cartdim_get	int MPI_Cartdim_get(*MPI_Comm comm, int *ndims*);
MPI_CARTDIM_GET	MPI_CARTDIM_GET(*INTEGER COMM,INTEGER NDIMS,INTEGER IERROR*)
MPI_Cart_get	int MPI_Cart_get(*MPI_Comm comm,int maxdims,int *dims,int *periods,int *coords*);
MPI_CART_GET	MPI_CART_GET(*INTEGER COMM,INTEGER MAXDIMS,INTEGER DIMS(*),INTEGER PERIODS(*),INTEGER COORDS(*),INTEGER IERROR*)
MPI_Cart_rank	int MPI_Cart_rank(*MPI_Comm comm,int *coords,int *rank*);
MPI_CART_RANK	MPI_CART_RANK(*INTEGER COMM,INTEGER COORDS(*),INTEGER RANK,INTEGER IERROR*)
MPI_Cart_coords	int MPI_Cart_coords(*MPI_Comm comm,int rank,int maxdims,int *coords*);
MPI_CART_COORDS	MPI_CART_COORDS(*INTEGER COMM,INTEGER RANK,INTEGER MAXDIMS,INTEGER COORDS(*),INTEGER IERROR*)
MPI_Graph_neighbors_count	int MPI_Graph_neighbors_count(*MPI_Comm comm,int rank,int *nneighbors*);
MPI_GRAPH_NEIGHBORS_COUNT	MPI_GRAPH_NEIGHBORS_COUNT(*INTEGER COMM,INTEGER RANK,INTEGER NEIGHBORS, INTEGER IERROR*)
MPI_Graph_neighbors	int MPI_Graph_neighbors(*MPI_Comm comm,int rank,int maxneighbors,int *neighbors*);
MPI_GRAPH_NEIGHBORS	MPI_GRAPH_NEIGHBORS(*MPI_COMM COMM,INTEGER RANK,INTEGER MAXNEIGHBORS,INTEGER NNEIGHBORS(*),INTEGER IERROR*)
MPI_Cart_shift	int MPI_Cart_shift(*MPI_Comm comm,int direction,int disp,int *rank_source,int *rank_dest*);
MPI_CART_SHIFT	MPI_CART_SHIFT(*INTEGER COMM,INTEGER DIRECTION,INTEGER DISP, INTEGER RANK_SOURCE,INTEGER RANK_DEST,INTEGER IERROR*)

Table 10 (Page 3 of 3). Bindings for Topologies

C/FORTRAN Subroutine	**C/FORTRAN Binding**
MPI_Cart_sub	int MPI_Cart_sub(*MPI_Comm comm,int *remain_dims,MPI_Comm *newcomm*);
MPI_CART_SUB	MPI_CART_SUB(*INTEGER COMM,INTEGER REMAIN_DIMS,INTEGER NEWCOMM, INTEGER IERROR*)
MPI_Cart_map	int MPI_Cart_map(*MPI_Comm comm,int ndims,int *dims,int *periods,int *newrank*);
MPI_CART_MAP	MPI_CART_MAP(*INTEGER COMM,INTEGER NDIMS,INTEGER DIMS(*),INTEGER PERIODS(*),INTEGER NEWRANK,INTEGER IERROR*)
MPI_Graph_map	int MPI_Graph_map(*MPI_Comm comm,int nnodes,int *index,int *edges,int *newrank*);
MPI_GRAPH_MAP	MPI_GRAPH_MAP(*INTEGER COMM,INTEGER NNODES,INTEGER INDEX(*),INTEGER EDGES(*),INTEGER NEWRANK,INTEGER IERROR*)

Bindings for Environment Management

Table 11 lists the C and FORTRAN bindings for environment management routines.

Table 11 (Page 1 of 2). Bindings for Environment Management

C/FORTRAN Subroutine	**C/FORTRAN Binding**
MPI_File_create_errhandler	int MPI_File_create_errhandler (*MPI_File_errhandler_fn *function, MPI_Errhandler *errhandler*);
MPI_FILE_CREATE_ERRHANDLER	MPI_FILE_CREATE_ERRHANDLER(*EXTERNAL FUNCTION,INTEGER ERRHANDLER, INTEGER IERROR*)
MPI_File_get_errhandler	int MPI_File_get_errhandler (*MPI_File file,MPI_Errhandler *errhandler*);
MPI_FILE_GET_ERRHANDLER	MPI_FILE_GET_ERRHANDLER (*INTEGER FILE,INTEGER ERRHANDLER, INTEGER IERROR*)
MPI_File_set_errhandler	int MPI_File_set_errhandler (*MPI_File fh, MPI_Errhandler errhandler*);
MPI_FILE_SET_ERRHANDLER	MPI_FILE_SET_ERRHANDLER(*INTEGER FH,INTEGER ERRHANLDER, INTEGER IERROR*)
MPI_Get_version	int MPI_Get_version(*int *version,int *subversion*);
MPI_GET_VERSION	MPI_GET_VERSION(*INTEGER VERSION,INTEGER SUBVERSION,INTEGER IERROR*)
MPI_Get_processor_name	int MPI_Get_processor_name(*char *name,int *resultlen*);
MPI_GET_PROCESSOR_NAME	MPI_GET_PROCESSOR_NAME(*CHARACTER NAME(*),INTEGER RESULTLEN,INTEGER IERROR*)
MPI_Errhandler_create	int MPI_Errhandler_create(*MPI_Handler_function *function, MPI_Errhandler *errhandler*);
MPI_ERRHANDLER_CREATE	MPI_ERRHANDLER_CREATE(*EXTERNAL FUNCTION,INTEGER ERRHANDLER, INTEGER IERROR*)
MPI_Errhandler_set	int MPI_Errhandler_set(*MPI_Comm comm,MPI_Errhandler errhandler*);
MPI_ERRHANDLER_SET	MPI_ERRHANDLER_SET(*INTEGER COMM,INTEGER ERRHANDLER,INTEGER IERROR*)

Table 11 (Page 2 of 2). Bindings for Environment Management

C/FORTRAN Subroutine	**C/FORTRAN Binding**
MPI_Errhandler_get	int MPI_Errhandler_get(*MPI_Comm comm,MPI_Errhandler *errhandler*);
MPI_ERRHANDLER_GET	MPI_ERRHANDLER_GET(*INTEGER COMM,INTEGER ERRHANDLER,INTEGER IERROR*)
MPI_Errhandler_free	int MPI_Errhandler_free(*MPI_Errhandler *errhandler*);
MPI_ERRHANDLER_FREE	MPI_ERRHANDLER_FREE(*INTEGER ERRHANDLER,INTEGER IERROR*)
MPI_Error_string	int MPI_Error_string(*int errorcode, char *string, int *resultlen*);
MPI_ERROR_STRING	MPI_ERROR_STRING(*INTEGER ERRORCODE,CHARACTER STRING(*),INTEGER RESULTLEN,INTEGER IERROR*)
MPI_Error_class	int MPI_Error_class(*int errorcode, int *errorclass*);
MPI_ERROR_CLASS	MPI_ERROR_CLASS(*INTEGER ERRORCODE,INTEGER ERRORCLASS,INTEGER IERROR*)
MPI_Wtime	double MPI_Wtime(*void*);
MPI_WTIME	DOUBLE PRECISION MPI_WTIME()
MPI_Wtick	double MPI_Wtick(*void*);
MPI_WTICK	DOUBLE PRECISION MPI_WTICK()
MPI_Init	int MPI_Init(*int *argc, char ***argv*);
MPI_INIT	MPI_INIT(*INTEGER IERROR*)
MPI_Finalize	int MPI_Finalize(*void*);
MPI_FINALIZE	MPI_FINALIZE(*INTEGER IERROR*)
MPI_Initialized	int MPI_Initialized(*int *flag*);
MPI_INITIALIZED	MPI_INITIALIZED(*INTEGER FLAG,INTEGER IERROR*)
MPI_Abort	int MPI_Abort(*MPI_Comm comm, int errorcode*);
MPI_ABORT	MPI_ABORT(*INTEGER COMM,INTEGER ERRORCODE,INTEGER IERROR*)

Bindings for Profiling

Table 12 lists the C and FORTRAN bindings for profiling.

Table 12. Bindings for Profiling

C/FORTRAN Subroutine	**C/FORTRAN Binding**
MPI_Pcontrol	int MPI_Pcontrol(*const int level,...*);
MPI_PCONTROL	MPI_PCONTROL(*INTEGER LEVEL,...*)

Bindings for Files

Table 13 lists the C and FORTRAN bindings for files.

Table 13 (Page 1 of 2). Bindings for MPI I/O

C/FORTRAN Subroutine	C/FORTRAN Binding
MPI_File_close	int MPI_File_close (MPI_File *fh);
MPI_FILE_CLOSE	MPI_FILE_CLOSE(INTEGER FH,INTEGER IERROR)
MPI_File_delete	int MPI_File_delete (char *filename,MPI_Info info);
MPI_FILE_DELETE	MPI_FILE_DELETE(CHARACTER*(*) FILENAME,INTEGER INFO, INTEGER IERROR)
MPI_File_get_amode	int MPI_File_get_amode (MPI_File fh,int *amode);
MPI_FILE_GET_AMODE	MPI_FILE_GET_AMODE(INTEGER FH,INTEGER AMODE,INTEGER IERROR)
MPI_File_get_atomicity	int MPI_File_get_atomicity (MPI_File fh,int *flag);
MPI_FILE_GET_ATOMICITY	MPI_FILE_GET_ATOMICITY (INTEGER FH,LOGICAL FLAG,INTEGER IERROR)
MPI_File_get_group	int MPI_File_get_group (MPI_File fh,MPI_Group *group);
MPI_FILE GET_GROUP	MPI_FILE GET_GROUP (INTEGER FH,INTEGER GROUP,INTEGER IERROR)
MPI_File_get_info	int MPI_File_get_info (MPI_File fh,MPI_Info *info_used);
MPI_FILE_GET_INFO	MPI_FILE_GET_INFO (INTEGER FH,INTEGER INFO_USED, INTEGER IERROR)
MPI_File_get_size	int MPI_File_get_size (MPI_File fh,MPI_Offset size);
MPI_FILE_GET_SIZE	MPI_FILE_GET_SIZE (INTEGER FH,INTEGER(KIND=MPI_OFFSET_KIND) SIZE, INTEGER IERROR)
MPI_File_get_view	int MPI_File_get_view (MPI_File fh,MPI_Offset *disp, MPI_Datatype *etype,MPI_Datatype *filetype,char *datarep);
MPI_FILE_GET_VIEW	MPI_FILE_GET_VIEW (INTEGER FH,INTEGER(KIND=MPI_OFFSET_KIND) DISP, INTEGER ETYPE,INTEGER FILETYPE,INTEGER DATAREP,INTEGER IERROR)
MPI_File_iread_at	int MPI_File_iread_at (MPI_File fh,MPI_Offset offset,void *buf, int count,MPI_Datatype datatype,MPI_Request *request);
MPI_FILE_IREAD_AT	MPI_FILE_IREAD_AT (INTEGER FH,INTEGER (KIND=MPI_OFFSET_KIND) OFFSET, CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER REQUEST, INTEGER IERROR)
MPI_File_iwrite_at	int MPI_File_iwrite_at (MPI_File fh,MPI_Offset offset,void *buf, int count,MPI_Datatype datatype,MPI_Request *request);
MPI_FILE_IWRITE_AT	MPI_FILE_IWRITE_AT(INTEGER FH,INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER REQUEST, INTEGER IERROR)
MPI_File_open	int MPI_File_open (MPI_Comm comm,char *filename,int amode,MPI_info, MPI_File *fh);
MPI_FILE_OPEN	MPI_FILE_OPEN(INTEGER COMM,CHARACTER FILENAME(*),INTEGER AMODE, INTEGER INFO,INTEGER FH,INTEGER IERROR)

Table 13 (Page 2 of 2). Bindings for MPI I/O

C/FORTRAN Subroutine	C/FORTRAN Binding
MPI_File_read_at	int MPI_File_read_at (*MPI_File fh,MPI_Offset offset,void *buf, int count,MPI_Datatype datatype,MPI_Status *status*);
MPI_FILE_READ_AT	MPI_FILE_READ_AT(*INTEGER FH,INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE, INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR*)
MPI_File_read_at_all	int MPI_File_read_at_all (*MPI_File fh,MPI_Offset offset,void *buf, int count,MPI_Datatype datatype,MPI_Status *status*);
MPI_FILE_READ_AT_ALL	MPI_FILE_READ_AT_ALL(*INTEGER FH,INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE, INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR*)
MPI_File_set_info	int MPI_File_set_info (*MPI_File fh,MPI_Info info*);
MPI_FILE_SET_INFO	MPI_FILE_SET_INFO(*INTEGER FH,INTEGER INFO,INTEGER IERROR*)
MPI_File_set_size	int MPI_File_set_size (*MPI_File fh,MPI_Offset size*);
MPI_FILE_SET_SIZE	MPI_FILE_SET_SIZE (*INTEGER FH,INTEGER(KIND=MPI_OFFSET_KIND) SIZE, INTEGER IERROR*)
MPI_File_set_view	int MPI_File_set_view (*MPI_File fh,MPI_Offset disp, MPI_Datatype etype,MPI_Datatype filetype, char *datarep,MPI_Info info*);
MPI_FILE_SET_VIEW	MPI_FILE_SET_VIEW (*INTEGER FH,INTEGER(KIND=MPI_OFFSET_KIND) DISP, INTEGER ETYPE,INTEGER FILETYPE,CHARACTER DATAREP(*),INTEGER INFO, INTEGER IERROR*)
MPI_File_sync	int MPI_File_sync (*MPI_File fh*);
MPI_FILE_SYNC	MPI_FILE_SYNC (*INTEGER FH,INTEGER IERROR*)
MPI_File_write_at	int MPI_File_write_at (*MPI_File fh,MPI_Offset offset,void *buf, int count,MPI_Datatype datatype,MPI_Status *status*);
MPI_FILE_WRITE_AT	MPI_FILE_WRITE_AT(*INTEGER FH,INTEGER(KIND_MPI_OFFSET_KIND) OFFSET, CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE, INTEGER STATUS(MPI_STATUS_SIZE), INTEGER IERROR*)
MPI_File_write_at_all	int MPI_File_write_at_all (*MPI_File fh,MPI_Offset offset,void *buf, int count,MPI_Datatype datatype,MPI_Status *status*);
MPI_FILE_WRITE_AT_ALL	MPI_FILE_WRITE_AT_ALL (*INTEGER FH, INTEGER (KIND=MPI_OFFSET_KIND) OFFSET, CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE, INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR*)

Bindings for info Objects

Table 14 lists the C and FORTRAN bindings for **info** objects.

Table 14 (Page 1 of 2). Bindings for info Objects

C/FORTRAN Subroutine	C/FORTRAN Binding
MPI_Info_create	int MPI_Info_create (*MPI_Info *info*);

Table 14 (Page 2 of 2). Bindings for info Objects

C/FORTRAN Subroutine	**C/FORTRAN Binding**
MPI_INFO_CREATE	MPI_INFO_CREATE (INTEGER INFO,INTEGER IERROR)
MPI_Info_delete	int MPI_Info_delete (MPI_Info info,char *key);
MPI_INFO_DELETE	MPI_INFO_DELETE (INTEGER INFO,CHARACTER KEY(*), INTEGER IERROR)
MPI_Info_dup	int MPI_Info_dup (MPI_Info info,MPI_Info *newinfo);
MPI_INFO_DUP	MPI_INFO_DUP (INTEGER INFO,INTEGER NEWINFO,INTEGER IERROR)
MPI_Info_free	int MPI_Info_free (MPI_Info *info);
MPI_INFO_FREE	MPI_INFO_FREE (INTEGER INFO,INTEGER IERROR)
MPI_Info_get	int MPI_Info_get (MPI_Info info,char *key,int valuelen, char *value,int *flag);
MPI_INFO_GET	MPI_INFO_GET (INTEGER INFO,CHARACTER KEY(*),INTEGER VALUELEN, CHARACTER VALUE(*),LOGICAL FLAG,INTEGER IERROR)
MPI_Info_get_nkeys	int MPI_Info_get_nkeys (MPI_Info info,int *nkeys);
MPI_INFO_GET_NKEYS	MPI_INFO_GET_NKEYS (INTEGER INFO,INTEGER NKEYS,INTEGER IERROR)
MPI_Info_get_nthkey	int MPI_Info_get_nthkey (MPI_Info info, int n, char *key);
MPI_INFO_GET_NTHKEY	MPI_INFO_GET_NTHKEY (INTEGER INFO,INTEGER N,CHARACTER KEY(*), INTEGER IERROR)
MPI_Info_get_valuelen	int MPI_Info_get_valuelen (MPI_Info info,char *key,int *valuelen, int *flag);
MPI_INFO_GET_VALUELEN	MPI_INFO_GET_VALUELEN (INTEGER INFO,CHARACTER KEY(*), INTEGER VALUELEN,LOGICAL FLAG,INTEGER IERROR)
MPI_Info_set	int MPI_Info_set(MPI_Info info,char *key,char *value);
MPI_INFO_SET	MPI_INFO_SET (INTEGER INFO,CHARACTER KEY(*),CHARACTER VALUE(*), INTEGER IERROR)

Appendix B. Profiling Message Passing

AIX Profiling

If you use **prof**, **gprof** or **xprofiler** and the appropriate compiler flag (-p or -pg), you can profile your program.

The message passing library is not enabled for **prof** or **gprof**, profiling counts. You can obtain profiling information by using the name-shifted MPI functions provided.

MPI Nameshift Profiling

To use nameshift profiling routines that are written to the C bindings with an MPI program written in C, or the FORTRAN bindings with an MPI program written in FORTRAN, do the following:

1. Create .o files for your profiling routines.
2. Use one of the following commands to list both the MPI program **.o** files and the profiling **.o** files as inputs:
   - **mpcc**
   - **mpxlf**
   - **mpcc_r**
   - **mpxlf_r**
   - **mpCC**
   - **mpCC_r**

   See *IBM Parallel Environment for AIX: Operation and Use, Volume 1* for more information on these commands.
3. Run the resulting executable normally.

To use nameshift profiling routines which are written to the C bindings with an MPI program written in FORTRAN, follow these steps:

- If you are both the creator and user of the profiling library, follow all the steps (1 through 17).
- If you are the creator of the profiling library, follow steps 1 through 6. You also need to provide the user with the file created in step 2.
- If you are the user of the profiling library, follow steps 7 through 17. For step 14, use the file generated by the creator of the profiling library in step 2.

Based on the above, follow the appropriate steps:

1. Create a source file containing profiling versions of all the MPI routines you want to profile. As an example, create a source file called myprof.c containing the following code:

```
#include <stdio.h>
#include "mpi.h"
int MPI_Init(int *argc, char ***argv) {
  int rc;

  printf("hello from profiling layer MPI_Init...\n");
  rc = PMPI_Init(argc, argv);
  printf("goodbye from profiling layer MPI_Init...\n");
  return(rc);
}
```

2. Create an export file containing all of the symbols your profiling library will export. Begin this file with the name your profiling library will have and the name of the **.o** that will have the object code of your profiling routines. As an example, create a file called myprof.exp containing the following statements:

   ```
   #!libmyprof.a(newmyprof.o)
   MPI_Init
   ```

3. Create a file called **mpicore.imp**. This file will import all of the PMPI symbols that your profiling library needs. Begin this file with the statement #!libmpi.a(mpicore.o). The following is an example of mpicore.imp:

   ```
   #!libmpi.a(mpicore.o)
   PMPI_Init
   ```

4. Compile the source file containing your profiling MPI routines. For example:

   ```
   cc -c myprof.c -I/usr/lpp/ppe.poe/include
   ```

   The -I defines the location of mpi.h.

5. Create your profiling MPI library. Use the file created in step 2 as the export file and the file created in step 3 as the import file. Include any other libraries your profiling code needs, such as libc. For example:

   ```
   ld -o newmyprof.o myprof.o -bM:SRE -H512 -T512 -bnoentry
   -bI:mpicore.imp -bE:myprof.exp -lc
   ```

6. Archive the object module created in step 5 into a library. The library name should be the same as that listed in the first statement of the export file created in step 2. For example:

   ```
   ar rv libmyprof.a newmyprof.o
   ```

7. Use the following command to extract mpifort.o from libmpi.a:

   ```
   ar -xv /usr/lpp/ppe.poe/lib/libmpi.a mpifort.o
   ```

8. Use the following command to create a non-shared version of mpifort.o:

   ```
   ld -o mpifort.tmp mpifort.o -r -bnso -bnoentry
   ```

9. Use the following command to extract mpicore.o from libmpi.a:

   ```
   ar -xv /usr/lpp/ppe.poe/lib/libmpi.a mpicore.o
   ```

10. Use the following command to create an export list from the extracted mpicore.o:

    ```
    /usr/bin/dump -nvp mpicore.o  | /usr/bin/grep "∧\[" | cut -f2-
    | cut -c26- | grep -y "^exp" | cut -c35- | sort | uniq > mpicore.exp
    ```

11. Delete all of the symbols selected for profiling in step 2 from mpicore.exp. Then create a new line at the top of the file. We'll call this the *new line 1* To the *new line 1*, add #!libmpi.a(mpicore.o). Continuing with our example: MPI_Init would

now be deleted from mpicore.exp. and #!libmpi.a(mpicore.o) would now comprise line 1 of mpicore.exp.

12. Create a file called **vt.exp** with the following statements:

    ```
    #!libvtd.a(dynamic.o)
    VT_instaddr_depth
    ```

13. Use the following command to create an export list from the extracted mpifort.o:

    ```
    /usr/bin/dump -nvp mpifort.o  | /usr/bin/grep "∧\[" | cut -f2-
    | cut -c26- | grep -y "^exp" | cut -c35- | sort | uniq > mpifort.exp
    ```

    insert **#!libpmpi.a**(newmpifort.o) as the first line of the new mpifort.exp file

14. Create a new version of mpifort.o from the non-shared version you created in step 8. It will import the symbols representing your profiling functions from your profiling library using the file created in step 2. It will import the remaining MPI symbols from mpicore.o using the file created in step 11. One additional symbol must be imported using the file created in step 12. The new mpifort.o will export symbols using the file created in step 13.

    ```
    ld -o newmpifort.o mpifort.tmp -bI:
    mpicore.exp -bI:myprof.exp -bI:vt.exp
    -bE:mpifort.exp -bM:SRE -H512 -T512 -bnoentry
    ```

15. Use the following command to create a library containing a FORTRAN object which will reference your profiling library:

    ```
    ar rv libpmpi.a newmpifort.o
    ```

16. Create a program that uses an MPI function you've profiled. An example would be a file called hwinit.f that contains the following statements:

    ```
     c ------------------------------------
             program  hwinit
             include 'mpif.h'
             integer forterr
     c
             call MPI_INIT(forterr)
     c
     c  Write comments to screen.
     c
             write(6,*)'Hello from task '
     c
             call MPI_FINALIZE(forterr)
     c
             stop
             end
     c
    ```

17. Compile your program linking in the library created in step 15. For example:

    ```
    mpxlf -o hwinit hwinit.f -lpmpi -L.
    ```

Sample CPU MPI Time Program

The following is a sample MPI program that uses the name-shifted MPI interface to separate the amount of user and system CPU time used by MPI.

CPU MPI Time Example

```
#include "mpi.h"
#include <sys/types.h>
#include <time.h>
#include <sys/times.h>

#define ARRAY_SIZE 1000000
#define VALUE 123

struct tms mpitms;
double mpi_elapsed;

void main()
{
   int in[ARRAY_SIZE],out[ARRAY_SIZE],tasks,me,src,dest;
   int i;
   MPI_Status status[2];
   MPI_Request msgid [2];

   for (i=0;i<ARRAY_SIZE;i++)out[i]=VALUE;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&tasks);
   MPI_Comm_rank(MPI_COMM_WORLD,&me);

   mpi_elapsed = MPI_Wtime();

   dest = (me==tasks-1) ? 0 : me+1;
   MPI_Isend(out,ARRAY_SIZE,MPI_INT,dest,5,MPI_COMM_WORLD,&msgid[0]);
   src = (me==0) ? tasks-1 : me-1;
   MPI_Irecv(in,ARRAY_SIZE,MPI_INT,src,5,MPI_COMM_WORLD,&msgid[1]);
   MPI_Waitall(2,msgid,status);

   for (i=0; i< ARRAY_SIZE; i++) {
   if(in[i] != VALUE )
      printf("ERROR on node %d, in = %d\n",me,in[i]);
          break;
          }

   MPI_Barrier(MPI_COMM_WORLD);

   mpi_elapsed = MPI_Wtime() - mpi_elapsed;

   printf("MPI CPU times: user %f, system %f, total %f sec\n",
             ((float)mpitms.tms_utime)/CLK_TCK,
             ((float)mpitms.tms_stime)/CLK_TCK,
             (float)(mpitms.tms_utime+mpitms.tms_stime)/CLK_TCK);
   printf("MPI Elapsed time: %f sec\n", mpi_elapsed);

   MPI_Finalize();
}

/****************************************/
/* Replacement functions for profiling */
/****************************************/

int MPI_Isend(void* buf, int count, MPI_Datatype datatype,
       int dest, int tag, MPI_Comm comm, MPI_Request *request)
{
        struct tms beforetms, aftertms;
        int rc;
        times(&beforetms);

    rc = PMPI_Isend(buf,count,datatype,dest,tag,comm,request);

        times(&aftertms);
        mpitms.tms_utime += (aftertms.tms_utime - beforetms.tms_utime);
        mpitms.tms_stime += (aftertms.tms_stime - beforetms.tms_stime);
        return (rc);
        }

int MPI_Waitall(int count, MPI_Request *array_of_requests,
              MPI_Status *array_of_statuses)
{
        struct tms beforetms, aftertms;
        int rc;
        times(&beforetms);

    rc = PMPI_Waitall(count,array_of_requests,array_of_statuses);

        times(&aftertms);
        mpitms.tms_utime += (aftertms.tms_utime - beforetms.tms_utime);
        mpitms.tms_stime += (aftertms.tms_stime - beforetms.tms_stime);
        return (rc);
        }

int MPI_Irecv(void* buf, int count, MPI_Datatype datatype,
       int source, int tag, MPI_Comm comm, MPI_Request *request)
{
        struct tms beforetms, aftertms;
        int rc;
        times(&beforetms);

    rc = PMPI_Irecv(buf,count,datatype,source,tag,comm,request);

        times(&aftertms);
        mpitms.tms_utime += (aftertms.tms_utime - beforetms.tms_utime);
        mpitms.tms_stime += (aftertms.tms_stime - beforetms.tms_stime);
        return (rc);
        }

int MPI_Barrier(MPI_Comm comm )
{
        struct tms beforetms, aftertms;
        int rc;
        times(&beforetms);

    rc = PMPI_Barrier(comm);

        times(&aftertms);
        mpitms.tms_utime += (aftertms.tms_utime - beforetms.tms_utime);
        mpitms.tms_stime += (aftertms.tms_stime - beforetms.tms_stime);
        return (rc);
        }
```

Appendix C. MPI Size Limits

MPI Tunables and Limits

The following is a list of MPI size limits. This list includes system limits on the size of various MPI elements and the relevant environment variable or tunable parameter.

- Number of tasks: MP_PROCS
- Maximum number of tasks: 1024 (2048 for IP library)
- Maximum message size for Point-to-Point communication: No specific limit
- Default receive buffer size: (MP_BUFFER_MEM)

  When using Internet Protocol (IP): 2,800,000 bytes
  When using User Space (US): 64MB
- Maximum receive buffer size: 64MB
- Default eager limit: See Table 15
- Maximum eager limit: 64K bytes
- To ensure that at least 32 messages can be outstanding between any two tasks, MP_EAGER_LIMIT is adjusted according to Table 15 (and: when MP_USE_FLOW_CONTROL=YES and MP_EAGER_LIMIT and MP_BUFFER_MEM have not been set by the user):

Table 15. MPI Eager Limits

Number of Tasks	MP_EAGER_LIMIT
1 to 16	4096
17 to 32	2048
33 to 64	1024
65 to 128	512
129 to 256	256
257 to the maximum number of tasks supported by the implementation	128

- The maximum number of outstanding unmatched send requests (smaller than MP_EAGER_LIMIT) per node for any single destination node is given by the formula:

  ```
  (0.75*MP_BUFFER_MEM)/(MP_PROCS*(max(MP_EAGER_LIMIT,64)))
  ```
- Maximum aggregate unsent data, per task: No specific limit
- Maximum number of communicators: approximately 2000
- Maximum number of data types: Depends on MP_BUFFER_MEM
- Maximum data type depth: Default is 5 (MP_MAX_TYPEDEPTH)
- Maximum number of distinct tags: All non-negative integers less than 2**32-1

Appendix D. Reduction Operations

Predefined Reduction Operations

The following is a list of the predefined operations for use with MPI_REDUCE, MPI_ALLREDUCE, MPI_REDUCE_SCATTER and MPI_SCAN. To invoke a predefined operation, place any of the following in **op**.

Reduction Operation	**Description**
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical AND
MPI_BAND	bitwise AND
MPI_LOR	logical OR
MPI_BOR	bitwise OR
MPI_LXOR	logical XOR
MPI_BXOR	bitwise XOR
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

Reduction Operations - Valid Datatype Arguments Operations

The reduction operations have the following basic **datatype** arguments.

Type	**Valid Datatype Arguments**
C integer	MPI_INT MPI_LONG MPI_LONG_LONG_INT MPI_SHORT MPI_UNSIGNED MPI_UNSIGNED_LONG MPI_UNSIGNED_LONG_LONG MPI_UNSIGNED_SHORT
FORTRAN integer	MPI_INTEGER MPI_INTEGER8
Floating point	MPI_DOUBLE MPI_DOUBLE_PRECISION MPI_FLOAT MPI_LONG_DOUBLE MPI_REAL
Logical	MPI_LOGICAL
Complex	MPI_COMPLEX

Type	Valid Datatype Arguments
Byte	MPI_BYTE
C Pair	MPI_DOUBLE_INT MPI_FLOAT_INT MPI_LONG_INT MPI_LONG_DOUBLE_INT MPI_SHORT_INT MPI_2INT
FORTRAN Pair	MPI_2DOUBLE_PRECISION MPI_2INTEGER MPI_2REAL

op Option - Valid Datatypes

The following are the valid datatypes for each **op** option.

Type	Valid Datatypes For op Option
C integer	MPI_BAND MPI_BOR MPI_BXOR MPI_LAND MPI_LOR MPI_LXOR MPI_MAX MPI_MIN MPI_SUM MPI_PROD
FORTRAN integer	MPI_BAND MPI_BOR MPI_BXOR MPI_MAX MPI_MIN MPI_PROD MPI_SUM
Floating point	MPI_MAX MPI_MIN MPI_PROD MPI_SUM
Logical	MPI_LAND MPI_LOR MPI_LXOR
Complex	MPI_PROD MPI_SUM
Byte	MPI_BAND MPI_BOR MPI_BXOR
C Pair	MPI_MAXLOC MPI_MINLOC
FORTRAN Pair	MPI_MAXLOC MPI_MINLOC

Examples

Examples of user-defined reduction functions for integer vector addition.

C Example

```
void int_sum (int *in, int *inout,
     int *len, MPI_Datatype *type);

{
   int i
   for (i=0; i<*len; i++)  {
      inout[i] + = in[i];
   }
}
```

FORTRAN Example

```
      SUBROUTINE INT_SUM(IN,INOUT,LEN,TYPE)

      INTEGER IN(*),INOUT(*),LEN,TYPE,I

      DO I = 1,LEN
         INOUT(I) = IN(I) + INOUT(I)
      ENDDO
      END
```

User-supplied reduction operations have four arguments:

- The first argument, **in**, is an array or scalar variable. The length, in elements, is specified by the third argument, **len**.

  This argument is an input array to be reduced.

- The second argument, **inout**, is an array or scalar variable. The length, in elements, is specified by the third argument, **len**.

  This argument is an input array to be reduced and the result of the reduction will be placed here.

- The third argument, **len** is the number of elements in **in** and **inout** to be reduced.

- The fourth argument **type** is the datatype of the elements to be reduced.

Users may code their own reduction operations, with the restriction that the operations must be associative. Also, C programmers should note that the values of **len** and **type** will be passed as pointers. No communication calls are allowed in user-defined reduction operations. See “Limitations In Setting The Thread Stacksize” on page 426 in Appendix G, “Programming Considerations for User Applications in POE” on page 411 for thread stacksize considerations when using the threaded MPI library.

Appendix E. Parallel Utility Functions

This chapter contains the man pages for the Parallel Utility functions. These user-callable, thread-safe functions exploit features of the IBM Parallel Environment for AIX. Included are functions for:

- updating the Program Marker Array lights
- controlling distribution of STDIN and STDOUT
- synchronizing parallel tasks without using the message passing library
- improving control of interrupt driven programs.

There is a C version and a Fortran version for most of the functions.

The Parallel Utility functions are:

MP_CHKPT, mp_chkpt
starts user-initiated checkpointing.

MP_DISABLEINTR, mpc_disableintr
disables packet arrival interrupts on the task on which it is executed.

MP_ENABLEINTR, mpc_enableintr
enables interrupts on the task on which it is executed.

MP_FLUSH, mpc_flush
flushes output buffers to STDOUT. This is a synchronizing call across all parallel tasks.

MP_MARKER, mpc_marker
requests that the numeric and text data passed in the call be forwarded to the Program Marker Array for display.

MP_NLIGHTS, mpc_nlights
returns the number of Program Marker Array lights defined for this session.

MP_QUERYINTR, mpc_queryintr
returns the state of interrupts on a task.

MP_QUERYINTRDELAY, mpc_queryintrdelay
returns, in microseconds, the current interrupt delay time.

MP_SETINTRDELAY, mpc_setintrdelay
sets the delay parameter to the specified value in **val**. This call can be made multiple times in a program with different values being passed to it each time.

MP_STDOUT_MODE, mpc_stdout_mode
requests that STDOUT be set to single, ordered, or unordered mode. In single mode, only one task output is displayed. In unordered mode, output is displayed in the order received at the home node. In ordered mode, each parallel task writes output data to its own buffer; when a flush request is made all the task buffers are flushed, in order of the task id, to home node's STDOUT.

MP_STDOUTMODE_QUERY, mpc_stdoutmode_query
returns the mode to which STDOUT is currently set.

mpc_isatty determines if a device is a terminal on the home node.

For more information on the Program Marker Array, or on controlling STDIN and STDOUT using POE, refer to *IBM Parallel Environment for AIX: Operation and Use, Volume 1*.

MP_CHKPT, mp_chkpt

Purpose

Starts user-initiated checkpointing.

Version

libmpi.a

C Synopsis

```
#include <pm_util.h>
int mp_chkpt(int flags);
```

Fortran Synopsis

```
i = MP_CHKPT(%val(j))
```

Parameters

In C, **flags** can be set to MP_CUSER, which indicates complete user-initiated checkpointing.

In Fortran, **j** should be set to 0 (zero), which is the value of MP_CUSER.

Description

MP_CHPKT initiates complete user-initiated checkpointing. When this function is reached, the program's execution is suspended. At that point, the state of the application is captured, along with all data, and saved to a file pointed to by the MP_CHECKFILE and MP_CHECKDIR environment variables.

Only POE/MPI applications submitted under LoadLeveler in batch mode are able to call this function. LoadLeveler is required for programs to call this function. Checkpointing of interactive POE applications is not allowed.

Notes

In complete user-initiated checkpointing, all instances of the parallel program must call MP_CHKPT. After all instances of the application have issued the MP_CHKPT call and have been suspended, a local checkpoint is taken on each node, with or without saving the message state, depending on the stage of the implementation.

Upon returning from the MP_CHKPT call, the application continues to run. It may, however, be a restarted application that is now running, rather than the original.

There are certain limitations associated with checkpointing an application. See "Checkpoint/Restart Limitations" on page 424 for details.

For general information on checkpointing and restarting programs, refer to *IBM Parallel Environment for AIX: Operation and Use, Volume 1*.

For more information on the use of LoadLeveler and checkpointing, refer to *Using and Administering LoadLeveler*.

Return Values

0	indicates successful completion
-1	indicates that an error occurred. A message describing the error will be issued.
1	indicates that a restart operation occurred.

MP_DISABLEINTR, mpc_disableintr

Purpose

Disables message arrival interrupts on a node.

Version

libmpi.a

C Synopsis

```
#include <pm_util.h>
int mpc_disableintr();
```

Fortran Synopsis

```
MP_DISABLEINTR(INTEGER RC)
```

Parameters

In Fortran, **rc** contains the values as described below in Return Values.

Description

This Parallel Utility function disables message arrival interrupts on the individual node on which it is run. Use this function to dynamically control masking interrupts on a node.

Notes

- This function overrides the setting of the environment variable MP_CSS_INTERRUPT.
- Inappropriate use of the interrupt control functions may reduce performance.
- This function can be used for IP and US protocols.
- This function is thread safe.
- | Using this function will suppress the MPI-directed switching of interrupt mode, leaving the user in control for the rest of the run. See MPI_FILE_OPEN.

Return Values

0	indicates successful completion
-1	indicates that an error occurred. A message describing the error will be issued.

Examples

C Example

```
/*
 * Running this program, after compiling with mpcc,
 * without setting the MP_CSS_INTERRUPT environment variable,
 * and without using the "-css_interrupt" command-line option,
 * produces the following output:
 *
 *    Interrupts are DISABLED
 *    About to enable interrupts..
 *    Interrupts are ENABLED
 *    About to disable interrupts...
 *    Interrupts are DISABLED
 */

#include "pm_util.h"

#define QUERY if (intr = mpc_queryintr()) {\
   printf("Interrupts are ENABLED\n");\
  } else {\
   printf("Interrupts are DISABLED\n");\
  }

main()
{
 int intr;

 QUERY

 printf("About to enable interrupts...\n");
 mpc_enableintr();

 QUERY

 printf("About to disable interrupts...\n");
 mpc_disableintr();

 QUERY
}
```

Fortran Example

Running the following program, after compiling with mpxlf, without setting the MP_CSS_INTERRUPT environment variable, and without using the "-css_interrupt" command-line option, produces the following output:

```
      Interrupts are DISABLED
      About to enable interrupts..
      Interrupts are ENABLED
      About to disable interrupts...
      Interrupts are DISABLED
```

```
      PROGRAM INTR_EXAMPLE

      INTEGER RC
      CALL MP_QUERYINTR(RC)
      IF (RC .EQ. 0) THEN
         WRITE(6,*)'Interrupts are DISABLED'
      ELSE
         WRITE(6,*)'Interrupts are ENABLED'
      ENDIF

      WRITE(6,*)'About to enable interrupts...'
      CALL MP_ENABLEINTR(RC)

      CALL MP_QUERYINTR(RC)
      IF (RC .EQ. 0) THEN
         WRITE(6,*)'Interrupts are DISABLED'
      ELSE
         WRITE(6,*)'Interrupts are ENABLED'
      ENDIF

      WRITE(6,*)'About to disable interrupts...'
      CALL MP_DISABLEINTR(RC)

      CALL MP_QUERYINTR(RC)
      IF (RC .EQ. 0) THEN
         WRITE(6,*)'Interrupts are DISABLED'
      ELSE
         WRITE(6,*)'Interrupts are ENABLED'
      ENDIF

      STOP
      END
```

Related Information

Functions:

- MP_ENABLEINTR, mpc_enableintr
- MP_QUERYINTR, mpc_queryintr
- MP_QUERYINTRDELAY, mpc_queryintrdelay
- MP_SETINTRDELAY, mpc_setintrdelay

MP_ENABLEINTR, mpc_enableintr

Purpose

Enables message arrival interrupts on a node.

Version

libmpi.a

C Synopsis

```
#include <pm_util.h>
int mpc_enableintr();
```

Fortran Synopsis

```
MP_ENABLEINTR(INTEGER RC)
```

Parameters

In Fortran, **rc** contains the values as described below in **Return Values.**

Description

This Parallel Utility function enables message arrival interrupts on the individual node on which it is run. Use this function to dynamically control masking interrupts on a node.

Notes

- This function overrides the setting of the environment variable MP_CSS_INTERRUPT.
- Inappropriate use of the interrupt control functions may reduce performance.
- This function can be used for IP and US protocols.
- This function is thread safe.
- | Using this function will suppress the MPI-directed switching of interrupt mode, leaving the user in control for the rest of the run. See MPI_FILE_OPEN.

Return Values

0	indicates successful completion
-1	indicates that an error occurred. A message describing the error will be issued.

Examples

C Example

```
/*
 * Running this program, after compiling with mpcc,
 * without setting the MP_CSS_INTERRUPT environment variable,
 * and without using the "-css_interrupt" command-line option,
 * produces the following output:
 *
 *    Interrupts are DISABLED
 *    About to enable interrupts..
 *    Interrupts are ENABLED
 *    About to disable interrupts...
 *    Interrupts are DISABLED
 */

#include "pm_util.h"

#define QUERY if (intr = mpc_queryintr()) {\
   printf("Interrupts are ENABLED\n");\
  } else {\
   printf("Interrupts are DISABLED\n");\
  }

main()
{
 int intr;

 QUERY

 printf("About to enable interrupts...\n");
 mpc_enableintr();

 QUERY

 printf("About to disable interrupts...\n");
 mpc_disableintr();

 QUERY
}
```

Fortran Example

Running this program, after compiling with mpxlf, without setting the MP_CSS_INTERRUPT environment variable, and without using the "-css_interrupt" command-line option, produces the following output:

```
      Interrupts are DISABLED
      About to enable interrupts..
      Interrupts are ENABLED
      About to disable interrupts...
      Interrupts are DISABLED



      PROGRAM INTR_EXAMPLE

      INTEGER RC
```

```
CALL MP_QUERYINTR(RC)
IF (RC .EQ. 0) THEN
   WRITE(6,*)'Interrupts are DISABLED'
ELSE
   WRITE(6,*)'Interrupts are ENABLED'
ENDIF

WRITE(6,*)'About to enable interrupts...'
CALL MP_ENABLEINTR(RC)

CALL MP_QUERYINTR(RC)
IF (RC .EQ. 0) THEN
   WRITE(6,*)'Interrupts are DISABLED'
ELSE
   WRITE(6,*)'Interrupts are ENABLED'
ENDIF

WRITE(6,*)'About to disable interrupts...'
CALL MP_DISABLEINTR(RC)

CALL MP_QUERYINTR(RC)
IF (RC .EQ. 0) THEN
   WRITE(6,*)'Interrupts are DISABLED'
ELSE
   WRITE(6,*)'Interrupts are ENABLED'
ENDIF

STOP
END
```

Related Information

Functions:

- MP_DISABLEINTR, mpc_disableintr
- MP_QUERYINTR, mpc_queryintr
- MP_QUERYINTRDELAY, mpc_queryintrdelay
- MP_SETINTRDELAY, mpc_setintrdelay

MP_FLUSH, mpc_flush

Purpose

Flushes task output buffers.

Version

libmpi.a

C Synopsis

```
#include <pm_util.h>
int mpc_flush(int option);
```

Fortran Synopsis

```
MP_FLUSH(INTEGER OPTION)
```

Parameters

option is an AIX file descriptor. The only valid value is:

1	to flush STDOUT buffers.

Description

This Parallel Utility function flushes output buffers from all of the parallel tasks to STDOUT at the home node. This is a synchronizing call across all parallel tasks.

If the current STDOUT mode is ordered, then when all tasks have issued this call or when any of the output buffers are full:

1. all STDOUT buffers are flushed and put out to the user screen (or redirected) in task order.
2. an acknowledgement is sent to all tasks and control is returned to the user.

If current STDOUT mode is unordered and all tasks have issued this call, all output buffers are flushed and put out to the user screen (or redirected).

If the current STDOUT mode is single and all tasks have issued this call, the output buffer for the current single task is flushed and put out to the user screen (or redirected).

Notes

- This is a synchronizing call regardless of the current STDOUT mode.
- All STDOUT buffers are flushed at the end of the parallel job.
- If mpc_flush is not used, standard output streams not terminated with a new-line character are buffered, even if a subsequent read to standard input is made. This may cause prompt message to appear only after input has been read.
- This function is thread safe.

Return Values

In C and C++ calls, the following applies:

0	indicates successful completion
-1	indicates that an error occurred. A message describing the error will be issued.

Examples

C Example

The following program uses **poe** with the **-labelio yes** option and three tasks:

```
    #include <pm_util.h>

main()
{
 mpc_stdout_mode(STDIO_ORDERED);
 printf("These lines will appear in task order\n");
 /*
  * Call mpc_flush here to make sure that one task
  * doesn't change the mode before all tasks have
  * sent the previous printf string to the home node.
  */
 mpc_flush(1);
 mpc_stdout_mode(STDIO_UNORDERED);
 printf("These lines will appear in the order received by the home node\n");
 /*
  * Since synchronization is not used here, one task could actually
  * execute the next statement before one of the other tasks has
  * executed the previous statement, causing one of the unordered
  * lines not to print.
  */
 mpc_stdout_mode(1);
 printf("Only 1 copy of this line will appear from task 1\n");
}
```

Running the above C program produces the following output (task order of lines 4-6 may differ):

```
   0 : These lines will appear in task order.

   1 : These lines will appear in task order.

   2 : These lines will appear in task order.

   1 : These lines will appear in the order received by the home node.

   2 : These lines will appear in the order received by the home node.
```

```
0 : These lines will appear in the order received by the home node.


1 : Only 1 copy of this line will appear from task 1.
```

Fortran Example

```
 CALL MP_STDOUT_MODE(-2)
 WRITE(6, *) 'These lines will appear in task order'
 CALL MP_FLUSH(1)
 CALL MP_STDOUT_MODE(-3)
 WRITE(6, *) 'These lines will appear in the order received by the
xhome node'
 CALL MP_STDOUT_MODE(1)
 WRITE(6, *) 'Only 1 copy of this line will appear from task 1'
 END
```

Related Information

Functions:

- MP_STDOUT_MODE, mpc_stdout_mode
- MP_STDOUTMODE_QUERY, mpc_stdoutmode_query

MP_MARKER, mpc_marker

Purpose

Passes numeric and text data to the Program Marker Array.

Version

libmpi.a

C Synopsis

```
#include <mp_marker.h>
void mpc_marker(int light, int color, char *string);
```

Fortran Synopsis

```
MP_MARKER(INTEGER LIGHT, INTEGER COLOR,
          CHARACTER STRING)
```

Parameters

light is the light number to be colored. The lights in each task row are numbered, left to right, from 0 to one less than the number of lights. The row on which the light is colored is that of the calling task.

If the value of light is out of range, the parameter is ignored. No light is colored by the subroutine. Setting the light to a negative number lets you update the string only.

color is the color you want to make the light. Supported values range from 0 to 102. The range 0 to 99 is, roughly, a spectrum starting with black and going through brown, green, blue, purple, red, orange, yellow, and ending with white. 100, 101, and 102 are three shades of gray growing increasingly dark. These are approximations, as the actual colors used are requested from the default X-Windows color map. If the X-Server is not capable of providing the colors in the RGB intensities requested, it colors the light with a close approximation.

If the value of color is out of range, the parameter is ignored. The subroutine does not give the light a new color.

string is the output string to be passed to the Program Marker Array. In C programs, the string must be null-terminated. This is not necessary in Fortran programs. The string can be any length, although only the first 80 characters can display. A null string can be passed.

Description

This Parallel Utility function requests that the numeric and text data passed in the call be forwarded to the Program Marker Array for display. This call waits for a specific acknowledgement from the Partition Manager if the number of lights (specified by the **MP_PMLIGHTS** environment variable) is positive. The program returns only after the message has been acknowledged by the POE home node.

Hence, this call will slow down the user's application and synchronize it approximately with the Program Marker Array.

If **MP_PMLIGHTS** is set to *0*, no message is sent.

Notes

- Creates a Visualization Tool (VT) trace record (marker event).
- Sends a message only if PMLIGHTS is greater than 0. Each call waits for an acknowledgement from the home node.
- The Program Marker Array X-Windows display routine is distributed as a sample program.
- This function is thread safe.

Examples

C Example

The C statement:

```
   #include <mp_marker.h>
 mpc_marker(2, 0, "Starting task 2");
```

gives the third light of the calling task the color black and after the lights on that task's row, prints the string "Starting task 2".

Fortran Example

The Fortran statement:

```
CALL MP_MARKER(2, 0, 'Starting task 2')
```

gives the third light of the calling task the color black and after the lights on that task's row, prints the string 'Starting task 2'.

Related Information

Commands: pmarray

Functions: MP_NLIGHTS, mpc_nlights

MP_NLIGHTS, mpc_nlights

Purpose

Gets the number of Program Marker Array lights defined for this session.

Version

libmpi.a

C Synopsis

```
#include <mp_marker.h>
int mpc_nlights();
```

Fortran Synopsis

```
MP_NLIGHTS(INTEGER NLIGHT)
```

Parameters

In Fortran, the number of Program Marker Array lights defined for this session is returned in the variable **NLIGHT**.

Description

This Parallel Utility function returns the number of Program Marker Array lights defined for this session.

Notes

- You can set the number of Program Marker Array lights by using the environment variable **MP_PMLIGHTS** or the command line option **-pmlights**.
- This function is thread safe.

Return Values

In C, the current number of the Program Marker Array lights is the return value.

Examples

C Example

The following program uses poe with the **-pmlights 3** option and 1 task:

```
   #include <mp_marker.h>

main()
{
 printf("Number of Program Marker Array lights for this session is %d\n",
 mpc_nlights());
}
```

Running the above C program produces the following output:

```
    Number of Program Marker Array lights for this session is 3.
```

Fortran Example

The following program uses poe with the **-pmlights 3** option and 1 task:

```
      INTEGER LIGHTS

      CALL MP_NLIGHTS(LIGHTS)
      WRITE(6, *) 'Number of Program Marker Array lights for this
     xsession is ', LIGHTS
      END
```

Running the above produces the following output:

```
    Number of Program Marker Array lights for this session is 3.
```

Related Information

Commands: pmarray

Functions: MP_MARKER, mpc_marker

MP_QUERYINTR, mpc_queryintr

Purpose

Returns the state of interrupts on a node.

Version

libmpi.a

C Synopsis

```
#include <pm_util.h>
int mpc_queryintr();
```

Fortran Synopsis

```
MP_QUERYINTR(INTEGER RC)
```

Parameters

In Fortran, **rc** contains the values as described below in RETURN VALUES.

Description

This Parallel Utility function returns the state of interrupts on a node.

Notes

- This function is thread safe.

Return Values

0	indicates that interrupts are disabled on the node from which this function is called.
1	indicates that interrupts are enabled on the node from which this function is called.

Examples

C Example

```
/*
 * Running this program, after compiling with mpcc,
 * without setting the MP_CSS_INTERRUPT environment variable,
 * and without using the "-css_interrupt" command-line option,
 * produces the following output:
 *
 *    Interrupts are DISABLED
 *    About to enable interrupts..
 *    Interrupts are ENABLED
 *    About to disable interrupts...
 *    Interrupts are DISABLED
 */

#include "pm_util.h"

#define QUERY if (intr = mpc_queryintr()) {\
   printf("Interrupts are ENABLED\n");\
  } else {\
   printf("Interrupts are DISABLED\n");\
  }

main()
{
 int intr;

 QUERY

 printf("About to enable interrupts...\n");
 mpc_enableintr();

 QUERY

 printf("About to disable interrupts...\n");
 mpc_disableintr();

 QUERY
}
```

Fortran Example

Running this program, after compiling with mpxlf, without setting the MP_CSS_INTERRUPT environment variable, and without using the "-css_interrupt" command-line option, produces the following output:

```
Interrupts are DISABLED
About to enable interrupts..
Interrupts are ENABLED
About to disable interrupts...
Interrupts are DISABLED
```

```
      PROGRAM INTR_EXAMPLE

      INTEGER RC

      CALL MP_QUERYINTR(RC)
      IF (RC .EQ. 0) THEN
         WRITE(6,*)'Interrupts are DISABLED'
      ELSE
         WRITE(6,*)'Interrupts are ENABLED'
      ENDIF

      WRITE(6,*)'About to enable interrupts...'
      CALL MP_ENABLEINTR(RC)

      CALL MP_QUERYINTR(RC)
      IF (RC .EQ. 0) THEN
         WRITE(6,*)'Interrupts are DISABLED'
      ELSE
         WRITE(6,*)'Interrupts are ENABLED'
      ENDIF
      WRITE(6,*)'About to disable interrupts...'
      CALL MP_DISABLEINTR(RC)

      CALL MP_QUERYINTR(RC)
      IF (RC .EQ. 0) THEN
         WRITE(6,*)'Interrupts are DISABLED'
      ELSE
         WRITE(6,*)'Interrupts are ENABLED'
      ENDIF

      STOP
      END
```

Related Information

Functions:

- MP_DISABLEINTR, mpc_disableintr
- MP_ENABLEINTR, mpc_enableintr
- MP_QUERYINTRDELAY, mpc_queryintrdelay
- MP_SETINTRDELAY, mpc_setintrdelay

MP_QUERYINTRDELAY, mpc_queryintrdelay

Purpose

Returns the current interrupt delay time.

Version

libmpi.a

C Synopsis

```
#include <pm_util.h>
int mpc_queryintrdelay();
```

Fortran Synopsis

```
MP_QUERYINTRDELAY(INTEGER RC)
```

Parameters

In Fortran, **rc** contains the values as described below in RETURN VALUES.

Description

This Parallel Utility function returns the current interrupt delay time in microseconds.

Notes

- The default interrupt delay time is 35 microseconds for TB2 and 1 microsecond for TB3.
- This function is thread safe.

Return Values

The current interrupt delay time in microseconds.

Examples

C Example

```
/*
 * Running this program, after compiling with mpcc,
 * without setting the MP_INTRDELAY environment variable,
 * and without using the "-intrdelay" command-line option,
 * produces the following output:
 *
 *    Current interrupt delay time is 35
 *    About to set interrupt delay time to 100...
 *    Current interrupt delay time is 100
 */

#include "pm_util.h"

main()
{
 printf("Current interrupt delay time is %d\n", mpc_queryintrdelay());
 printf("About to set interrupt delay time to 100...\n");
 mpc_setintrdelay(100);

 printf("Current interrupt delay time is %d\n", mpc_queryintrdelay());
}
```

Fortran Example

Running this program, after compiling with mpxlf, without setting the MP_INTRDELAY environment variable, and without using the "-intrdelay" command-line option, produces the following output:

```
Current interrupt delay time is 35
About to set interrupt delay time to 100...
Current interrupt delay time is 100
```

```
PROGRAM INTRDELAY_EXAMPLE

INTEGER DELAY, RC

CALL  MP_QUERYINTRDEALY(DELAY)
WRITE(6,*)'Current interrupt delay time is', delay

WRITE(6,*)'About to set interrupt delay time to 100...'
DELAY = 100
CALL MP_SETINTRDELAY(DELAY, RC)

CALL  MP_QUERYINTRDELAY(DELAY)
WRITE(6,*)'Current interrupt delay time is', delay

STOP
END
```

Related Information

Functions:

- MP_DISABLEINTR, mpc_disableintr
- MP_ENABLEINTR, mpc_enableintr
- MP_QUERYINTR, mpc_queryintr
- MP_SETINTRDELAY, mpc_setintrdelay

MP_SETINTRDELAY, mpc_setintrdelay

Purpose

Sets the delay parameter.

Version

libmpi.a

C Synopsis

```
#include <pm_util.h>
int mpc_setintrdelay(integer val);
```

Fortran Synopsis

```
MP_SETINTRDELAY(INTEGER VAL, INTEGER RC)
```

Parameters

val	is the delay parameter in microseconds
rc	in Fortran, **rc** contains the values as described below in RETURN VALUES.

Description

This Parallel Utility function sets the delay parameter to the value specified in **val**. This call can be made multiple times in a program with different values being passed to it each time.

You can use the environment variable MP_INTERDELAY to set an integer value before running your program. In this way, you can tune your delay parameter without having to recompile existing applications.

The cost of servicing an interrupt is quite high. For an application with few nodes exchanging small messages, it will help latency if the interrupt delay is kept small. For an application with a large number of nodes or one which exchanges large messages, keeping the delay parameter large will help the bandwidth. This allows multiple read transmissions to occur in a single read cycle. You should experiment with different values functions to achieve the desired performance depending on the communication pattern.

Notes

- The default interrupt delay time is 35 microseconds for TB2 and 1 microsecond for TB3.
- Overrides the setting of the environment variable MP_INTERDELAY.
- This function is thread safe.

Return Values

0	indicates successful completion
-1	indicates that an error occurred. A message describing the error will be issued.

Examples

C Example

```
/*
 * Running this program, after compiling with mpcc,
 * without setting the MP_INTRDELAY environment variable,
 * and without using the "-intrdelay" command-line option,
 * produces the following output:
 *
 *    Current interrupt delay time is 35
 *    About to set interrupt delay time to 100...
 *    Current interrupt delay time is 100
 */

#include "pm_util.h"

main()
{
 printf("Current interrupt delay time is %d\n", mpc_queryintrdelay());

 printf("About to set interrupt delay time to 100...\n");
 mpc_setintrdelay(100);

 printf("Current interrupt delay time is %d\n", mpc_queryintrdelay());
}
```

Fortran Example

Running this program, after compiling with mpxlf, without setting the MP_INTRDELAY environment variable, and without using the "-intrdelay" command-line option, produces the following output:

```
      Current interrupt delay time is 35
      About to set interrupt delay time to 100...
      Current interrupt delay time is 100
```

```
      PROGRAM INTRDELAY_EXAMPLE

      INTEGER DELAY, RC

      CALL  MP_QUERYINTRDELAY(DELAY)
      WRITE(6,*)'Current interrupt delay time is', delay

      WRITE(6,*)'About to set interrupt delay time to 100...'
      DELAY = 100
      CALL MP_SETINTRDELAY(DELAY, RC)
```

```
CALL  MP_QUERYINTRDELAY(DELAY)
WRITE(6,*)'Current interrupt delay time is', delay

STOP
END
```

Related Information

Functions:

- MP_DISABLEINTR, mpc_disableintr
- MP_ENABLEINTR, mpc_enableintr
- MP_QUERYINTR, mpc_queryintr
- MP_QUERYINTRDELAY, mpc_queryintrdelay

MP_STDOUT_MODE, mpc_stdout_mode

Purpose

Sets the mode for STDOUT.

Version

libmpi.a

C Synopsis

```
#include <pm_util.h>
int mpc_stdout_mode(int mode);
```

Fortran Synopsis

```
MP_STDOUT_MODE(INTEGER MODE)
```

Parameters

mode is the mode to which STDOUT is to be set. The valid values are:

taskid specifies single mode for STDOUT, where *taskid* is the task identifier of the new single task. This value must be between 0 and *n*-1, where *n* is the total of tasks in the current partition. The *taskid* requested does not have to be the issuing task.

-2 specifies ordered mode for STDOUT. The macro STDIO_ORDERED is supplied for use in C programs.

-3 specifies unordered mode for STDOUT. The macro STDIO_UNORDERED is supplied for use in C programs.

Description

This Parallel Utility function requests that STDOUT be set to single, ordered, or unordered mode. In single mode, only one task output is displayed. In unordered mode, output is displayed in the order received at the home node. In ordered mode, each parallel task writes output data to its own buffer. When a flush request is made all the task buffers are flushed, in order of task ID, to STDOUT home node.

Notes

- All current STDOUT buffers are flushed before the new STDOUT mode is established.
- The initial mode for STDOUT is set by using the environment variable **MP_STDOUTMODE**, or by using the poe command line option **-stdoutmode**, with the latter overriding the former. The default STDOUT mode is unordered.
- This function is implemented with a half second sleep interval to ensure that the mode change request is processed before subsequent writes to STDOUT.
- This function is thread safe.

Return Values

In C and C++ calls, the following applies:

0 indicates successful completion.

-1 indicates that an error occurred. A message describing the error will be issued.

Examples

C Example

The following program uses poe with the **-labelio yes** option and three tasks:

```
    #include <pm_util.h>

main()
{
 mpc_stdout_mode(STDIO_ORDERED);
 printf("These lines will appear in task order\n");
 /*
  * Call mpc_flush here to make sure that one task
  * doesn't change the mode before all tasks have
  * sent the previous printf string to the home node.
  */
 mpc_flush(1);
 mpc_stdout_mode(STDIO_UNORDERED);
 printf("These lines will appear in the order received by the home node\n");
 /*
  * Since synchronization is not used here, one task could actually
  * execute the next statement before one of the other tasks has
  * executed the previous statement, causing one of the unordered
  * lines not to print.
  */
 mpc_stdout_mode(1);
 printf("Only 1 copy of this line will appear from task 1\n");
}
```

Running the above C program produces the following output (task order of lines 4-6 may differ):

```
   0 : These lines will appear in task order.

   1 : These lines will appear in task order.

   2 : These lines will appear in task order.

   1 : These lines will appear in the order received by the home node.

   2 : These lines will appear in the order received by the home node.
```

```
0 : These lines will appear in the order received by the home node.


1 : Only 1 copy of this line will appear from task 1.
```

Fortran Example

```
  CALL MP_STDOUT_MODE(-2)
  WRITE(6, *) 'These lines will appear in task order'
  CALL MP_FLUSH(1)
  CALL MP_STDOUT_MODE(-3)
  WRITE(6, *) 'These lines will appear in the order received by the
 xhome node'
  CALL MP_STDOUT_MODE(1)
  WRITE(6, *) 'Only 1 copy of this line will appear from task 1'
  END
```

Running the above program produces the following output (task order of lines 4-6 may differ):

```
0 : These lines will appear in task order.


1 : These lines will appear in task order.


2 : These lines will appear in task order.


1 : These lines will appear in the order received by the home node.


2 : These lines will appear in the order received by the home node.


0 : These lines will appear in the order received by the home node.


1 : Only 1 copy of this line will appear from task 1.
```

Related Information

Functions:

- MP_FLUSH, mpc_flush
- MP_STDOUTMODE_QUERY, mpc_stdoutmode_query
- MP_SYNCH, mpc_synch
- mpcc
- mpCC
- mpxlf

MP_STDOUTMODE_QUERY, mpc_stdoutmode_query

Purpose

Queries the current STDOUT mode setting.

Version

libmpi.a

C Synopsis

```
#include <pm_util.h>
int mpc_stdoutmode_query(int *mode);
```

Fortran Synopsis

```
MP_STDOUTMODE_QUERY(INTEGER MODE)
```

Parameters

mode is the address of an integer in which the current STDOUT mode setting will be returned. Possible return values are:

taskid	indicates that the current STDOUT mode is single, i.e. output for only task taskid is displayed.
-2	indicates that the current STDOUT mode is ordered. The macro STDIO_ORDERED is supplied for use in C programs.
-3	indicates that the current STDOUT mode is unordered. The macro STDIO_UNORDERED is supplied for use in C programs.

Description

This Parallel Utility function returns the mode to which STDOUT is currently set.

Notes

- Between the time one task issues a mode query request and receives a response, it is possible that another task can change the STDOUT mode setting to another value unless proper synchronization is used.
- This function is thread safe.

Return Values

In C and C++ calls, the following applies:

0	indicates successful completion
-1	indicates that an error occurred. A message describing the error will be issued.

Examples

C Example

The following program uses poe with one task:

```
#include <pm_util.h>

main()
{
 int mode;

 mpc_stdoutmode_query(&mode);
 printf("Initial (default) STDOUT mode is %d\n", mode);
 mpc_stdout_mode(STDIO_ORDERED);
 mpc_stdoutmode_query(&mode);
 printf("New STDOUT mode is %d\n", mode);
}
```

Running the above program produces the following output:

```
Initial (default) STDOUT mode is -3


New STDOUT mode is -2
```

Fortran Example

The following program uses poe with one task:

```
      INTEGER MODE

      CALL MP_STDOUTMODE_QUERY(mode)
      WRITE(6, *) 'Initial (default) STDOUT mode is', mode
      CALL MP_STDOUT_MODE(-2)
      CALL MP_STDOUTMODE_QUERY(mode)
      WRITE(6, *) 'New STDOUT mode is', mode
      END
```

Running the above program produces the following output:

```
Initial (default) STDOUT mode is -3


New STDOUT mode is -2
```

Related Information

Functions:

- MP_FLUSH, mpc_flush
- MP_STDOUT_MODE, mpc_stdout_mode
- MP_SYNCH, mpc_synch

- mpcc
- mpCC
- mpxlf

mpc_isatty

Purpose

Determines whether a device is a terminal on the home node.

Version

libmpi.a

C Synopsis

```
#include <pm_util.h>
int mpc_isatty(int FileDescriptor);
```

Fortran Synopsis

A Fortran version of this function is not available.

Parameters

FileDescriptor is the file descriptor number of the device. Valid values are:

0 or STDIN	specifies STDIN as the device to be checked.
1 or STDOUT	specifies STDOUT as the device to be checked.
2 or STDERR	specifies STDERR as the device to be checked.

Description

This Parallel Utility function determines whether the file descriptor specified by the FileDescriptor parameter is associated with a terminal device on the home node. In a Parallel Operating Environment partition, these three file descriptors are implemented as pipes to the Partition Manager Daemon. Therefore, the AIX isatty() function will always return FALSE for each of them. This function is provided for use by remote tasks that may want to know whether one of these devices is actually a terminal on the home node, for example, to determine whether or not to output a prompt.

Notes

- This function is thread safe.

Return Values

In C and C++ calls, the following applies:

0	indicates that the device is *not* associated with a terminal on the home node.
1	indicates that the device *is* associated with a terminal on the home node.
-1	indicates an invalid FileDescriptor parameter.

Examples

C Example

```
/*
 * Running this program, after compiling with mpcc,
 * without redirecting STDIN, produces the following output:
 *
 *     isatty() reports STDIN as a non-terminal device
 *     mpc_isatty() reports STDIN as a terminal device
 */

#include "pm_util.h"

main()
{
 if (isatty(STDIN)) {
  printf("isatty() reports STDIN as a terminal device\n");
 } else {
  printf("isatty() reports STDIN as a non-terminal device\n");
  if (mpc_isatty(STDIN)) {
   printf("mpc_isatty() reports STDIN as a terminal device\n");
  } else {
   printf("mpc_isatty() reports STDIN as a non-terminal device\n");
  }
 }
}
```

Appendix F. Tracing Routines

This chapter contains the syntax man pages for the tracing routines used in the Visualization Tool Tracing Subsystem Tracing Subsystem. These user-callable routines allow the programmer to customize the trace collection from within the application being traced. Included are functions for:

- modifying trace collection parameters
- starting and stopping the trace from within the program, dynamically and selectively
- writing trace records back to the home node.

There is a C version and a Fortran version for each of the routines.

The tracing routines are:

VT_TRC_FLUSH and VT_trc_flush_c flushes the memory buffer to the trace collection directory.

VT_TRC_SET_PARAMS and VT_trc_set_params_c sets certain tracing parameters at which the dig should sample kernel statistics.

VT_TRC_START and VT_trc_start_c requests a trace from within the program, dynamically and selectively.

VT_TRC_STOP and VT_trc_stop_c requests that the collecting of trace events be discontinued.

For more information, see *IBM Parallel Environment for AIX: Operation and Use, Volume 2*.

VT_TRC_FLUSH, VT_trc_flush_c

Purpose

Flushes the memory buffer to the trace collection directory.

Version

- C Library (libvtd.a)
- Fortran Library (libvtd.a)

  The above are automatically included by the POE functions.

C Synopsis

```
#include <VT_trc.h>
int VT_trc_flush_c();
```

Fortran Synopsis

```
VT_TRC_FLUSH(INTEGER RETURN_CODE)
```

Parameters

In Fortran, the following applies:

return_code integer which receives the return value.

0	indicates successful completion
-1	indicates an error occurred.

Description

This routine may be called from the application to flush the memory buffer to the trace collection directory. This is specified by the MP_TRACE directory environment variable or -tracedir parameter and defaults to the directory from which the application was stored.

Return Values

In C and C++ calls, the following applies:

0	indicates successful completion
-1	indicates that an error occurred. A message describing the error will be issued.

Errors

- 0033-3090 write failed to the dig daemon, ..., error is ...

  The application could not contact the kernel statistics sampling daemon.

For more information about error conditions, refer to *IBM Parallel Environment for AIX: Messages* .

Examples

C Example

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>
#include <VT_trc.h>

#define COUNT 1024

int main(int argc, char **argv)
{
  int i;
  int numtask;
  int taskid;
  int msg_in[COUNT], msg_out[COUNT];
  int src, dest, len, send_msgid, recv_msgid, nBytes;

  /* Find out number of tasks/nodes. */
  MPI_Init( &argc, &argv);
  MPI_Comm_size( MPI_COMM_WORLD, &numtask);
  MPI_Comm_rank( MPI_COMM_WORLD, &taskid);

  /*****************************************************

    Set tracing parameters as follows:
      Memory buffer = 500K
      Temporary trace file = 1M
      Sample every second
      Do not use wrap around buffer

   *****************************************************/

  if(0!=VT_trc_set_params_c(500000,1000000,1000,0))
    {
      printf("Could not reset trace parameters!\n");
    }
  /* Disable tracing while the message buffer is being initialized */
  if(0!=VT_trc_stop_c())
    {
      printf("Could not stop tracing!\n");
    }
  /* Flush the memory buffer to disk */
  if(0!=VT_trc_flush_c())
    {
      printf("Could not flush trace buffer!\n");
    }
  for(i=0;i<COUNT;i++)
    msg_out[i]=taskid;

  /* Re-enable tracing.  Level 9 asks for everything */
  /* but only events enabled by the command line or  */
  /* environment variable will be re-enabled)        */
  if(0!=VT_trc_start_c(9))
    {
      printf("Could not restart tracing!\n");
    }
  dest = (taskid<(numtask-1))?(taskid+1):0;
  MPI_Send(msg_out, COUNT, MPI_INT, dest, 0, MPI_COMM_WORLD);

  src = (taskid>0)?(taskid-1):(numtask-1);
  MPI_Recv(msg_in, COUNT, MPI_INT, src, 0, MPI_COMM_WORLD);

  MPI_Finalize();
}
```

Fortran Example

```
        PROGRAM TRCDEMO
C
        INCLUDE "mpif.h"
        IMPLICIT   NONE
        INTEGER    COUNT, I
        INTEGER    BUFFSZ, FILESZ
        INTEGER    SMPL, WRAP
        PARAMETER  ( COUNT = 1024 )
        PARAMETER  ( BUFFSZ = 500000, FILESZ = 1000000 )
        PARAMETER  ( SMPL = 1000, WRAP = 0 )
        INTEGER    MSG_IN(COUNT), MSG_OUT(COUNT)
        INTEGER    NUMTASK, TASKID
        INTEGER    SRC, DEST
        INTEGER    RC
C
C FIND OUT NUMBER OF TASKS

        CALL MPI_INIT( RC )
        CALL MPI_COMM_SIZE( MPI_COMM_WORLD, NUMTASK, RC )
        CALL MPI_COMM_RANK( MPI_COMM_WORLD, TASKID, RC )

C
C
C    SET TRACING PARAMETERS AS FOLLOWS:
C      MEMORY BUFFER = 500K
C      TEMPORARY TRACE FILE = 1M
C      SAMPLE EVERY SECOND
C      DO NOT USE WRAP AROUND BUFFER
C
C

        CALL VT_TRC_SET_PARAMS(BUFFSZ, FILESZ, SMPL, WRAP, RC)
        IF(RC .NE. 0) THEN
          WRITE(6,*)'Could not reset trace parameters!'
        ENDIF

C DISABLE TRACING WHILE THE MESSAGE BUFFER IS BEING INITIALIZED
        CALL VT_TRC_STOP( RC )
        IF(RC .NE. 0) THEN
          WRITE(6,*)'Could not stop tracing!'
        ENDIF

C FLUSH THE MEMORY BUFFER TO DISK
        CALL VT_TRC_FLUSH( RC )
        IF(RC .NE. 0) THEN
          WRITE(6,*)'Could not flush trace buffer!'
```

```
      ENDIF

      DO I = 1,COUNT
        MSG_OUT(I) = TASKID
      ENDDO

C RE-ENABLE TRACING.  LEVEL 9 ASKS FOR EVERYTHING
C BUT ONLY EVENTS ENABLED BY THE COMMAND LINE OR
C ENVIRONMENT VARIABLE WILL BE RE-ENABLED)
      CALL VT_TRC_START( 9, RC )
      IF(RC .NE. 0) THEN
        WRITE(6,*)'Could not restart tracing!'
      ENDIF

      IF( TASKID .GE. NUMTASK-1 ) THEN
        DEST = 0
      ELSE
        DEST = TASKID + 1
      ENDIF

      CALL MPI_SEND(MSG_OUT, COUNT, MPI_INTEGER, DEST, 0,
     +          MPI_COMM_WORLD, RC )

      IF( TASKID .LE. 0 ) THEN
        SRC = NUMTASK - 1
      ELSE
        SRC = TASKID - 1
      ENDIF

      CALL MPI_RECV(MSG_IN, COUNT, MPI_INTEGER, SRC, 0,
     +         MPI_COMM_WORLD, RC)

      CALL MPI_FINALIZE( RC )

      STOP
      END
C
```

Related Information

Functions:

- VT_TRC_SET_PARAMS, VT_trc_set_params_c
- VT_TRC_START, VT_trc_start_c
- VT_TRC_STOP, VT_trc_stop_c

For more information about VT tracing, see *IBM Parallel Environment for AIX: Operation and Use, Volume 2*.

VT_TRC_SET_PARAMS, VT_trc_set_params_c

Purpose

Lets applications set tracing parameters.

Version

- C Library (libvtd.a) functions.
- Fortran Library (libvtd.a)

  The above are automatically included by the POE.

C Synopsis

```
#include <VT_trc.h>
int VT_trc_set_params_c(size_t buff_size, size_t file_size,
                        int sampling_freq, int wrap_around_flag);
```

Fortran Synopsis

```
VT_TRC_SET_PARAMS(INTEGER BUFF_SIZE, INTEGER FILE_SIZE,
                  INTEGER SAMPLING_FREQ, INTEGER WRAP_AROUND_FLAG,
                  INTEGER RETURN_CODE)
```

Parameters

buff_size	the size of the memory buffer
file_size	the size of the node trace files
sampling_freq	the interval with which to sample AIX units (microseconds)
wrap_around_flag	if non-zero, a wrap around memory buffer will be used.
return_code	integer which receives the return value.

0	indicates successful completion
-1	indicates an error occurred.

Description

This routine allows applications to set certain tracing parameters, such as:

- The size of the memory buffer and trace files
- The kernel statistics sampling interval

Notes

- Parameter values replace command line values.

Return Values

In C and C++ calls, the following applies:

0	indicates successful completion
-1	indicates that an error occurred. A message describing the error will be issued.

Errors

- 0033-3101 Setting Temp File size to threshold size. Set Size =, Minimum size = "

  You tried to set the size of the temporary file less than the minimum threshold size. The program automatically set the size to the minimum and continued trace generation.

  Change the parameters in VT_TRC_SET_PARAMS call above the threshold value.
- 0033-3103 Setting Buffer size to Threshold size. Set size = , Minimum size =

  You tried to set the size of the trace buffer less than the minimum threshold size. The program automatically set the size to the threshold size and continued trace generation.

  Change the parameters in VT_TRC_SET_PARAMS call above the threshold value.
- 0033-3104 Setting system statistics sampling frequency to the threshold value. Set Size = , Minimum size =

  You tried to set the sampling frequency to be less than the minimum threshold value. The program automatically set the size to the threshold value and continued trace generation.

  Change the parameters in VT_TRC_SET_PARAMS call above the threshold value.
- 0033-3011 VT_trc_set-params(), reallocation of ... bytes failed

  The application could not obtain the memory required to change parameters.
- 0033-3090 write failed to the dig daemon, ..., error is ...

  The application could not contact the kernel statistics sampling daemon.

For more information about error conditions, refer to *IBM Parallel Environment for AIX: Messages* .

Examples

C Example

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>
#include <VT_trc.h>

#define COUNT 1024

int main(int argc, char **argv)
{
  int i;
  int numtask;
  int taskid;
  int msg_in[COUNT], msg_out[COUNT];
  int src, dest, len, send_msgid, recv_msgid, nBytes;

  /* Find out number of tasks/nodes. */
  MPI_Init( &argc, &argv);
  MPI_Comm_size( MPI_COMM_WORLD, &numtask);
  MPI_Comm_rank( MPI_COMM_WORLD, &taskid);

  /*****************************************************

    Set tracing parameters as follows:
      Memory buffer = 500K
      Temporary trace file = 1M
      Sample every second
      Do not use wrap around buffer

   *****************************************************/

  if(0!=VT_trc_set_params_c(500000,1000000,1000,0))
    {
      printf("Could not reset trace parameters!\n");
    }
  /* Disable tracing while the message buffer is being initialized */
  if(0!=VT_trc_stop_c())
    {
      printf("Could not stop tracing!\n");
    }
  /* Flush the memory buffer to disk */
  if(0!=VT_trc_flush_c())
    {
      printf("Could not flush trace buffer!\n");
    }
  for(i=0;i<COUNT;i++)
    msg_out[i]=taskid;

  /* Re-enable tracing.  Level 9 asks for everything */
  /* but only events enabled by the command line or  */
  /* environment variable will be re-enabled)        */
  if(0!=VT_trc_start_c(9))
    {
      printf("Could not restart tracing!\n");
    }

  dest = (taskid<(numtask-1))?(taskid+1):0;
  MPI_Send(msg_out, COUNT, MPI_INT, dest, 0, MPI_COMM_WORLD);

  src = (taskid>0)?(taskid-1):(numtask-1);
  MPI_Recv(msg_in, COUNT, MPI_INT, src, 0, MPI_COMM_WORLD);

  MPI_Finalize();
}
```

Fortran Example

```
      PROGRAM TRCDEMO
C
      INCLUDE "mpif.h"
      IMPLICIT   NONE
      INTEGER    COUNT, I
      INTEGER    BUFFSZ, FILESZ
      INTEGER    SMPL, WRAP
      PARAMETER  ( COUNT = 1024 )
      PARAMETER  ( BUFFSZ = 500000, FILESZ = 1000000 )
```

```
        PARAMETER  ( SMPL = 1000, WRAP = 0 )
        INTEGER    MSG_IN(COUNT), MSG_OUT(COUNT)
        INTEGER    NUMTASK, TASKID
        INTEGER    SRC, DEST
        INTEGER    RC
C
C FIND OUT NUMBER OF TASKS

        CALL MPI_INIT( RC )
        CALL MPI_COMM_SIZE( MPI_COMM_WORLD, NUMTASK, RC )
        CALL MPI_COMM_RANK( MPI_COMM_WORLD, TASKID, RC )

C
C
C    SET TRACING PARAMETERS AS FOLLOWS:
C      MEMORY BUFFER = 500K
C      TEMPORARY TRACE FILE = 1M
C      SAMPLE EVERY SECOND
C      DO NOT USE WRAP AROUND BUFFER
C
C

        CALL VT_TRC_SET_PARAMS(BUFFSZ, FILESZ, SMPL, WRAP, RC)
        IF(RC .NE. 0) THEN
          WRITE(6,*)'Could not reset trace parameters!'
        ENDIF

C DISABLE TRACING WHILE THE MESSAGE BUFFER IS BEING INITIALIZED
        CALL VT_TRC_STOP( RC )
        IF(RC .NE. 0) THEN
          WRITE(6,*)'Could not stop tracing!'
        ENDIF

C FLUSH THE MEMORY BUFFER TO DISK
        CALL VT_TRC_FLUSH( RC )
        IF(RC .NE. 0) THEN
          WRITE(6,*)'Could not flush trace buffer!'
        ENDIF

        DO I = 1,COUNT
          MSG_OUT(I) = TASKID
        ENDDO

C RE-ENABLE TRACING.  LEVEL 9 ASKS FOR EVERYTHING
C BUT ONLY EVENTS ENABLED BY THE COMMAND LINE OR
C ENVIRONMENT VARIABLE WILL BE RE-ENABLED)
        CALL VT_TRC_START( 9, RC )
        IF(RC .NE. 0) THEN
          WRITE(6,*)'Could not restart tracing!'
        ENDIF

        IF( TASKID .GE. NUMTASK-1 ) THEN
          DEST = 0
        ELSE
          DEST = TASKID + 1
        ENDIF

        CALL MPI_SEND(MSG_OUT, COUNT, MPI_INTEGER, DEST, 0,
     +           MPI_COMM_WORLD, RC )

        IF( TASKID .LE. 0 ) THEN
          SRC = NUMTASK - 1
        ELSE
          SRC = TASKID - 1
        ENDIF

        CALL MPI_RECV(MSG_IN, COUNT, MPI_INTEGER, SRC, 0,
     +          MPI_COMM_WORLD, RC)

        CALL MPI_FINALIZE( RC )

        STOP
        END
C
```

Related Information

Functions:

- VT_TRC_FLUSH, VT_trc_flush_c
- VT_TRC_START, VT_trc_start_c
- VT_TRC_STOP, VT_trc_stop_c

For more information about VT tracing, see *IBM Parallel Environment for AIX: Operation and Use, Volume 2*.

VT_TRC_START, VT_trc_start_c

Purpose

Lets an application start a trace from within the program.

Version

- C Library (libvtd.a)
- Fortran Library (libvtd.a)

The above is automatically included by the POE functions.

C Synopsis

```
#include <VT_trc.h>
int VT_trc_start_c(int flag);
```

Fortran Synopsis

```
VT_TRC_START(INTEGER FLAG, INTEGER RETURN_CODE)
```

Parameters

flag the trace control flags that define what information is traced.

return_code integer which receives the return value.

0	indicates successful completion
-1	indicates an error occurred.

Description

This routine can be called by the application program. This lets the program start the trace from within the program, dynamically and selectively. The flag has the same meaning as the trace flag passed by PM upon startup. If the flag indicates VT_AIX_TRACE, this routine also sends a message to the monitoring daemon task.

Notes

- Tracing is initially on when the application starts.
- If a trace event, such as kernel statistics or communication events, was not selected at the start of the application it cannot be enabled by this routine.

Return Values

In C and C++ calls, the following applies:

0	indicates successful completion
-1	indicates that an error occurred. A message describing the error will be issued.

Errors

- 0033-3090 write failed to the dig daemon ...., error is ...

  The application could not contact the kernel statistics sampling daemon.

For more information about error conditions, see *IBM Parallel Environment for AIX: Messages*.

Examples

C Example

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>
#include <VT_trc.h>

#define COUNT 1024

int main(int argc, char **argv)
{
  int i;
  int numtask;
  int taskid;
  int msg_in[COUNT], msg_out[COUNT];
  int src, dest, len, send_msgid, recv_msgid, nBytes;

  /* Find out number of tasks/nodes. */
  MPI_Init( &argc, &argv);
  MPI_Comm_size( MPI_COMM_WORLD, &numtask);
  MPI_Comm_rank( MPI_COMM_WORLD, &taskid);

  /****************************************************

    Set tracing parameters as follows:
      Memory buffer = 500K
      Temporary trace file = 1M
      Sample every second
      Do not use wrap around buffer

   ****************************************************/

  if(0!=VT_trc_set_params_c(500000,1000000,1000,0))
    {
      printf("Could not reset trace parameters!\n");
    }
  /* Disable tracing while the message buffer is being initialized */
  if(0!=VT_trc_stop_c())
    {
      printf("Could not stop tracing!\n");
    }
  /* Flush the memory buffer to disk */
  if(0!=VT_trc_flush_c())
    {
      printf("Could not flush trace buffer!\n");
    }
  for(i=0;i<COUNT;i++)
    msg_out[i]=taskid;
  /* Re-enable tracing.  Level 9 asks for everything */
  /* but only events enabled by the command line or  */
  /* environment variable will be re-enabled)        */
  if(0!=VT_trc_start_c(9))
    {
      printf("Could not restart tracing!\n");
    }

  dest = (taskid<(numtask-1))?(taskid+1):0;
  MPI_Send(msg_out, COUNT, MPI_INT, dest, 0, MPI_COMM_WORLD);

  src = (taskid>0)?(taskid-1):(numtask-1);
  MPI_Recv(msg_in, COUNT, MPI_INT, src, 0, MPI_COMM_WORLD);

  MPI_Finalize();
}
```

Fortran Example

```
         PROGRAM TRCDEMO
 C
         INCLUDE "mpif.h"
         IMPLICIT   NONE
         INTEGER    COUNT, I
         INTEGER    BUFFSZ, FILESZ
         INTEGER    SMPL, WRAP
         PARAMETER  ( COUNT = 1024 )
         PARAMETER  ( BUFFSZ = 500000, FILESZ = 1000000 )
         PARAMETER  ( SMPL = 1000, WRAP = 0 )
         INTEGER    MSG_IN(COUNT), MSG_OUT(COUNT)
         INTEGER    NUMTASK, TASKID
         INTEGER    SRC, DEST
         INTEGER    RC
 C
 C FIND OUT NUMBER OF TASKS

         CALL MPI_INIT( RC )
         CALL MPI_COMM_SIZE( MPI_COMM_WORLD, NUMTASK, RC )
         CALL MPI_COMM_RANK( MPI_COMM_WORLD, TASKID, RC )

 C
 C
 C    SET TRACING PARAMETERS AS FOLLOWS:
 C      MEMORY BUFFER = 500K
 C      TEMPORARY TRACE FILE = 1M
 C      SAMPLE EVERY SECOND
 C      DO NOT USE WRAP AROUND BUFFER
 C
 C

         CALL VT_TRC_SET_PARAMS(BUFFSZ, FILESZ, SMPL, WRAP, RC)
         IF(RC .NE. 0) THEN
           WRITE(6,*)'Could not reset trace parameters!'
         ENDIF

 C DISABLE TRACING WHILE THE MESSAGE BUFFER IS BEING INITIALIZED
         CALL VT_TRC_STOP( RC )
```

```
        IF(RC .NE. 0) THEN
          WRITE(6,*)'Could not stop tracing!'
        ENDIF

C FLUSH THE MEMORY BUFFER TO DISK
        CALL VT_TRC_FLUSH( RC )
        IF(RC .NE. 0) THEN
          WRITE(6,*)'Could not flush trace buffer!'
        ENDIF

        DO I = 1,COUNT
          MSG_OUT(I) = TASKID
        ENDDO

C RE-ENABLE TRACING.  LEVEL 9 ASKS FOR EVERYTHING
C BUT ONLY EVENTS ENABLED BY THE COMMAND LINE OR
C ENVIRONMENT VARIABLE WILL BE RE-ENABLED)
        CALL VT_TRC_START( 9, RC )
        IF(RC .NE. 0) THEN
          WRITE(6,*)'Could not restart tracing!'
        ENDIF

        IF( TASKID .GE. NUMTASK-1 ) THEN
          DEST = 0
        ELSE
          DEST = TASKID + 1
        ENDIF

        CALL MPI_SEND(MSG_OUT, COUNT, MPI_INTEGER, DEST, 0,
     +           MPI_COMM_WORLD, RC )

        IF( TASKID .LE. 0 ) THEN
          SRC = NUMTASK - 1
        ELSE
          SRC = TASKID - 1
        ENDIF

        CALL MPI_RECV(MSG_IN, COUNT, MPI_INTEGER, SRC, 0,
     +          MPI_COMM_WORLD, RC)

        CALL MPI_FINALIZE( RC )

        STOP
        END
C
```

Related Information

Functions:

- VT_TRC_FLUSH, VT_trc_flush_c
- VT_TRC_SET_PARAMS, VT_trc_setparams_c
- VT_TRC_STOP, VT_trc_stop_c

For more information about VT tracing, see *IBM Parallel Environment for AIX: Operation and Use, Volume 2*.

VT_TRC_STOP, VT_trc_stop_c

Purpose

Stops collecting trace events.

Version

- C Library (libvtd.a)
- Fortran Library (libvtd.a)

The above is automatically included by the POE functions.

C Synopsis

```
#include <VT_trc.h>
int VT_trc_stop_c();
```

Fortran Synopsis

```
VT_TRC_STOP(INTEGER RETURN_CODE)
```

Parameters

return_code integer which receives the return value.

0	indicates successful completion
-1	indicates an error occurred.

Description

This routine can be called by the application program to stop tracing.

Return Values

In C and C++ calls, the following applies:

0	indicates successful completion
-1	indicates that an error occurred. A message describing the error will be issued.

Errors

- 0033-3090 write failed to the dig daemon ...., error is ...

  The application could not contact the kernel statistics sampling daemon.

For more information about error conditions, see *IBM Parallel Environment for AIX: Messages* .

Examples

C Example

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>
#include <VT_trc.h>

#define COUNT 1024

int main(int argc, char **argv)
{
  int i;
  int numtask;
  int taskid;
  int msg_in[COUNT], msg_out[COUNT];
  int src, dest, len, send_msgid, recv_msgid, nBytes;

  /* Find out number of tasks/nodes. */
  MPI_Init( &argc, &argv);
  MPI_Comm_size( MPI_COMM_WORLD, &numtask);
  MPI_Comm_rank( MPI_COMM_WORLD, &taskid);

  /****************************************************

    Set tracing parameters as follows:
      Memory buffer = 500K
      Temporary trace file = 1M
      Sample every second
      Do not use wrap around buffer

   ****************************************************/

  if(0!=VT_trc_set_params_c(500000,1000000,1000,0))
    {
      printf("Could not reset trace parameters!\n");
    }
  /* Disable tracing while the message buffer is being initialized */
  if(0!=VT_trc_stop_c())
    {
      printf("Could not stop tracing!\n");
    }
  /* Flush the memory buffer to disk */
  if(0!=VT_trc_flush_c())
    {
      printf("Could not flush trace buffer!\n");
    }
  for(i=0;i<COUNT;i++)
    msg_out[i]=taskid;

  /* Re-enable tracing.  Level 9 asks for everything */
  /* but only events enabled by the command line or  */
  /* environment variable will be re-enabled)        */
  if(0!=VT_trc_start_c(9))
    {
      printf("Could not restart tracing!\n");
    }

  dest = (taskid<(numtask-1))?(taskid+1):0;
  MPI_Send(msg_out, COUNT, MPI_INT, dest, 0, MPI_COMM_WORLD);
  src = (taskid>0)?(taskid-1):(numtask-1);
  MPI_Recv(msg_in, COUNT, MPI_INT, src, 0, MPI_COMM_WORLD);

  MPI_Finalize();
}
```

Fortran Example

```
        PROGRAM TRCDEMO
C
        INCLUDE "mpif.h"
        IMPLICIT   NONE
        INTEGER    COUNT, I
        INTEGER    BUFFSZ, FILESZ
        INTEGER    SMPL, WRAP
        PARAMETER  ( COUNT = 1024 )
        PARAMETER  ( BUFFSZ = 500000, FILESZ = 1000000 )
        PARAMETER  ( SMPL = 1000, WRAP = 0 )
        INTEGER    MSG_IN(COUNT), MSG_OUT(COUNT)
        INTEGER    NUMTASK, TASKID
        INTEGER    SRC, DEST
        INTEGER    RC
C
C FIND OUT NUMBER OF TASKS

        CALL MPI_INIT( RC )
        CALL MPI_COMM_SIZE( MPI_COMM_WORLD, NUMTASK, RC )
        CALL MPI_COMM_RANK( MPI_COMM_WORLD, TASKID, RC )

C
C
C    SET TRACING PARAMETERS AS FOLLOWS:
C      MEMORY BUFFER = 500K
C      TEMPORARY TRACE FILE = 1M
C      SAMPLE EVERY SECOND
C      DO NOT USE WRAP AROUND BUFFER
C
C

        CALL VT_TRC_SET_PARAMS(BUFFSZ, FILESZ, SMPL, WRAP, RC)
        IF(RC .NE. 0) THEN
          WRITE(6,*)'Could not reset trace parameters!'
        ENDIF

C DISABLE TRACING WHILE THE MESSAGE BUFFER IS BEING INITIALIZED
        CALL VT_TRC_STOP( RC )
        IF(RC .NE. 0) THEN
          WRITE(6,*)'Could not stop tracing!'
        ENDIF

C FLUSH THE MEMORY BUFFER TO DISK
        CALL VT_TRC_FLUSH( RC )
        IF(RC .NE. 0) THEN
          WRITE(6,*)'Could not flush trace buffer!'
        ENDIF

        DO I = 1,COUNT
          MSG_OUT(I) = TASKID
        ENDDO

C RE-ENABLE TRACING.  LEVEL 9 ASKS FOR EVERYTHING
C BUT ONLY EVENTS ENABLED BY THE COMMAND LINE OR
C ENVIRONMENT VARIABLE WILL BE RE-ENABLED)
        CALL VT_TRC_START( 9, RC )
        IF(RC .NE. 0) THEN
          WRITE(6,*)'Could not restart tracing!'
        ENDIF

        IF( TASKID .GE. NUMTASK-1 ) THEN
          DEST = 0
        ELSE
          DEST = TASKID + 1
        ENDIF

        CALL MPI_SEND(MSG_OUT, COUNT, MPI_INTEGER, DEST, 0,
     +           MPI_COMM_WORLD, RC )

        IF( TASKID .LE. 0 ) THEN
          SRC = NUMTASK - 1
        ELSE
          SRC = TASKID - 1
        ENDIF

        CALL MPI_RECV(MSG_IN, COUNT, MPI_INTEGER, SRC, 0,
     +          MPI_COMM_WORLD, RC)

        CALL MPI_FINALIZE( RC )

        STOP
        END
C
```

Related Information

Functions:

- VT_TRC_FLUSH, VT_trc_flush_c
- VT_TRC_SETPARAMS, VT_trc_setparams_c
- VT_TRC_START, VT_trc_start_c

For more information about VT tracing, see *IBM Parallel Environment for AIX: Operation and Use, Volume 2*.

Appendix G. Programming Considerations for User Applications in POE

This appendix documents various limitations, restrictions, and programming considerations for user applications written to run under the IBM Parallel Environment for AIX (PE) licensed program.

PE includes two versions of the message passing libraries. These are called the signal-handling library and the threaded library.

- The signal-handling library uses AIX signals as an asynchronous way to move data in and out of message buffers. They also ensure that message packets are acknowledged and retransmitted when necessary. It supports both MPL and MPI calls.
- The threaded library uses AIX kernel threads for the same message passing tasks. It supports MPI only. The threaded library also supports message passing on user-created threads. The threaded library is required if MPI coexists with the other user space protocols, for example, the LAPI interface on the IBM RS/6000 SP.

This appendix consists of sections that list the programming considerations common to both libraries, as well as those unique to either the signal-handling library or the threaded library. There is also a subsection on using POE and the Fortran compiler. Specifically, the sections are as follows:

- “MPI Signal-Handling and MPI Threaded Library Considerations”
- “MPI Signal-Handling Library Considerations” on page 421
- “MPI Threaded Library Considerations” on page 424
- “Fortran Considerations” on page 428

MPI Signal-Handling and MPI Threaded Library Considerations

The information in this section pertains to both the (MPL/MPI) signal-handling library and the MPI threaded library.

Environment Overview

As the end user, you are encouraged to think of the Parallel Operating Environment(POE) (also referred to as the **poe** command) as an ordinary (serial) command. It accepts redirected I/O, can be run under the **nice** and **time** commands, interprets command flags, and can be invoked in shell scripts.

An *n*-task parallel job running in the Parallel Operating Environment actually consists of the *n* user tasks, an equal number (*n*) of instances of the IBM Parallel Environment for AIX **pmd** daemon (which is the parent task of the user's task), and the POE home node task in which the **poe** command runs. A **pmd** daemon is started by the POE home node on each machine on which each user task runs, and serves as the point of contact between the home node and the user's tasks.

The POE home node routes standard input, standard output and standard error streams between the home node and the user's tasks via the **pmd** daemon, using TCP/IP sockets for this purpose. The sockets are created when the POE home node starts the **pmd** daemon for each task of a parallel job. The POE home node and **pmd** also use the sockets to exchange control messages to provide task synchronization, exit status and signaling. These capabilities do not depend upon the message passing library and are available to control any parallel program run by the **poe** command.

Exit Status

Exit status is a value between 0 and 255 inclusive. It is returned from POE on the home node reflecting the composite exit status of your parallel application, as follows:

- If MPI_ABORT(comm,nn>0,ierror) or MPI_Abort(comm,nn>0) is called, the exit status is nn (mod 256).
- If MP_STOPALL(nn>=0) or mpc_stopall(nn>=0) is called, the exit status is nn (mod 256). This does not apply to threaded libraries.
- If all tasks terminate via exit(MM>=0) or STOP MM>=0 and MM is not equal to 1 and is <128 for all nodes, then POE provides a synchronization barrier at the exit. The exit status is the largest value of MM from any parallel job (mod 256).
- If any task terminates via exit(MM =1) or STOP MM =1, then POE will immediately terminate the parallel job, as if MP_STOPALL(1) or MPI_Abort(MPI_COMM_WORLD,1) had been called. This may also occur if a Fortran I/O library error occurs.
- If any task terminates via a signal (for example, a segment violation), the exit status is 128+signal and the entire job is immediately terminated.
- If POE terminates before the start of the user's application, the exit status is =1.
- If the user's application cannot be loaded or fails before the user's main() is called, the exit status is =255.
- You should explicitly call exit(MM) or STOP MM to set the desired exit code. A program exiting without an explicit exit value returns unpredictable status, and may result in causing premature termination of the parallel application.

POE Job Step Function

The POE job-step function is intended for the execution of a sequence of separate yet inter-related dependent programs. Therefore, it provides you with a job control mechanism that allows both job-step progression and job-step termination. The job control mechanism is the program's exit code.

- Job-step progression:

  POE continues the job-step sequence if the task exit code is 0 or in the range of 2 - 127.

- Job-step termination:

  POE terminates the parallel job, and does not execute any remaining user programs in the job-step list if the task exit code is 1 or greater than 127.

- Default termination:

  Any POE infrastructure detected failure (such as failure to open pipes to the child task or an exec failure to start the user's executable) terminates the parallel job, and does not execute any remaining user programs in the job-step queue.

POE Additions To The User Executable

POE links in the following routines when your executable is compiled with any of the POE compilation scripts (mpcc, mpcc_r, mpxlf,etc.).

Signal Handlers

POE installs signal handlers for most signals that cause program termination in order to notify the other tasks of termination and to complete the VT trace file, if enabled. POE then causes the program to exit normally with a code of (128+signal). When running non-threaded applications under POE, you may install a signal handler for any of these signals, and it should call the POE registered signal handler if the task decides to terminate. (See “Let POE Handle Signals When Possible” on page 415.) When running threaded applications, any attempt to install a signal handler is ignored.

Signals that are specifically handled by POE or the message passing library follow:

- SIGHUP

  Caught and exits with an exit code of 128+SIGHUP.

- SIGINT

  Caught and exits with an exit code of 128+SIGINT.

  Note: This signal may be caught by user or by dbx, in which case this usage is ignored.

- SIGQUIT

  Caught, sets the default signal handler and calls exit handler with an exit code of 128+SIGQUIT. The exit handler dumps the user's context and takes the default signal action.

- SIGFPE

  Caught, sets the default signal handler and calls exit handler with an exit code of 128+SIGFPE. The exit handler dumps the user's context and takes the default signal action.

- SIGSEGV

  Caught, sets the default signal handler and calls exit handler with an exit code of 128+SIGSEGV. The exit handler dumps the user's context and takes the default signal action.

- SIGBUS

  Caught, sets the default signal handler and calls exit handler with an exit code of 128+SIGBUS. The exit handler dumps the user's context and takes the default signal action.

- SIGTERM

Caught and exits with an exit code of 128+SIGTERM. This is also used by POE to signal orderly termination of a parallel job. If it must be caught by the user, please read carefully the section on program termination (below).

- SIGSTOP

  Default action (cannot be caught)

- SIGTSTP

  Default action

- SIGCONT

  Default action

- SIGPWR

  Caught, sets the default signal handler and calls exit handler with an exit code of 128+SIGPWR. The exit handler dumps the user's context and takes the default signal action.

- SIGDANGER

  Caught and exits with an exit code of 128+SIGDANGER.

The signal-handling library uses SIGIO, SIGALRM and SIGPIPE for its operations and it also handles these signals. For more information about the signal-handling
| library, see “MPI Signal-Handling Library Considerations” on page 421. For more
| information about signals, see “Use of AIX Signals” on page 425.

Replacement exit/atexit

POE requires its own versions of the library exit()/atexit() functions, and expects to load them dynamically from its own version of **libc.a** (or **libc_r.a**) in **/usr/lpp/ppe.poe/lib**; therefore, do not code your own exit function to override the library function. This is to synchronize profiling and to provide barrier synchronization upon exit.

Let POE Handle Signals When Possible

Programs that handle signals must coordinate with POE's handling of most of the common signals (see above).

DO NOT issue message passing calls from signal handlers. Also, many AIX library calls are not "signal safe", and should not be issued from signal handlers. Check the AIX Technical Reference (function sigaction()) for a list of AIX functions callable from signal handlers.

POE sets up signal handlers for all the signals that normally terminate program execution. It does this so that it can terminate the entire parallel job in an orderly fashion if one task terminates abnormally (via signal). A user program may install a handler for any or all of these signals, but should save the address of the POE signal handler. If the user program decides to terminate, it should call the POE signal handler. If the user program decides not to terminate, it should just return to the interrupted code. SIGTERM is used by POE to shutdown the parallel job in a variety of abnormal circumstances, and should be allowed to terminate the job.

The POE home node converts a user's SIGTSTP signal (Ctrl-z) to a SIGSTOP signal to all the remote nodes, and passes the SIGCONT signal sent by the **fg** or **bg** command to all the remote nodes to restart the job.

Don't Hard Code File Descriptor Numbers

Do not use hard coded file descriptor numbers beyond those specified by STDIN, STDOUT and STDERR.

POE opens several files and uses file descriptors as message passing handles. These are allocated before the user gets control, so the first file descriptor allocated to a user is unpredictable.

Termination Of A Parallel Job

POE provides for orderly termination of a parallel job, so that all tasks terminate at the same time. This is accomplished in the atexit routine registered at program initialization. For normal exits (codes 0, 2-127), the atexit routine sends a control message to the POE home node, and waits for a positive response. For abnormal exits and those which don't go through the atexit routine, the **pmd** daemon catches the exit code and sends a control message to the POE home node.

For normal exits, when POE gets a control message for every task, it responds to each node, allowing that node to exit normally with its individual exit code. The **pmd** daemon monitors the exit code and passes it back to the POE home node for presentation to the user.

For abnormal exits and those detected by **pmd**, POE sends a message to each **pmd** asking that it send a SIGTERM signal to its task, thereby terminating the task. When the task finally exits, **pmd** sends its exit code back to the POE home node and exits itself.

User-initiated termination of the POE home node via SIGINT (Ctrl-c) and/or SIGQUIT (Ctrl-\) causes a message to be sent to **pmd** asking that the appropriate signal be sent to the parallel task. Again, **pmd** waits for the task to die then terminates itself.

Your Program Can't Run As Root

To prevent uncontrolled root access to the entire parallel job computation resource, POE checks to see that the user is not root as part of its authentication.

AIX Function Limitations

The use of the following AIX functions may be limited, but no formal testing has been done:

- wide character sets
- shared memory - the message passing library uses shared memory for adapter mapping. You can use the remaining data segments as desired.
- getuinfo does not show terminal information, since the user program running in the parallel partition does not have an attached terminal.

Shell Execution

You can have POE run a shell script which is loaded and run on the remote nodes as if it were a binary file.

If the POE home node task is not started under the Korn shell, mounted file system names may not be mapped correctly to the names defined for the automount daemon or AIX equivalent running on the IBM RS/6000 SP. See the *IBM Parallel Environment for AIX: Operation and Use, Volume 1* for a discussion of alternative name mapping techniques.

The program executed by POE on the parallel nodes does not run under a shell on those nodes. Redirection and piping of STDIO applies to the POE home node (**poe** binary), and not the user's code. If shell processing of a command line is desired on the remote nodes, invoke a shell script on the remote nodes to provide the desired preprocessing before the user's application is executed.

Do Not Rewind stdin, stdout Or stderr

The partition manager daemon uses pipes to direct stdin, stdout and stderr to the user's program, therefore, do not rewind these files.

Ensuring String Arguments Are Passed To Your Program Correctly

Quotation marks, either single or double, used as argument delimiters are stripped away by the shell and are never "seen" by poe. Therefore, the quotation marks must be escaped to allow the quoted string to be passed correctly to the remote task(s) as one argument. For example, if you want to pass the following string to the user program (including the imbedded blank)

```
a b
```

then you need to enter the following:

```
poe user_program \"a b\"
```

user_program is passed the following argument as one token:

```
a b
```

Without the backslashes, the string would have been treated as two arguments (a *and* b).

POE behaves like rsh when arguments are passed to POE. Therefore, the following:

```
poe user_program "a b"
```

is equivalent to:

```
rsh some_machine user_program "a b"
```

In order to pass the string argument as one token, the quotes have to be escaped.

Network Tuning Considerations

Programs generating large volumes of STDOUT or STDERR may overload the home node. As described previously, standard output and standard error files generated by a user's program are piped to **pmd**, then forwarded to the poe binary via a TCP/IP socket. It is possible to generate so much data that the IP message buffers on the home node are exhausted, the poe binary hangs and possibly the entire node may hang). Note that the option -stdoutmode (environment variable MP_STDOUTMODE) controls which output stream is displayed by the poe binary, but does not limit the standard output traffic received from the remote nodes, even if set to display the output of just one node.

The POE environment variable MP_SNDBUF can be used to override the default network settings for the size of the TCP buffers used.

If you have large volumes of standard I/O, work with your network administrator to establish appropriate TCP/IP tuning parameters. You may also want to examine if using named pipes is appropriate for your application.

Standard I/O Requires Special Attention

When your program runs on the remote nodes, it has no controlling terminal. STDIN and STDOUT, STDERR are always piped.

Programs that depend on piping standard input or standard output as part of a processing sequence may wish to bypass the home node poe binary. Running the **poe** command (or starting a program compiled with one of the POE compile scripts) causes the poe binary to be loaded on the machine on which you typed the command (the POE home node). The poe binary, in turn, starts a daemon named **pmd** on each parallel node assigned to run the job, and then requests **pmd** to run your executable (via fork and exec). The poe binary reads STDIN and passes it to each of the parallel tasks via a TCP/IP socket connection to the **pmd** daemon, which pipes it to the user. Similarly, STDOUT and STDERR from the user are piped to **pmd** and sent on the socket back to the home node, where it is written to the poe binary's STDOUT and STDERR descriptors. If you know that the task reading STDIN or writing STDOUT must be on the same node (processor) as the poe binary (the poe home node), named pipes can be used to bypass poe's reading and forwarding STDIN and STDOUT.

If STDIN is piped or redirected to the poe binary (via ordinary pipes), and your application is linked with the signal handling message passing library, (via mpcc, mpxlf, or mpCC), then set the environment variable MP_HOLD_STDIN to "yes". This lets poe initialize the signal-handling library before handling the STDIN file.

If your application is linked with the threaded library, see “Standard I/O Requires Special Attention” on page 427 for more information.

STDIN/STDOUT Piping Example

The following two scripts show how STDIN and STDOUT can be piped directly between pre- and post-processing steps, bypassing the POE home node task. This example assumes that parallel task 0 is known or forced to be on the same node as the POE home node.

The script compute_home runs on the home node; the script compute_parallel runs on the parallel nodes (those running tasks 0 through *n*-1).

```
compute_home:
#! /bin/ksh
# Example script compute_home runs three tasks:
#    data_generator creates/gets data and writes to stdout
#    data_processor is a parallel program that reads data
#      from stdin, processes it in parallel, and writes
#      the results to stdout.
#    data_consumer reads data from stdin and summarizes it
#
mkfifo poe_in_$$
mkfifo poe_out_$$
export MP_STDOUTMODE=0
export MP_STDINMODE=0
data_generator >poe_in_$$ |
     poe compute_parallel poe_in_$$ poe_out_$$ data_processor |
     data_consumer <poe_out_$$
 rc=$?
 rm poe_in_$$
 rm poe_out_$$
 exit rc
```

```
compute_parallel:
#! /bin/ksh
# Example script compute_parallel is a shell script that
#    takes the following arguments:
#    1) name of input named pipe (stdin)
#    2) name of output named pipe (stdout)
#    3) name of program to be run (and arguments)
#
poe_in=$1
poe_out=$2
shift 2
$*   <$poe_in   >$poe_out
```

Reserved Environment Variables

Environment variables starting with MP_ are intended for use by POE, and should be set only as instructed in the documentation. POE also uses a handful of MP_... environment variables for internal purposes, which should not be interfered with.

AIX Message Catalog Considerations

POE assumes that NLSPATH contains the appropriate POE message catalogs, even if LANG is set to "C" or is unset. Duplicate message catalogs are provided for languages "En_US", "en_US", and "C".

Language Bindings

The Fortran, C and C++ bindings for MPI are contained in the same library and can be freely intermixed.

- libmpi.a for the signal-handling version
- libmpi_r.a for the threaded version

Refer to “Fortran Considerations” on page 428 for more information about the Fortran compiler.

The AIX compilers support the flag -qarch. This option allows you to target code generation to a particular processor architecture. While this option can provide performance enhancements on specific platforms, it inhibits portability, particularly between the Power and PowerPC machines. The MPI library is not targeted to a specific architecture and is the same on PowerPC and Power nodes.

| The MPI-IO functions from MPI-2 are only available with the threaded library.

| Available Virtual Memory Segments

| AIX makes available up to 11 additional address segments for end user programs.
| The MPI libraries use some of these as listed in Table 16. The remaining are
| available to the user for either extended heap (-bmaxdata option) or shared
| memory (shmget). Very large jobs, which include all jobs with more than 1000
| tasks, will need to use the -bmaxdata option to ensure a large enough heap.

| *Table 16. Memory Segments Used By the MPI and LAPI Libraries*

Component	RS/6000 SP node with switch	RS/6000 workstation or no switch
MPI User Space	2	not available
MPI IP	1*	0
VT Trace Capture	1	0
LAPI User Space	2	not available

| * If the environment variable MP_CLOCK_SOURCE=AIX, the value is 0.

| Using the SP Switch Clock as a Time Source

| The RS/6000 SP switch clock is a globally-synchronized counter that may be used
| as a source for the MPI_WTIME function, provided that all tasks are run on nodes
| of the same SP system. The environment variable MP_CLOCK_SOURCE provides
| additional control. Table 17 shows how the clock source is determined. MPI
| guarantees that MPI_WTIME_IS_GLOBAL has the same value at every task.

| *Table 17 (Page 1 of 2). How the Clock Source Is Determined*

MP_CLOCK_SOURCE	Library Version	All Nodes SP?	Source Used	MPI_WTIME_IS_GLOBAL
not set	ip	yes	switch	false
		no	AIX	false
	us	yes	switch	true
		no	Error	
SWITCH	ip	yes*	switch	false
		no	AIX	false
	us	yes	switch	true
		no	Error	
AIX	ip	yes	AIX	false
		no	AIX	false
	us	yes	AIX	false

Table 17 (Page 2 of 2). How the Clock Source Is Determined				
MP_CLOCK_SOURCE	**Library Version**	**All Nodes SP?**	**Source Used**	**MPI_WTIME_IS_GLOBAL**
		no	AIX	false

* The user is responsible for ensuring all of the nodes are in the same SP system.

32-Bit and 64-Bit Support

POE compiles and runs all applications as 32-bit applications. 64-bit applications are not supported yet.

Running Applications With Large Numbers of Tasks

If you plan to run your parallel applications with a large number of tasks (more than 256), the following tips may improve stability and performance:

- Use a host list file with the switch IP names, instead of the IP host name.
- You may avoid a potential problem running out of memory by linking applications with a data buffer using data segment three (3), by specifying the **-bD:0x3000000** loader option. The default is to use data segment zero.
- To avoid potential problems opening sockets, increase the user resource limit for the number of open file descriptors (nofiles) to at least 10,000, using the **ulimit** command. For example:

```
ulimit -n 10000
```

MPI Signal-Handling Library Considerations

The information in this subsection provides you with specific additional programming considerations for when you are using POE and the MPL/MPI signal-handling library.

POE Gets Control First And Handles Task Initialization

POE sets up its environment environment via the entry point mp_main(). mp_main() initializes the message passing library, sets up signal handlers, sets up an atexit routine, and initializes VT trace data collection before calling your main program.

Using Message Passing Handlers

Only a subset of MPL message passing is allowed on handlers created by the MPL Receive and Call function (mpc_rcvncall or MP_RCNVCALL). MPI calls on these handlers are not supported.

POE Additions To The User Executable

POE links in the following routines when your executable is compiled with mpcc, mpxlf or mpCC. These are routines specific for the signal handling environment.

Message Passing Initialization Module

POE initializes the parallel message passing library and determines that all nodes can communicate successfully before the user main() program gains control. As a result, any program compiled with the POE compiler scripts must be run under the control of POE and is not suitable as a serial program.

If communication initialization fails, the parallel task is terminated with an appropriate exit code.

Signal Handlers

The message passing library sets up signal handlers for SIGALRM, SIGIO and SIGPIPE to manage message passing activity. A user program may install a handler for any or all of these signals, but should save the address of and invoke the POE signal handler before returning to the interrupted code. The sigaction() function returns the required structure. Also, set SA_RESTART as well as the mask so all signals are masked when the signal handler is running.

The following are the signals used and specifically handled by the message passing library in a signal handling environment:

- SIGPIPE

  Caught by the non-threaded User Space message passing library to manage the RS/6000 SP switch. If your application catches this signal, it should call the registered message passing signal handler before returning to the main code.

  Do not block this signal for more than a few milliseconds.

- SIGALRM

  Caught by message passing library to manage message traffic. If you provide your own interval timing mechanism, then you should arrange to call the POE signal handler approximately every 200-800 milliseconds. Message passing calls from user programs may be blocked until the POE signal handler is called.

  If the user application catches this signal but doesn't do interval timing, it should call the registered message passing signal handler before returning to the main code.

- SIGIO

  Caught by the user space message passing library to manage message traffic. If your application catches this signal, it should call the registered message passing signal handler before returning to the main code.

Interrupted System Calls

The message passing library uses an interval timer to manage message traffic, specifically to ensure that messages progress even when message passing calls are not being made. When this interval timer expires, a SIGALRM signal is sent to the program, interrupting whatever computation is in progress. The message passing library has a signal handler set, and normally handles the signal and returns to the user's program without the program's knowledge. However, the following library and system calls are interrupted and do not complete normally. The user is responsible for testing whether an interrupt occurred and recovering from the interrupt. In many cases, this is accomplished by just retrying the call.

- sleep(see note below)/usleep/nsleep
- select

- open/close/fopen/fclose
- pause
- sigpause
- accept
- connect
- recv/recvfrom/recvmsg
- send/sendto/sendmsg
- aio_read/aio_write/aio_suspend
- fork
- system
- exec/execv/...
- msem_lock/semop
- AIX msg... routines
- poll

Note: The normal timer interval is less than 500 milliseconds. So a sleep call (with time specified in seconds) returns to the original sleep interval, due to rounding, and can't be used to determine how much time remains in the interval. You should use the functions usleep and nsleep instead. See also the “Sample Replacement Sleep Program” on page 431 in Appendix H, “Using Signals and the IBM PE Programs” on page 431.

With the exception of sleep, system and exec, the routines listed above set the system error indicator (the variable errno) to EINTR, which can be tested by the user's program. See the “Sample Replacement Select Program” on page 431 in Appendix H, “Using Signals and the IBM PE Programs” on page 431.

Normal file read and write are restarted automatically by AIX, and should not require any special treatment by the user.

The **system** and **fork** calls create a new task in which the interval timer is still running. If a fork is followed by an exec (which is what **system** does), the signal handler for the timer is overlaid, and the task is terminated when the interval timer expires.

To handle this for the **system** call, temporarily turn the interval timer off (using the **alarm**(0) call) before the call, and turn it on again (**ualarm**(500000, 500000) will do) after the **system** call.

To handle the interval timer for a forked child, merely turn off the interval timer via **alarm**(0) in the child.

There are other restrictions on fork described below.

Forks Are Limited

As described earlier, if a task forks, the forked child inherits the running timer. The timer should be turned off before forking another program. If the forked child does not exec another program, it should be aware that an atexit routine has been registered for the parent which is also inherited by the child. In most cases, the atexit routine will request POE to terminate the task (parent). A forked child should terminate with an _exit(0) system call to prevent the atexit routine from being called. Also, if the forked parent terminates before the child, the child task will not be cleaned up by POE.

A forked child *must not* call the message passing library.

Checkpoint/Restart Limitations

A user may initiate a checkpoint sequence from within a parallel MPI program by calling the MP_CHKPT function. All tasks in the parallel job must issue the call, which does not return until the checkpoint files have been created for all tasks. If the job subsequently fails and is restarted, the restart returns from the MP_CHKPT function with an indication that the parallel job has been restarted.

Programs using the signal handling (non-threaded) MPI library may be linked as a checkpointable executable, which is run as a LoadLeveler batch job. LoadLeveler Version 2.1 or later is required. Restrictions on the program follow:

- For some processes, it is impossible to obtain or recreate the state of the process. For this reason, you should only checkpoint programs with states that are simple to checkpoint and recreate. A program that is long-running, computation-intensive, and does not fork any processes is an example of a job that is well-suited for checkpointing.
- In order to prevent unpredictable results from occurring, checkpointing jobs should not use the following system services:
  - Administrative (**audit** and **swapqry**, for example)
  - Dynamic loading
  - Forks
  - Internal timers
  - Messages
  - Semaphores
  - Set user ID or group ID
  - Shared memory
  - Signals
  - Threads
- Another limitation of checkpointing jobs is file I/O. Because individual write calls are not traced, the file recovery scheme requires that all I/O operations, when repeated, must yield the same result. A job that opens all files as read-only can be checkpointed. A job that writes to a file and then reads the data back can also be checkpointed. An example of I/O that could cause unpredictable results is: reading an area of a file, writing to it, and then reading the same area of the file again.

MPI Threaded Library Considerations

When programming in a threaded environment specific skills and considerations are required. The information in this subsection provides you with specific programming considerations when using POE and the MPI threaded library. It assumes you are familiar with POSIX threads in general including mutexes, thread condition waiting, thread-specific storage, thread creation and termination.

POE Gets Control First And Handles Task Initialization

POE sets up its environment via the entry point mp_main_r(). mp_main_r() sets up signal handlers, initializes VT, and sets up an atexit routine before calling your main program.

Note: In the threaded library, message passing initialization takes place when MPI_INIT is called and not by mp_main_r. The threaded library and the signal-handling library differ significantly in this regard.

Language Bindings

The Fortran, C and C++ bindings for MPI are contained in the same library (libmpi_r.a) and can be freely intermixed.

Refer to “Fortran Considerations” on page 428 for more information about running Fortran programs in a threaded environment.

MPI-IO Requires GPFS To Be Used Effectively

The subset implementation of MPI-IO provided in the thread library depends on all tasks running on a single file system. IBM Generalized Parallel File System (GPFS) is able to present a single file system to all nodes of an SP. Shared file systems (NFS and AFS, for example) do not have the same rigorous management of file consistency when updates occur from more than one node.

MPI-IO can be used with most file systems as long as all tasks are on a single node. This single node approach may be useful in learning to use MPI-IO, but is not likely to be worthwhile in any production context.

Any production use of MPI-IO must be based on GPFS.

Use of AIX Signals

The threaded POE run-time environment creates a thread to handle the following asynchronous signals:

- SIGQUIT
- SIGPWR
- SIGDANGER
- SIGTSTP
- SIGTERM
- SIGHUP
- SIGINT

A user signal handler must not be invoked to handle the above signals, which are handled by **sigwait**.

The following signals, which are used by MPI in the non-threaded library, are handled as described below.

SIGALRM

The threaded library does not use SIGALRM and *long* system calls such as **sleep** are not interrupted by the message passing library. For example, **sleep** runs its entire duration unless interrupted by a user-generated event.

SIGIO

PE blocks SIGIO before calling your program. SIGIO is used in the IP version of the library to notify you of an I/O event or the arrival of a message packet. This notification is enabled via the environment variable MP_CSS_INTERRUPT. If this environment variable is set to YES, the message packet arrival dispatches the interrupt service thread to process the packet.

The User Space version of the library receives notification of an arriving packet via an AIX kernel event and does not use SIGO. You may unblock it or use sigwait to process SIGIO signals.

If you've registered a signal handler (via sigaction) for SIGIO before MPI_INIT is called, the function is added to the interrupt service thread and is executed each time the service thread is dispatched. Although registered as a signal handler, the function is not required to be signal safe because it is executed on a thread. You can use pthread calls to communicate with other threads. You cannot call MPI functions in this handler.

After MPI_FINALIZE is called, your signal handler is restored but you need to unblock SIGIO in order to receive subsequent SIGIO signals.

If you register or change the SIGIO signal handler after calling MPI_INIT, your changes are ignored by the MPI library but your changes are not undone by MPI_FINALIZE.

SIGPIPE

Neither the threaded or non-threaded IP libraries use SIGPIPE. The threaded User Space library polls a variable set by the AIX kernel to determine if the switch has faulted and needs to be restarted. As a result, it does not use SIGPIPE.

Limitations In Setting The Thread Stacksize

The main thread stacksize is the same as the stacksize used for non-threaded applications. If you write your own MPI reduce functions to use with nonblocking collective communications or a SIGIO handler that will be executed on one of the
| library service threads, you are limited to a stacksize of 96KB by default. To increase your thread stacksize, use the environment variable
| MP_THREAD_STACKSIZE. For more information about the default and your ability
| to change the default, see the manpage for AIX_PTHREAD_SET_STACKSIZE.

Forks Are Limited

If a task forks, only the thread that forked exists in the child task. Therefore, the message passing library will not operate properly. Also, if the forked child does not exec another program, it should be aware that an atexit routine has been registered for the parent which is also inherited by the child. In most cases, the atexit routine requests that POE terminate the task (parent). A forked child should terminate with an _exit(0) system call to prevent the atexit routine from being called. Also, if the forked parent terminates before the child, the child task will not be cleaned up by POE.

A forked child MUST NOT call the message passing library.

Standard I/O Requires Special Attention

When your program runs on the remote nodes, it has no controlling terminal. STDIN and STDOUT, STDERR are always piped.

If your threaded MPI program processes STDIN from a large file on the home node, you must do one of the following:

- Invoke MPI_Init() before performing any STDIN processing, or
- Ensure that all STDIN has been processed (EOF) before invoking MPI_Init().

This also includes programs which may not explicitly use MPI.

If STDIN is piped (or redirected) to the poe binary (via ordinary pipes) and your application is linked with the threaded library, then handle STDIN in the following way:

- If all of STDIN is read by your program before MPI_Init is called, set the environment variable MP_HOLD_STDIN=NO.
- If none of STDIN is read before MPI_Init is called, set the environment variable MP_HOLD_STDIN=YES.
- If STDIN is less than approximately 4000 bytes in length, set MP_HOLD_STDIN=NO.
- If none of the above applies, it may not be possible to run your program correctly, and you will have to devise some other mechanism for providing data to your program.

Thread-Safe Libraries

AIX provides thread-safe versions of some libraries, such as libc_r.a. However, not all libraries have a thread-safe version. It is your responsibility to determine whether the libraries you use can be safely called by more than one thread.

Program And Thread Termination

MPI_FINALIZE terminates the MPI service threads but does not affect user-created threads. Use pthread_exit to terminate any user-created threads, and exit(m) to terminate the main program (initial thread). The value of m is used to set POE's exit status as explained on "Exit Status" on page 413.

Other Thread-Specific Considerations

Order Requirement For System Includes

For threaded programs, AIX requires that the system include <pthread.h> must be first with <stdio.h> or other system includes following it. <pthread.h> defines some conditional compile variables that modify the code generation of subsequent includes, particularly <stdio.h>. Please note that <pthread.h> is not required unless your file uses thread-related calls or data.

MPI_INIT

Call MPI_INIT once per task not once per thread. MPI_INIT does not have to be called on the main thread but MPI_INIT and MPI_FINALIZE must be called on the same thread.

MPI calls on other threads must adhere to the MPI standard in regard to the following:

- A thread cannot make MPI calls until MPI_INIT has been called.
- A thread cannot make MPI calls after MPI_FINALIZE has been called.
- Unless there is a specific thread protocol programmed, you cannot rely on any specific order or speed of thread processing.

Collective Communications

Collective communications must meet the MPI standard requirement that all participating tasks execute collective communications on any given communicator in the same order. If collective communications calls are made on multiple threads, it is your responsibility to ensure the proper sequencing or to use distinct communicators.

Support for M:N Threads

By default, user threads are created with process contention scope, and M user threads are mapped to N kernel threads. The values of the ratio M:N and the default contention scope are settable by AIX environment variables. The service threads created by MPI, POE, and LAPI have system contention scope, that is, they are mapped 1:1 to kernel threads.

For PSSP 2.3 and 2.4, you must create system contention scope threads. For PSSP 3.1, you can create process contention scope threads, but any such thread will be converted to a system contention scope thread when it makes its first MPI call.

Fortran Considerations

The information in this subsection provides you with some specific programming considerations for when you are using POE and the Fortran compiler.

Fortran 90 and MPI

Incompatibilities exist between Fortran 90 and MPI which may effect the ability to use such programs. Refer to the information in

```
/usr/lpp/ppe.poe/samples/mpif90/README.mpif90
```

for further details. PE, Version 2, Release 2 provided the header file mpif90.h for use with Fortran 90. The file is still available in PE, Version 2, Release 4 , but should not be used by new code. The mpif.h header file is formatted to work with either **mpxlf90** or **mpxlf** compilation.

Fortran and Threads

| Version 5 of the AIX XLF Fortran compiler supports threads.

| Version 4.1 of the AIX XLF Fortran compiler is not thread-safe. However, XLF Version 4.1.0.1 provides a partial thread-support XLF runtime library. It supports multi-threaded applications that have one Fortran thread. Be sure you thoroughly test such use.

The partial thread-support library is **libxlf90_t.a** and is installed as **/usr/lib/libxlf90_t.a**. When you use the **mpxlf_r** command, this library is included automatically.

Restrictions

When you use **libxlf90_t.a** the following restrictions apply. Therefore, only one Fortran thread in a multi-threaded application may use the library.

- Routines in the library are not thread-reentrant.
- Use of routines in the math library (**libm.a**) by more than one thread may produce unpredictable results.

Appendix H. Using Signals and the IBM PE Programs

This section applies to the signal-handling version of the Message Passing library. Any AIX function that is interruptible by a signal may not behave as expected because the message passing subsystem uses timer signals to manage message traffic. For example, the user's program does not sleep for the full time but returns quickly with an error code of EINTR. This indicates the sleep was interrupted by a signal. This happens for select system call as well.

The following are some sample programs to replace sleep and select.

Sample Replacement Sleep Program

The following sample replacement program for sleep guarantees to sleep for the specified interval, even if interrupted.

Sleep Example

```
#include <errno.h>
#include <sys/time.h>
int SLEEP(int amount)
{
 struct timestruc_t Requested, Remaining;
 double famount = amount;
 int rc;
 while (famount > 0.0) {
  Requested.tv_sec = (int) famount;
  Requested.tv_nsec =
   (int) ((famount - Requested.tv_sec)*1000000000.);
  rc = nsleep ( &Requested, &Remaining );
  if ((rc == -1) && (errno == EINTR)) {
   /* Sleep interrupted.  Resume it */
   famount = Remaining.tv_sec + Remaining.tv_nsec /
       1000000000.;
   continue;
  }
  else /* Completed sleep.  Set return to zero */
  {
    return (0);
  }
 }    /* end of while */

 /* famount = 0; exit */
 return (0);
}
```

Sample Replacement Select Program

The following is a sample replacement program for select. SELECT restores the status of the file descriptor bit masks and handles the remaining time after an interrupt.

Select Example

```
#include <stdio.h>
#include <sys/select.h>
#include <sys/types.h>
#include <sys/time.h>
#include <errno.h>

int SELECT(int maxfds, fd_set *reads, fd_set *writes, fd_set *errors,
  struct timeval *timeout)
{
 struct timestruc_t Timer1, Timer2;
 struct timeval timetogo;
 static fd_set readcopy;
 static fd_set writecopy;
 static fd_set errcopy;
 int rc;
 double worktime;
 double remaining;

 /* If we get interrupted, will need to restore select bits */
 if (reads) bcopy(reads,&readcopy,sizeof(fd_set));
 if (writes) bcopy(writes,&writecopy,sizeof(fd_set));
 if (errors) bcopy(errors,&errcopy,sizeof(fd_set));

 /* two cases: if timeout specifies a time structure, we
  need to worry about timeouts.  Otherwise, we can
  ignore it */

 if (timeout == NULL) {
  while (TRUE) {
   rc = select(maxfds,reads,writes,errors,NULL);
   if ((rc == -1) && (errno == EINTR)) { /* interrupted */
    if (reads) bcopy(&readcopy,reads,sizeof(fd_set));
    if (writes) bcopy(&writecopy,writes,sizeof(fd_set));
    if (errors) bcopy(&errcopy,errors,sizeof(fd_set));
    continue;
   }
   else return(rc);
  }
 }
 else  { /* timeout is not null */
  timetogo.tv_sec = timeout->tv_sec;
  timetogo.tv_usec = timeout->tv_usec;
  remaining = timetogo.tv_sec + timetogo.tv_usec/1000000.;
/*
  fprintf(stderr,"remaining time = %f\n",remaining);
  fflush(stderr);
*/
  gettimer(TIMEOFDAY, &Timer2);
  while (TRUE) {
   Timer1.tv_sec = Timer2.tv_sec;
   Timer1.tv_nsec = Timer2.tv_nsec;
   rc = select(maxfds,reads,writes,errors,&timetogo);
   if ((rc == -1) && (errno == EINTR)) { /* interrupted */
    gettimer(TIMEOFDAY, &Timer2);
    /* compute amount remaining */
    worktime = (Timer2.tv_sec - Timer1.tv_sec) +
     (Timer2.tv_nsec - Timer1.tv_nsec)/1000000000.;
    remaining = remaining - worktime;
    timetogo.tv_sec = (int) remaining;
    timetogo.tv_usec = (int) ((remaining - timetogo.tv_sec)*
        1000000.);
    /* restore the select bits */
    if (reads) bcopy(&readcopy,reads,sizeof(fd_set));
    if (writes) bcopy(&writecopy,writes,sizeof(fd_set));
    if (errors) bcopy(&errcopy,errors,sizeof(fd_set));
    continue;
   }
   else return(rc);
  }
 }
}
```

Appendix I. Predefined Datatypes

The following is a list of the various predefined MPI datatypes by category that you can use with MPI .

Special Purpose

Datatype	Description
MPI_LB	Explicit lower bound marker
MPI_UB	Explicit upper bound marker
MPI_BYTE	Untyped byte data
MPI_PACKED	Packed data (byte)

For C Language Bindings

Datatype	Description
MPI_CHAR	8-bit character
MPI_UNSIGNED_CHAR	8-bit unsigned character
MPI_SIGNED_CHAR	8-bit signed character
MPI_SHORT	16-bit integer
MPI_INT	32-bit integer
MPI_LONG	32-bit integer
MPI_UNSIGNED_SHORT	16-bit unsigned integer
MPI_UNSIGNED	32-bit unsigned integer
MPI_UNSIGNED_LONG	32-bit unsigned integer
MPI_FLOAT	32-bit floating point
MPI_DOUBLE	64-bit floating point
MPI_LONG_DOUBLE	64-bit floating point
MPI_UNSIGNED_LONG_LONG	64-bit unsigned integer
MPI_LONG_LONG_INT	64-bit integer
MPI_WCHAR	Wide (16-bit) unsigned character

For FORTRAN Language Bindings

Datatype	Description
MPI_INTEGER1	8 bit integer
MPI_INTEGER2	16 bit integer
MPI_INTEGER4	32 bit integer
MPI_INTEGER	32 bit integer
MPI_INTEGER8	64 bit integer
MPI_REAL4	32 bit floating point
MPI_REAL	32 bit floating point
MPI_REAL8	64 bit floating point
MPI_DOUBLE_PRECISION	64 bit floating point
MPI_REAL16	128 bit floating point
MPI_COMPLEX8	32 bit float real, 32 bit float imag.
MPI_COMPLEX	32 bit float real, 32 bit float imag.
MPI_COMPLEX16	64 bit float real, 64 bit float imag.
MPI_DOUBLE_COMPLEX	64 bit float real, 64 bit float imag.
MPI_COMPLEX32	128 bit float real, 128 bit float imag.
MPI_LOGICAL1	8 bit logical
MPI_LOGICAL2	16 bit logical
MPI_LOGICAL4	32 bit logical
MPI_LOGICAL	32 bit logical
MPI_LOGICAL8	64 bit logical
MPI_CHARACTER	8 bit character

For Reduction Functions (C Reduction Types)

Datatype	Description
MPI_FLOAT_INT	{MPI_FLOAT, MPI_INT}
MPI_DOUBLE_INT	{MPI_DOUBLE, MPI_INT}
MPI_LONG_INT	{MPI_LONG, MPI_INT}
MPI_2INT	{MPI_INT, MPI_INT}
MPI_SHORT_INT	{MPI_SHORT, MPI_INT}
MPI_LONG_DOUBLE_INT	{MPI_LONG_DOUBLE, MPI_INT}

For Reduction Functions (FORTRAN Reduction Types)

Datatype	Description
MPI_2REAL	{MPI_REAL, MPI_REAL}
MPI_2DOUBLE_PRECISION	{MPI_DOUBLE_PRECISION, MPI_DOUBLE_PRECISION}
MPI_2INTEGER	{MPI_INTEGER, MPI_INTEGER}
MPI_2COMPLEX	{MPI_COMPLEX, MPI_COMPLEX}

Appendix J. MPI Environment Variables Quick Reference

Table 18 summarizes the environment variables and flags for the Message Passing Interface. These environment variables and flags allow you to change message and memory sizes, as well as other message passing information.

Table 18 (Page 1 of 2). POE Environment Variables and Command-Line Flags for MPI

Environment Variable Command-Line Flag	**Set:**	**Possible Values:**	**Default:**
MP_BUFFER_MEM -buffer_mem	To change the maximum size of memory used by the communication subsystem to buffer early arrivals.	An integer less than or equal to 64MB *nnnn* *nnnn*K *nn*M	2800000 bytes (IP) 64MB (US)
MP_CLOCK_SOURCE -clock_source	To use the SP switch clock as a time source. See "Using the SP Switch Clock as a Time Source" on page 420.	AIX SWITCH	
MP_CSS_INTERRUPT -css_interrupt	Whether or not arriving packets generate interrupts. This may provide better performance for certain applications. Setting this explicitly will suppress the MPI-directed switching of interrupt mode, leaving the user in control for the rest of the run. See MPI_FILE_OPEN.	yes no	no

MP_EAGER_LIMIT
 	
-eager_limit
	To change the threshold value for message size, above which rendezvous protocol is used.
	
	To ensure that at least 32 messages can be outstanding between any 2 tasks, **MP_EAGER_LIMIT** will be adjusted based on the number of tasks according to the following table (and: when MP_USE_FLOW_CONTROL=YES and **MP_EAGER_LIMIT** and **MP_BUFFER_MEM** have not been set by the user):
	
	`Number of`
	`Tasks     MP_EAGER_LIMIT`
	`------------------------`
	`  1 to  16          4096`
	` 17 to  32          2048`
	` 33 to  64          1024`
	` 65 to 128           512`
	`129 to 256           256`
	`257 to the maximum   128`
	`number of tasks`
	`supported by the`
	`implementation`
		An integer less than or equal to 65536
		
		*nnn*K
			4KB

Table 18 (Page 2 of 2). POE Environment Variables and Command-Line Flags for MPI

Environment Variable Command-Line Flag	**Set:**	**Possible Values:**	**Default:**
MP_INTRDELAY -intrdelay	To tune the delay parameter without having to recompile existing applications.	An integer greater than 0	35 μ (TB2) 1 μ (TB3)
MP_MAX_TYPEDEPTH -max_typedepth	To change the maximum depth of message derived data types.	An integer greater than or equal to 1	5
MP_SINGLE_THREAD -single_thread	To avoid lock overheads in a program that is known to be single-threaded. Note that MPI-IO cannot be used if this variable is set to **yes**. Results are undefined if this variable is set to **yes** with multiple message threads in use.	no yes	no
MP_THREAD_STACKSIZE -thread_stacksize	To specify the additional stacksize allocated for user programs executing on an MPI service thread. If you allocate insufficient space, the program may encounter a SIGSEGV exception.	*nnnnn* *nnn*K *nn*M (where K=1024 bytes and M=1024*1024 bytes)	None
MP_TIMEOUT (no associated command-line flag)	To change the length of time the communication subsystem will wait for a connection to be established during message passing initialization.	An integer greater than 0	150 seconds
MP_USE_FLOW_CONTROL -use_flow_control	To limit the maximum number of outstanding messages posted by a sender.	yes no	no
MP_WAIT_MODE -wait_mode	To specify how a thread or task behaves when it discovers it is blocked, waiting for a message to arrive.	poll yield sleep	poll (US) yield (IP)

Glossary of Terms and Abbreviations

This glossary includes terms and definitions from:

- The *Dictionary of Computing*, New York: McGraw-Hill, 1994.
- The *American National Standard Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies can be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018. Definitions are identified by the symbol (A) after the definition.
- The *ANSI/EIA Standard - 440A: Fiber Optic Terminology*, copyright 1989 by the Electronics Industries Association (EIA). Copies can be purchased from the Electronic Industries Association, 2001 Pennsylvania Avenue N.W., Washington, D.C. 20006. Definitions are identified by the symbol (E) after the definition.
- The *Information Technology Vocabulary* developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

This section contains some of the terms that are commonly used in the Parallel Environment books and in this book in particular.

IBM is grateful to the American National Standards Institute (ANSI) for permission to reprint its definitions from the American National Standard *Vocabulary for Information Processing* (Copyright 1970 by American National Standards Institute, Incorporated), which was prepared by Subcommittee X3K5 on Terminology and Glossary of the American National Standards Committee X3. ANSI definitions are preceded by an asterisk (*).

Other definitions in this glossary are taken from *IBM Vocabulary for Data Processing, Telecommunications, and Office Systems* (GC20-1699).

A

address. A value, possibly a character or group of characters that identifies a register, a device, a particular part of storage, or some other data source or destination.

AIX. Abbreviation for Advanced Interactive Executive, IBM's licensed version of the UNIX operating system. AIX is particularly suited to support technical computing applications, including high function graphics and floating point computations.

AIXwindows Environment/6000. A graphical user interface (GUI) for the RS/6000. It has the following components:

- A graphical user interface and toolkit based on OSF/Motif
- Enhanced X-Windows, an enhanced version of the MIT X Window System
- Graphics Library (GL), a graphical interface library for the applications programmer which is compatible with Silicon Graphics' GL interface.

API. Application Programming Interface.

application. The use to which a data processing system is put; for example, topayroll application, an airline reservation application.

argument. A parameter passed between a calling program and a called program or subprogram.

attribute. A named property of an entity.

B

| **bandwidth**. The difference, expressed in hertz,
| between the highest and the lowest frequencies of a
| range of frequencies. For example, analog transmission
| by recognizable voice telephone requires a bandwidth
| of about 3000 hertz (3 kHz). The bandwidth of an
| optical link designates the information-carrying capacity
| of the link and is related to the maximum bit rate that a
| fiber link can support.

blocking operation. An operation which does not complete until the operation either succeeds or fails. For example, a blocking receive will not return until a message is received or until the channel is closed and no further messages can be received.

breakpoint. A place in a program, specified by a command or a condition, where the system halts execution and gives control to the workstation user or to a specified program.

broadcast operation. A communication operation in which one processor sends (or broadcasts) a message to all other processors.

buffer. A portion of storage used to hold input or output data temporarily.

C

C. A general purpose programming language. It was formalized by ANSI standards committee for the C language in 1984 and by Uniforum in 1983.

C++. A general purpose programming language, based on C, which includes extensions that support an object-oriented programming paradigm. Extensions include:

- strong typing
- data abstraction and encapsulation
- polymorphism through function overloading and templates
- class inheritance.

call arc. The representation of a call between two functions within the Xprofiler function call tree. It appears as a solid line between the two functions. The arrowhead indicates the direction of the call; the function it points to is the one that receives the call. The function making the call is known as the *caller*, while the function receiving the call is known as the *callee*.

chaotic relaxation. An iterative relaxation method which uses a combination of the Gauss-Seidel and Jacobi-Seidel methods. The array of discrete values is divided into sub-regions which can be operated on in parallel. The sub-region boundaries are calculated using Jacobi-Seidel, whereas the sub-region interiors are calculated using Gauss-Seidel. See also *Gauss-Seidel*.

client. A function that requests services from a server, and makes them available to the user.

cluster. A group of processors interconnected through a high speed network that can be used for high-performance computing. It typically provides excellent price/performance.

collective communication. A communication operation which involves more than two processes or tasks. Broadcasts, reductions, and the MPI_Allreduce subroutine are all examples of collective communication operations. All tasks in a communicator must participate.

command alias. When using the PE command line debugger, pdbx, you can create abbreviations for existing commands using the **pdbx alias** command. These abbreviations are know as *command aliases*.

Communication Subsystem (CSS). A component of the Parallel System Support Programs that provides
| software support for the SP Switch. CSS provides two protocols: IP (Internet Protocol) for LAN-based communication and US (user space) as a message passing interface that is optimized for performance over the switch. See also *Internet Protocol* and *User Space.*

communicator. An MPI object that describes the communication context and an associated group of processes.

compile. To translate a source program into an executable program.

condition. One of a set of specified values that a data item can assume.

control workstation. A workstation attached to the IBM RS/6000 SP SP that serves as a single point of control allowing the administrator or operator to monitor and manage the system using Parallel System Support Programs.

core dump. A process by which the current state of a program is preserved in a file. Core dumps are usually associated with programs that have encountered an unexpected, system-detected fault, such as a Segmentation Fault, or severe user error. The current program state is needed for the programmer to diagnose and correct the problem.

core file. A file which preserves the state of a program, usually just before a program is terminated for an unexpected error. See also *core dump*.

current context. When using either of the PE parallel debuggers, control of the parallel program and the display of its data can be limited to a subset of the tasks that belong to that program. This subset of tasks is called the *current context*. You can set the current context to be a single task, multiple tasks, or all the tasks in the program.

D

data decomposition. A method of breaking up (or decomposing) a program into smaller parts to exploit parallelism. One divides the program by dividing the data (usually arrays) into smaller parts and operating on each part independently.

data parallelism. Refers to situations where parallel tasks perform the same computation on different sets of data.

dbx. A symbolic command line debugger that is often provided with UNIX systems. The PE command line debugger, **pdbx**, is based on the **dbx** debugger.

debugger. A debugger provides an environment in which you can manually control the execution of a program. It also provides the ability to display the program's data and operation.

distributed shell (dsh). An Parallel System Support Programs command that lets you issue commands to a group of hosts in parallel. See *IBM Parallel System Support Programs for AIX: Command and Technical Reference* for details.

domain name. The hierarchical identification of a host system (in a network), consisting of human-readable labels, separated by decimals.

E

environment variable. 1. A variable that describes the operating environment of the process. Common environment variables describe the home directory, command search path, and the current time zone. 2. A variable that is included in the current software environment and is therefore available to any called program that requests it.

event. An occurrence of significance to a task; for example, the completion of an asynchronous operation such as an input/output operation.

Ethernet. Ethernet is the standard hardware for TCP/IP LANs in the UNIX marketplace. It is a 10 megabit per second baseband type network that uses the contention based CSMA/CD (collision detect) media access method.

executable. A program that has been link-edited and therefore can be run in a processor.

execution. To perform the actions specified by a program or a portion of a program.

expression. In programming languages, a language construct for computing a value from one or more operands.

F

fairness. A policy in which tasks, threads, or processes must be allowed eventual access to a resource for which they are competing. For example, if multiple threads are simultaneously seeking a lock, then no set of circumstances can cause any thread to wait indefinitely for access to the lock.

FDDI. Fiber distributed data interface (100 Mbit/s fiber optic LAN).

file system. In the AIX operating system, the collection of files and file management structures on a physical or logical mass storage device, such as a diskette or minidisk.

fileset. 1) An individually installable option or update. Options provide specific function while updates correct an error in, or enhance, a previously installed product. 2) One or more separately installable, logically grouped units in an installation package. See also *Licensed Program Product* and *package*.

foreign host. See *remote host*.

Fortran. One of the oldest of the modern programming languages, and the most popular language for scientific and engineering computations. It's name is a contraction of *FORmula TRANslation*. The two most common Fortran versions are Fortran 77, originally standardized in 1978, and Fortran 90. Fortran 77 is a proper subset of Fortran 90.

function call tree. A graphical representation of all the functions and calls within an application, which appears in the Xprofiler main window. The functions are represented by green, solid-filled rectangles called function boxes. The size and shape of each function box indicates its CPU usage. Calls between functions are represented by blue arrows, called call arcs, drawn between the function boxes. See also *call arcs*.

function cycle. A chain of calls in which the first caller is also the last to be called. A function that calls itself recursively is not considered a function cycle.

functional decomposition. A method of dividing the work in a program to exploit parallelism. One divides the program into independent pieces of functionality which are distributed to independent processors. This is in contrast to data decomposition which distributes the same work over different data to independent processors.

functional parallelism. Refers to situations where parallel tasks specialize in particular work.

G

Gauss-Seidel. An iterative relaxation method for solving Laplace's equation. It calculates the general solution by finding particular solutions to a set of discrete points distributed throughout the area in question. The values of the individual points are obtained by averaging the values of nearby points. Gauss-Seidel differs from Jacobi-Seidel in that for the i+1st iteration Jacobi-Seidel uses only values calculated in the ith iteration. Gauss-Seidel uses a mixture of values calculated in the ith and i+1st iterations.

global max. The maximum value across all processors for a given variable. It is global in the sense that it is global to the available processors.

global variable. A variable defined in one portion of a computer program and used in at least one other portion of the computer program.

gprof. A UNIX command that produces an execution profile of C, Pascal, Fortran, or COBOL programs. The execution profile is in a textual and tabular format. It is useful for identifying which routines use the most CPU time. See the man page on **gprof**.

GUI (Graphical User Interface). A type of computer interface consisting of a visual metaphor of a real-world scene, often of a desktop. Within that scene are icons, representing actual objects, that the user can access and manipulate with a pointing device.

H

SP Switch. The high-performance message passing network, of the IBM RS/6000 SP(SP) machine, that connects all processor nodes.

HIPPI. High performance parallel interface.

hook. **hook** is a **pdbx** command that allows you to re-establish control over all task(s) in the current context that were previously unhooked with this command.

home node. The node from which an application developer compiles and runs his program. The home node can be any workstation on the LAN.

host. A computer connected to a network, and providing an access method to that network. A host provides end-user services.

host list file. A file that contains a list of host names, and possibly other information, that was defined by the application which reads it.

host name. The name used to uniquely identify any computer on a network.

hot spot. A memory location or synchronization resource for which multiple processors compete excessively. This competition can cause a disproportionately large performance degradation when one processor that seeks the resource blocks, preventing many other processors from having it, thereby forcing them to become idle.

I

IBM Parallel Environment for AIX. A program product that provides an execution and development environment for parallel FORTRAN, C, or C++ programs. It also includes tools for debugging, profiling, and tuning parallel programs.

installation image. A file or collection of files that are required in order to install a software product on a RS/6000 workstation or on SP system nodes. These files are in a form that allows them to be installed or removed with the AIX **installp** command. See also *fileset*, *Licensed Program Product*, and *package*.

Internet. The collection of worldwide networks and gateways which function as a single, cooperative virtual network.

Internet Protocol (IP). 1) The TCP/IP protocol that provides packet delivery between the hardware and
| user processes. 2) The SP Switch library, provided with the Parallel System Support Programs, that follows the IP protocol of TCP/IP.

IP. See *Internet Protocol*.

J

Jacobi-Seidel. See *Gauss-Seidel*.

| **job management system**.

| The software you use to manage the jobs across your
| system, based on the availability and state of system
| resources.

K

Kerberos. A publicly available security and authentication product that works with the Parallel System Support Programs software to authenticate the execution of remote commands.

kernel. The core portion of the UNIX operating system which controls the resources of the CPU and allocates them to the users. The kernel is memory-resident, is said to run in *kernel mode* (in other words, at higher execution priority level than *user mode*) and is protected from user tampering by the hardware.

L

Laplace's equation. A homogeneous partial differential equation used to describe heat transfer, electric fields, and many other applications.

The dimension-free version of Laplace's equation is:

$$\nabla^2 u = 0$$

The two-dimensional version of Laplace's equation may be written as:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

latency. The time interval between the instant at which an instruction control unit initiates a call for data transmission, and the instant at which the actual transfer of data (or receipt of data at the remote end) begins. Latency is related to the hardware characteristics of the system and to the different layers of software that are involved in initiating the task of packing and transmitting the data.

Licensed Program Product (LPP). A collection of software packages, sold as a product, that customers pay for to license. It can consist of packages and filesets a customer would install. These packages and filesets bear a copyright and are offered under the terms and conditions of a licensing agreement. See also *fileset* and *package*.

| **LoadLeveler**. A job management system that works
| with POE to allow users to run jobs and match
| processing needs with system resources, in order to
| better utilize the system.

local variable. A variable that is defined and used only in one specified portion of a computer program.

loop unrolling. A program transformation which makes multiple copies of the body of a loop, placing the copies also within the body of the loop. The loop trip count and index are adjusted appropriately so the new loop computes the same values as the original. This transformation makes it possible for a compiler to take additional advantage of instruction pipelining, data cache effects, and software pipelining.

See also *optimization*.

M

menu. A list of options displayed to the user by a data processing system, from which the user can select an action to be initiated.

message catalog. A file created using the AIX Message Facility from a message source file that contains application error and other messages, which can later be translated into other languages without having to recompile the application source code.

message passing. Refers to the process by which parallel tasks explicitly exchange program data.

MIMD (Multiple Instruction Multiple Data). A parallel programming model in which different processors perform different instructions on different sets of data.

MPMD (Multiple Program Multiple Data). A parallel programming model in which different, but related, programs are run on different sets of data.

MPI. Message Passing Interface; a standardized API for implementing the message passing model.

N

network. An interconnected group of nodes, lines, and terminals. A network provides the ability to transmit data to and receive data from other systems and users.

node. (1) In a network, the point where one or more functional units interconnect transmission lines. A computer location defined in a network. (2) In terms of the IBM RS/6000 SP, a single location or workstation in a network. An SP node is a physical entity (a processor).

node ID. A string of unique characters that identifies the node on a network.

nonblocking operation. An operation, such as sending or receiving a message, which returns immediately whether or not the operation was completed. For example, a nonblocking receive will not wait until a message is sent, but a blocking receive will wait. A nonblocking receive will return a status value that indicates whether or not a message was received.

O

object code. The result of translating a computer program to a relocatable, low-level form. Object code contains machine instructions, but symbol names (such as array, scalar, and procedure names), are not yet given a location in memory.

optimization. A not strictly accurate but widely used term for program performance improvement, especially for performance improvement done by a compiler or other program translation software. An optimizing compiler is one that performs extensive code transformations in order to obtain an executable that runs faster but gives the same answer as the original. Such code transformations, however, can make code debugging and performance analysis very difficult because complex code transformations obscure the correspondence between compiled and original source code.

option flag. Arguments or any other additional information that a user specifies with a program name. Also referred to as *parameters* or *command line options*.

P

package. A number of filesets that have been collected into a single installable image of program products, or LPPs. Multiple filesets can be bundled together for installing groups of software together. See also *fileset* and *Licensed Program Product*.

parallelism. The degree to which parts of a program may be concurrently executed.

parallelize. To convert a serial program for parallel execution.

Parallel Operating Environment (POE). An execution environment that smooths the differences between serial and parallel execution. It lets you submit and manage parallel jobs. It is abbreviated and commonly known as POE.

parameter. * (1) In Fortran, a symbol that is given a constant value for a specified application. (2) An item in a menu for which the operator specifies a value or for which the system provides a value when the menu is interpreted. (3) A name in a procedure that is used to refer to an argument that is passed to the procedure. (4) A particular piece of information that a system or application program needs to process a request.

partition. (1) A fixed-size division of storage. (2) In terms of the IBM RS/6000 SP, a logical definition of nodes to be viewed as one system or domain. System partitioning is a method of organizing the SP into groups of nodes for testing or running different levels of software of product environments.

Partition Manager. The component of the Parallel Operating Environment (POE) that allocates nodes, sets up the execution environment for remote tasks, and manages distribution or collection of standard input (STDIN), standard output (STDOUT), and standard error (STDERR).

pdbx. **pdbx** is the parallel, symbolic command line debugging facility of PE. **pdbx** is based on the **dbx** debugger and has a similar interface.

PE. The IBM Parallel Environment for AIX program product.

performance monitor. A utility which displays how effectively a system is being used by programs.

POE. See Parallel Operating Environment.

pool. Groups of nodes on an SP that are known to the Resource Manager, and are identified by a number.

point-to-point communication. A communication operation which involves exactly two processes or tasks. One process initiates the communication through a *send* operation. The partner process issues a *receive* operation to accept the data being sent.

procedure. (1) In a programming language, a block, with or without formal parameters, whose execution is invoked by means of a procedure call. (2) A set of related control statements that cause one or more programs to be performed.

process. A program or command that is actually running the computer. It consists of a loaded version of the executable file, its data, its stack, and its kernel data structures that represent the process's state within a multitasking environment. The executable file contains the machine instructions (and any calls to shared objects) that will be executed by the hardware. A process can contain multiple threads of execution.

The process is created via a **fork**() system call and ends using an **exit**() system call. Between **fork** and **exit**, the process is known to the system by a unique process identifier (pid).

Each process has its own virtual memory space and cannot access another process's memory directly. Communication methods across processes include pipes, sockets, shared memory, and message passing.

prof. A utility which produces an execution profile of an application or program. It is useful to identifying which routines use the most CPU time. See the man page for **prof**.

profiling. The act of determining how much CPU time is used by each function or subroutine in a program. The histogram or table produced is called the execution profile.

Program Marker Array. An X-Windows run time monitor tool provided with Parallel Operating Environment, used to provide immediate visual feedback on a program's execution.

pthread. A thread that conforms to the POSIX Threads Programming Model.

R

reduction operation. An operation, usually mathematical, which reduces a collection of data by one or more dimensions. For example, the arithmetic SUM operation is a reduction operation which reduces an array to a scalar value. Other reduction operations include MAXVAL and MINVAL.

remote host. Any host on a network except the one at which a particular operator is working.

remote shell (rsh). A command supplied with both AIX and the Parallel System Support Programs that lets you issue commands on a remote host.

Report. In Xprofiler, a tabular listing of performance data that is derived from the gmon.out files of an application. There are five types of reports that are generated by Xprofiler, and each one presents different statistical information for an application.

| **Resource Manager**. A server that runs on one of the
| nodes of a IBM RS/6000 SP (SP) machine. It prevents
| parallel jobs from interfering with each other, and
| reports job-related node information.

RISC. Reduced Instruction Set Computing (RISC), the technology for today's high-performance personal computers and workstations, was invented in 1975.

S

shell script. A sequence of commands that are to be executed by a shell interpreter such as C shell, korn shell, or Bourne shell. Script commands are stored in a file in the same form as if they were typed at a terminal.

segmentation fault. A system-detected error, usually caused by referencing an invalid memory address.

server. A functional unit that provides shared services to workstations over a network; for example, a file server, a print server, a mail server.

signal handling. A type of communication that is used by message passing libraries. Signal handling involves using AIX signals as an asynchronous way to move data in and out of message buffers.

source line. A line of source code.

source code. The input to a compiler or assembler, written in a source language. Contrast with object code.

SP. IBM RS/6000 SP; a scalable system from two to 128 processor nodes, arranged in various physical configurations, that provides a high-powered computing environment.

SPMD (Single Program Multiple Data). A parallel programming model in which different processors execute the same program on different sets of data.

standard input (STDIN). In the AIX operating system, the primary source of data entered into a command. Standard input comes from the keyboard unless redirection or piping is used, in which case standard input can be from a file or the output from another command.

standard output (STDOUT). In the AIX operating system, the primary destination of data produced by a command. Standard output goes to the display unless redirection or piping is used, in which case standard output can go to a file or to another command.

stencil. A pattern of memory references used for averaging. A 4-point stencil in two dimensions for a given array cell, x(i,j), uses the four adjacent cells, x(i-1,j), x(i+1,j), x(i,j-1), and x(i,j+1).

subroutine. (1) A sequence of instructions whose execution is invoked by a call. (2) A sequenced set of instructions or statements that may be used in one or more computer programs and at one or more points in a computer program. (3) A group of instructions that can be part of another routine or can be called by another program or routine.

synchronization. The action of forcing certain points in the execution sequences of two or more asynchronous procedures to coincide in time.

system administrator. (1) The person at a computer installation who designs, controls, and manages the use of the computer system. (2) The person who is responsible for setting up, modifying, and maintaining the Parallel Environment.

System Data Repository. A component of the Parallel System Support Programs software that provides configuration management for the SP system. It manages the storage and retrieval of system data across the control workstation, file servers, and nodes.

System Status Array. An X-Windows run time monitor tool, provided with the Parallel Operating Environment, that lets you quickly survey the utilization of processor nodes.

T

task. A unit of computation analogous to an AIX process.

thread. A single, separately dispatchable, unit of execution. There may be one or more threads in a process, and each thread is executed by the operating system concurrently.

tracing. In PE, the collection of data for the Visualization Tool (VT). The program is *traced* by collecting information about the execution of the program in trace records. These records are then accumulated into a trace file which a user visualizes with VT.

tracepoint. Tracepoints are places in the program that, when reached during execution, cause the debugger to print information about the state of the program.

trace record. In PE, a collection of information about a specific event that occurred during the execution of your program. For example, a trace record is created for each send and receive operation that occurs in your program (this is optional and may not be appropriate). These records are then accumulated into a trace file which allows the Visualization Tool to visually display the communications patterns from the program.

U

unrolling loops. See *loop unrolling*.

US. See *user space*.

user. (1) A person who requires the services of a computing system. (2) Any person or any thing that may issue or receive commands and message to or from the information processing system.

user space (US). A version of the message passing
| library that is optimized for direct access to the SP
| Switch , that maximizes the performance capabilities of the SP hardware.

utility program. A computer program in general support of computer processes; for example, a diagnostic program, a trace program, a sort program.

utility routine. A routine in general support of the processes of a computer; for example, an input routine.

V

variable. (1) In programming languages, a named object that may take different values, one at a time. The values of a variable are usually restricted to one data type. (2) A quantity that can assume any of a given set of values. (3) A name used to represent a data item whose value can be changed while the program is running. (4) A name used to represent data whose value can be changed, while the program is running, by referring to the name of the variable.

view. (1) In an information resource directory, the combination of a variation name and revision number that is used as a component of an access name or of a descriptive name.

Visualization Tool. The PE Visualization Tool. This tool uses information that is captured as your parallel program executes, and presents a graphical display of the program execution. For more information, see *IBM Parallel Environment for AIX: Operation and Use, Volume 2*.

VT. See *Visualization Tool*.

X

X Window System. The UNIX industry's graphics windowing standard that provides simultaneous views of several executing programs or processes on high resolution graphics displays.

xpdbx. This is the former name of the PE graphical interface debugging facility, which is now called **pedb**.

Xprofiler. An AIX tool that is used to analyze the performance of both serial and parallel applications, via a graphical user interface. Xprofiler provides quick access to the profiled data, so that the functions that are the most CPU-intensive can be easily identified.

Index

Communicating Your Comments to IBM

IBM Parallel Environment for AIX
MPI Programming and Subroutine
Reference
Version 2 Release 4

Publication No. GC23-3894-03

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM. Whichever method you choose, make sure you send your name, address, and telephone number if you would like a reply.

Feel free to comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. However, the comments you send should pertain to only the information in this manual and the way in which the information is presented. To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

If you are mailing a reader's comment form (RCF) from a country other than the United States, you can give the RCF to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:
  - FAX: (International Access Code)+1+914+432-9405
- If you prefer to send comments electronically, use this network ID:
  - IBM Mail Exchange: USIB6TC9 at IBMMAIL
  - Internet e-mail: mhvrcfs@us.ibm.com
  - World Wide Web: http://www.s390.ibm.com/os390

Make sure to include the following in your note:

- Title and publication number of this book
- Page number or topic to which your comment applies

Optionally, if you include your telephone number, we will be able to respond to your comments by phone.

Reader's Comments — We'd Like to Hear from You

IBM Parallel Environment for AIX
MPI Programming and Subroutine Reference
Version 2 Release 4

Publication No. GC23-3894-03

You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Today's date: ____________________________________

What is your occupation?

Newsletter number of latest Technical Newsletter (if any) concerning this publication:

How did you use this publication?

[ ]	As an introduction	[ ]	As a text (student)
[ ]	As a reference manual	[ ]	As a text (instructor)
[ ]	For another purpose (explain)		

Is there anything you especially like or dislike about the organization, presentation, or writing in this manual? Helpful comments include general usefulness of the book; possible additions, deletions, and clarifications; specific errors and omissions.

Page Number: Comment:

Name	Address
Company or Organization	
Phone No.	

Reader's Comments — We'd Like to Hear from You
GC23-3894-03

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Department 55JA, Mail Station P384
522 South Road
Poughkeepsie NY 12601-5400

Fold and Tape **Please do not staple** Fold and Tape

GC23-3894-03

Cut or Fold
Along Line

Program Number: 5765-543

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

GC23-3894-03