Parallel Environment Runtime Edition
Version 1 Release 3

MPI Subroutine Reference

SC23-6784-07

Parallel Environment Runtime Edition
Version 1 Release 3

MPI Subroutine Reference

SC23-6784-07

Note
Before using this information and the product it supports, read the information in "Notices" on page 617.

This edition applies to:

- version 1, release 3, modification 0 of IBM Parallel Environment Runtime Edition for Linux on x86 Architecture (product number 5641-PR1)
- version 1, release 3, modification 0 of IBM Parallel Environment Runtime Edition for Linux on x86 Architecture (product number 5641-PR2)
- version 1, release 3, modification 0 of IBM Parallel Environment Runtime Edition for Linux on X-Architecture (product number 5725-G00)

and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Tables

About this information

Attention

The functions or features found herein may not be available on all operating systems or platforms and do not indicate the availability of these functions or features within the IBM® product or future versions of the IBM product. The development, release, and timing of any future features or functionality is at IBM's sole discretion. IBM's plans, directions, and intent are subject to change or withdrawal without notice at IBM's sole discretion. The information mentioned is not a commitment, promise, or legal obligation to deliver any material, code or functionality. The information may not be incorporated into any contract and it should not be relied on in making a purchasing decision.

This information describes the subroutines provided by IBM's implementation of the Message Passing Interface (MPI) standard for IBM Parallel Environment Runtime Edition (IBM PE). Programmers can use these subroutines when writing parallel applications.

The IBM PE MPI product is a complete MPI implementation, designed to comply with all the requirements of the Message Passing Interface standard, *MPI: A Message-Passing Interface Standard, Version 2.2*, University of Tennessee, Knoxville, Tennessee, September 4, 2009.

IBM PE MPI provides support for all of the MPI 2.2 Enhancements. IBM PE Runtime Edition is also compliant with the revisions listed in the *Annex B Change-Log* of the MPI 2.2 standard.

If you believe that IBM PE MPI does not comply with the MPI standard, please contact IBM Service.

Information for Linux users

This information supports:

- IBM Parallel Environment Runtime Edition for Linux on X-Architecture® (5725-G00) Version 1 Release 3.
- IBM Parallel Environment Runtime Edition for Linux on x86 Architecture (5641-PR1) Version 1 Release 3.
- IBM Parallel Environment Runtime Edition for Linux on x86 Architecture (5641-PR2) Version 1 Release 3.

To make this information easier to read, the name *IBM Parallel Environment Runtime Edition* has been abbreviated to *IBM PE Runtime Edition*, *IBM PE for Linux*, *Parallel Environment*, or more generally, *IBM PE* throughout.

To use this information, you should be familiar with the Linux operating system. Where necessary, background information related to Linux is provided but, more commonly, it refers you to the appropriate documentation.

The IBM PE Runtime Edition for Linux information assumes that one of the currently-supported Linux distributions is already installed. For information about the supported Linux distributions, see *IBM Parallel Environment Runtime Edition: Installation*.

Who should use this information

This information is intended for experienced programmers who want to write parallel applications using the C, C++, or FORTRAN programming language. Readers of this information should know C, C++, and FORTRAN and should be familiar with AIX® or Linux 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.

Conventions and terminology used in this information

Table 1 shows the conventions used in this information:

Table 1. Conventions

Convention	Usage
bold	**Bold** words or characters represent system elements that you must use literally, such as commands, flags, path names, directories, file names, values, IBM PE Runtime Edition component names (**poe**, for example), and selected menu options.
bold underlined	**bold underlined** keywords are defaults. These take effect if you do not specify a different keyword.
`constant width`	Examples and information that the system displays appear in `constant-width` typeface.
italic	*Italic* words or characters represent variable values that you must supply. Italics are also used for unit titles, the first use of a glossary term, and general emphasis in text.
<*key*>	Angle brackets (less-than and greater-than) enclose the name of a key on the keyboard. For example, <**Enter**> refers to the key on your terminal or workstation that is labeled with the word *Enter*.

\	In command examples, a backslash indicates that the command or coding example continues on the next line. For example:
	`mkcondition -r IBM.FileSystem -e "PercentTotUsed > 90" \`
	`-E "PercentTotUsed < 85" -m d "FileSystem space used"`

{*item*}	Braces enclose a list from which you must choose an item in format and syntax descriptions.
[*item*]	Brackets enclose optional items in format and syntax descriptions.
<**Ctrl-***x*>	The notation <**Ctrl-*x***> indicates a control character sequence. For example, <**Ctrl-c**> means that you hold down the control key while pressing <**c**>.
item...	Ellipses indicate that you can repeat the preceding item one or more times.
|	• In *synopsis* statements, vertical lines separate a list of choices. In other words, a vertical line means *Or*. • In the margin of the document, vertical lines indicate technical changes to the information.

In addition to the highlighting conventions, this information uses the following conventions when describing how to perform tasks.

User actions appear in uppercase boldface type. For example, if the action is to enter the **tool** command, this information presents the instruction as:

ENTER
tool

Abbreviated names

Some of the abbreviated names used in this information follow.

AIX Advanced Interactive Executive

CSS communication subsystem

GUI graphical user interface

HFI Host Fabric Interface

IBM PE
IBM Parallel Environment Runtime Edition

IBM PE MPI
IBM's implementation of the MPI standard for IBM PE

IBM PE MPI-IO
IBM's implementation of MPI I/O for IBM PE

IP Internet Protocol

LAPI Low-level Application Programming Interface

LSF® IBM Platform Load Sharing Facility

MDCR
MetaCluster Distributed Checkpoint Restart

MPI Message Passing Interface

MPICH2
Implementation of the Message Passing Interface created by Argonne National Laboratory.

PAMI Parallel Active Messaging Interface

PDB Parallel Debugger

PNSD Protocol Network Services Daemon

POE Parallel Operating Environment

PTF Program Temporary Fix

rsh remote shell

SCI Scalable Communication Infrastructure

STDERR
standard error

STDIN
standard input

STDOUT
standard output

System x™
IBM System x

Prerequisite and related information

The Parallel Environment Runtime Edition for AIX and Linux library consists of:

- IBM Parallel Environment Runtime Edition: Installation, SC23-6780
- IBM Parallel Environment Runtime Edition: LAPI Programming Guide, SA23-2272
- IBM Parallel Environment Runtime Edition: Messages, SC23-6782
- IBM Parallel Environment Runtime Edition: MPI Programming Guide, SC23-6783
- IBM Parallel Environment Runtime Edition: MPI Subroutine Reference, SC23-6784
- IBM Parallel Environment Runtime Edition: NRT API Programming Guide, SC23-6785
- IBM Parallel Environment Runtime Edition: OpenSHMEM Programming Guide, SA23-1353 (IBM PE for Linux only)
- IBM Parallel Environment Runtime Edition: Operation and Use, SC23-6781
- IBM Parallel Environment Runtime Edition: PAMI Programming Guide, SA23-2273

To access the most recent Parallel Environment Runtime Edition documentation in PDF and HTML format, refer to the IBM Clusters Information Center (**http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp**), on the Web.

Both the current Parallel Environment Runtime Edition books and earlier versions of the library are also available in PDF format from the IBM Publication Center (**http://www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss**), on the Web.

It is easiest to locate a book in the IBM Publications Center by supplying the book's publication number. The publication number for each of the Parallel Environment books is listed after the book title in the preceding list.

You may also have the related product, IBM Parallel Environment Developer Edition. The IBM PE Developer Edition contains the IBM High Performance Toolkit (HPC Toolkit), which is a collection of tools that allow you to analyze the performance of both parallel and serial applications, written in C or FORTRAN, over the AIX or Linux operating system.

The IBM PE Developer Edition documentation can be found in the IBM Clusters Information Center (**http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp**), the IBM Publication Center (**http://www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss**), and also from the HPC Central Web site (**http://www.ibm.com/developerworks/wikis/display/hpccentral/HPC+Central**).

IBM Platform LSF (Load Sharing Facility) also works in conjunction with IBM PE Runtime Edition. The LSF publications that are referenced from within the IBM PE documentation can be found in the IBM Clusters Information Center (**http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp** and the IBM Publication Center (**http://www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss**).

How to send your comments

Your feedback is important in helping to provide the most accurate and high-quality information. If you have comments about this information or other IBM PE Runtime Edition documentation:

- Send your comments by e-mail to: mhvrcfs@us.ibm.com

  Be sure to include the name of the book, the part number of the book, the version of IBM PE Runtime Edition, and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).
- Fill out one of the forms at the back of this book and return it by mail, by fax, or by giving it to an IBM representative.

Summary of changes

IBM Parallel Environment Runtime Edition 1.3 contains the following functional changes.

- Support for detecting deadlock conditions in 64-bit MPI applications (IBM PE for Linux on x86-based servers or Power Systems™ servers only).
- Support for submitting and managing POE parallel jobs by way of IBM Platform Load Sharing Facility (LSF).
- For Linux applications, there is now a common set of RPMs required for installation, regardless of the Linux distribution you use.
- A new OpenSSH host-based authentication mechanism, which uses only host public keys, thereby eliminating the need for users to maintain keys (applies to IBM PE Runtime Edition 1.3 or later only).
- For MPICH2 applications, expanded support of IBM PE Runtime Edition function such as dynamic process management and memory management for early arrivals.
- For Linux applications running on x86-based servers with the InfiniBand interconnect, support for extended reliable connected (XRC) communication mode for better scalability.
- A mechanism for choosing the optimal collective algorithm for your system. See the *IBM Parallel Environment Runtime Edition: PAMI Programming Guide* for more information.

Chapter 1. A sample MPI subroutine

IBM PE MPI subroutines and functions are available for use in parallel programming.

For each subroutine or function, there are descriptions of some or all of the following headings, as appropriate:

- C synopsis
- C++ synopsis
- Fortran synopsis
- Description
- Parameters
- Notes
- Errors
- Related information

Review this sample before proceeding to better understand how the subroutine and function descriptions are structured.

A_SAMPLE_MPI_SUBROUTINE, A_Sample_MPI_subroutine

Provides a brief description of the subroutine or function.

C synopsis

Header file *mpi.h* supplies ANSI-C prototypes for every subroutine and function described in Chapter 3, "MPI subroutines and functions," on page 47.

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

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

C++ synopsis

```
#include mpi.h
type MPI::A_Sample_MPI_subroutine(one or more parameters);
```

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

For information about predefined constants for C++, see *IBM Parallel Environment Runtime Edition: MPI Programming Guide*.

Fortran synopsis

```
include 'mpif.h' or USE MPI
A_SAMPLE_MPI_SUBROUTINE (ONE OR MORE PARAMETERS);
```

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

Description

A detailed description of the subroutine or function.

Parameters

A list of argument or parameter definitions, as follows:

parameter_1
description of *parameter_1* (type)

⋮

parameter_n
description of *parameter_n* (type)

IERROR
is the Fortran return code. It is always the last argument.

Parameter types:

IN A call uses this parameter, but does not update an argument.

INOUT
A call uses this parameter and updates an argument.

OUT A call returns information by way of an argument, but does not use its input value.

Notes®

If applicable, contains notes about IBM PE MPI, as it relates to the requirements of the MPI standard. IBM PE MPI intends to comply fully with the requirements of the MPI standard. There are some issues, however, that the MPI 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 three 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 so you can identify the cause of failure.
- *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 so you can identify the cause of failure.

In almost every subroutine, the C version is invoked as a function returning integer. The Fortran version takes one more argument than the C version, which is used to return any error value.

For more information about errors, see *IBM Parallel Environment Runtime Edition: Messages*, which provides a listing of all the error messages issued as well as the error class to which the message belongs.

Related information

A list of the related subroutines or functions.

For C and Fortran, MPI uses the same spelling for subroutine names. The only distinction is the capitalization. For the purpose of clarity, when referring to a subroutine without specifying whether it is the Fortran version or the 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 offer array section and assumed shape arrays as parameters on calls. These are not safe with MPI.

Chapter 2. Nonblocking collective communication subroutines

There are a number of nonblocking collective communication subroutines that are available for parallel programming. These subroutines, which have a prefix of **MPE_I**, are extensions of the MPI standard.

The nonblocking collective communication subroutines that are provided by IBM PE MPI, which have a prefix of **MPE_I**, are extensions of the MPI standard. They are part of IBM's implementation of the MPI standard for IBM PE Runtime Edition. Review Chapter 1, "A sample MPI subroutine," on page 1 before proceeding to better understand how the subroutine descriptions are structured.

Note: MPICH2 does not support these nonblocking collective communication subroutines.

The **MPE_I** nonblocking collectives provided by IBM PE MPI were well-suited to the early implementation of MPI on AIX, which used signals for asynchronous progress. They are not well-suited to a threads-based implementation of MPI. The **MPE_I** nonblocking collectives will remain supported by the MPI library, but **the use of these nonstandard subroutines, in new or restructured applications, is now deprecated**.

Early versions of IBM PE MPI allowed the mixing of **MPE_I** (nonblocking) and **MPI_** (blocking) calls in a single collective operation. With IBM PE Version 4, there is a new shared memory based optimization for certain MPI collective operations, available in 64-bit executables and enabled by default. The shared memory optimization is not suitable for nonblocking collectives. With this optimization enabled, affected collective operations that mix blocking and nonblocking calls will deadlock. **MP_EUIDEVELOP** mode has been enhanced to detect this mix and issue an error message. For more information on the shared memory optimization, refer to the description of **MP_SHARED_MEMORY** in *IBM Parallel Environment Runtime Edition: MPI Programming Guide*.

For more information about matching blocking and nonblocking collectives in the same application, see the chapter *Programming considerations for user application in POE* in *IBM Parallel Environment Runtime Edition: MPI Programming Guide*.

The MPI-2 extensions related to collective communication are now available for all MPI blocking collectives. The **MPE_I** nonblocking collectives have not been enhanced with MPI-2 functionality. The **MPE_I** nonblocking collectives are semantically equivalent to MPI-1.

MPE_IALLGATHER, MPE_Iallgather

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' or USE MPI
MPE_IALLGATHER(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,
    CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER COMM,
    INTEGER REQUEST,INTEGER IERROR)
```

Description

This subroutine 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.

Parameters

sendbuf
The starting address of the send buffer (choice) (IN)

sendcount
The number of elements in the send buffer (integer) (IN)

sendtype
The data type of the send buffer elements (handle) (IN)

recvbuf
The address of the receive buffer (choice) (OUT)

recvcount
The number of elements received from any task (integer) (IN)

recvtype
The data type of the receive buffer elements (handle) (IN)

comm
The communicator (handle) (IN)

request
The communication request (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

The **MPE_I** nonblocking collectives provided by IBM PE MPI were well-suited to the early implementation of MPI on AIX, which used signals for asynchronous progress. They are not well-suited to a threads-based implementation of MPI. The **MPE_I** nonblocking collectives will remain supported by the MPI library, but **the use of these nonstandard subroutines, in new or restructured applications, is now deprecated**.

The MPE prefix used with this subroutine 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 than the nonblocking versions.

Early versions of IBM PE MPI allowed the mixing of **MPE_I** (nonblocking) and **MPI_** (blocking) calls in a single collective operation. With IBM PE Version 4, there is a new shared memory based optimization for certain MPI collective operations, available in 64-bit executables and enabled by default. The shared memory optimization is not suitable for nonblocking collectives, so with this optimization enabled, affected collective operations that mix blocking and nonblocking calls will deadlock. **MP_EUIDEVELOP** mode has been enhanced to detect this mix and issue an error message. For further information on the shared memory optimization, refer to the description of **MP_SHARED_MEMORY** in *IBM Parallel Environment Runtime Edition: MPI Programming Guide*.

The IBM PE/MPI library has a limit of seven 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 any call defined by the MPI standard.

Applications using nonblocking collective calls often perform best when they run in interrupt mode.

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator are started in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

MPE_I routines have not been enhanced to use MPI-2 extensions. This routine, and all **MPE_I** nonblocking collectives are semantically equivalent to MPI-1.

Use of **MPE_I** nonblocking collective communications rules out setting environment variable **MP_SINGLE_THREAD**, or the command-line flag **-single_thread** to **yes**.

Errors

Invalid communicator

Invalid communicator type
must be intra-communicator

Invalid counts
count < 0

Invalid datatypes

Type not committed

Unequal message length

MPI_IN_PLACE not valid

MPI not initialized

MPI already finalized

Develop mode error (returned in the WAIT) if:

Inconsistent message length

Match of blocking and non-blocking collectives

Related information

- MPI_ALLGATHER

MPE_IALLGATHERV, MPE_Iallgatherv

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' or USE MPI
MPE_IALLGATHERV(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,
     CHOICE RECVBUF,INTEGER RECVCOUNTS(*),INTEGER DISPLS(*),
     INTEGER RECVTYPE,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Description

This subroutine 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.

Parameters

sendbuf
The starting address of the send buffer (choice) (IN)

sendcount
The number of elements in the send buffer (integer) (IN)

sendtype
The data type of the send buffer elements (handle) (IN)

recvbuf
The address of the receive buffer (choice) (OUT)

recvcounts
An integer array (of length group size) that contains the number of elements received from each task (IN)

displs
An 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
The data type of the receive buffer elements (handle) (IN)

comm
The communicator (handle) (IN)

request
The communication request (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

The **MPE_I** nonblocking collectives provided by IBM PE MPI were well-suited to the early implementation of MPI on AIX, which used signals for asynchronous progress. They are not well-suited to a threads-based implementation of MPI. The

MPE_I nonblocking collectives will remain supported by the MPI library, but **the use of these nonstandard subroutines, in new or restructured applications, is now deprecated**.

The MPE prefix used with this subroutine 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 than the nonblocking versions.

Early versions of IBM PE MPI allowed the mixing of **MPE_I** (nonblocking) and **MPI_** (blocking) calls in a single collective operation. With IBM PE Version 4, there is a new shared memory based optimization for certain MPI collective operations, available in 64-bit executables and enabled by default. The shared memory optimization is not suitable for nonblocking collectives, so with this optimization enabled, affected collective operations that mix blocking and nonblocking calls will deadlock. **MP_EUIDEVELOP** mode has been enhanced to detect this mix and issue an error message. For further information on the shared memory optimization, refer to the description of **MP_SHARED_MEMORY** in *IBM Parallel Environment Runtime Edition Runtime Edition: MPI Programming Guide*.

The IBM PE/MPI library has a limit of seven 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 any call defined by the MPI standard.

Applications using nonblocking collective calls often perform best when they run in interrupt mode.

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator are started in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

MPE_I routines have not been enhanced to use MPI-2 extensions. This routine, and all **MPE_I** nonblocking collectives are semantically equivalent to MPI-1.

Use of **MPE_I** nonblocking collective communications rules out setting environment variable **MP_SINGLE_THREAD**, or the command-line flag **-single_thread** to **yes**.

Errors

Invalid communicator

Invalid communicator type
must be intra-communicator

Invalid counts
count < 0

Invalid datatypes

Type not committed

Unequal message length

MPI_IN_PLACE not valid

MPI not initialized

MPI already finalized

Develop mode error (returned in the WAIT) if:

Match of blocking and non-blocking collectives

Related information

- MPI_ALLGATHERV

MPE_IALLREDUCE, MPE_Iallreduce

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' or USE MPI
MPE_IALLREDUCE(CHOICE SENDBUF,CHOICE RECVBUF,INTEGER COUNT,
    INTEGER DATATYPE,INTEGER OP,INTEGER COMM,INTEGER REQUEST,
    INTEGER IERROR)
```

Description

This subroutine 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.

Parameters

sendbuf
The starting address of the send buffer (choice) (IN)

recvbuf
The starting address of the receive buffer (choice) (OUT)

count
The number of elements in the send buffer (integer) (IN)

datatype
The data type of elements in the send buffer (handle) (IN)

op The reduction operation (handle) (IN)

comm
The communicator (handle) (IN)

request
The communication request (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

The **MPE_I** nonblocking collectives provided by IBM PE MPI were well-suited to the early implementation of MPI on AIX, which used signals for asynchronous progress. They are not well-suited to a threads-based implementation of MPI. The **MPE_I** nonblocking collectives will remain supported by the MPI library, but **the use of these nonstandard subroutines, in new or restructured applications, is now deprecated**.

The MPE prefix used with this subroutine 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 than the nonblocking versions.

Early versions of IBM PE MPI allowed the mixing of **MPE_I** (nonblocking) and **MPI_** (blocking) calls in a single collective operation. With IBM PE Version 4, there is a new shared memory based optimization for certain MPI collective operations, available in 64-bit executables and enabled by default. The shared memory optimization is not suitable for nonblocking collectives, so with this optimization enabled, affected collective operations that mix blocking and nonblocking calls will deadlock. **MP_EUIDEVELOP** mode has been enhanced to detect this mix and issue an error message. For further information on the shared memory optimization, refer to the description of **MP_SHARED_MEMORY** in *IBM Parallel Environment Runtime Edition: MPI Programming Guide*.

The IBM PE/MPI library has a limit of seven 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 any call defined by the MPI standard.

Applications using nonblocking collective calls often perform best when they run in interrupt mode.

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator are started in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

MPE_I routines have not been enhanced to use MPI-2 extensions. This routine, and all **MPE_I** nonblocking collectives are semantically equivalent to MPI-1.

Use of **MPE_I** nonblocking collective communications rules out setting environment variable **MP_SINGLE_THREAD**, or the command-line flag **-single_thread** to **yes**.

Errors

Invalid count
: **count** < 0

Invalid datatype

Type not committed

Invalid op

Invalid communicator

Invalid communicator type
: must be intra-communicator

Unequal message length

MPI_IN_PLACE not valid

MPI not initialized

MPI already finalized

Develop mode error (returned in the WAIT) if:

Inconsistent datatype

Inconsistent message length

Inconsistent op

Match of blocking and non-blocking collectives

Related information

- MPI_ALLREDUCE

MPE_IALLTOALL, MPE_Ialltoall

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' or USE MPI
MPE_IALLTOALL(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,
    CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER COMM,
    INTEGER REQUEST,INTEGER IERROR)
```

Description

This subroutine 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.

Parameters

sendbuf
The starting address of the send buffer (choice) (IN)

sendcount
The number of elements sent to each task (integer) (IN)

sendtype
The data type of the send buffer elements (handle) (IN)

recvbuf
The address of the receive buffer (choice) (OUT)

recvcount
The number of elements received from any task (integer) (IN)

recvtype
The data type of the receive buffer elements (handle) (IN)

comm
The communicator (handle) (IN)

request
The communication request (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

The **MPE_I** nonblocking collectives provided by IBM PE MPI were well-suited to the early implementation of MPI on AIX, which used signals for asynchronous progress. They are not well-suited to a threads-based implementation of MPI. The **MPE_I** nonblocking collectives will remain supported by the MPI library, but **the use of these nonstandard subroutines, in new or restructured applications, is now deprecated**.

The MPE prefix used with this subroutine 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 than the nonblocking versions.

Early versions of IBM PE MPI allowed the mixing of **MPE_I** (nonblocking) and **MPI_** (blocking) calls in a single collective operation. With IBM PE Version 4, there is a new shared memory based optimization for certain MPI collective operations, available in 64-bit executables and enabled by default. The shared memory optimization is not suitable for nonblocking collectives, so with this optimization enabled, affected collective operations that mix blocking and nonblocking calls will deadlock. **MP_EUIDEVELOP** mode has been enhanced to detect this mix and issue an error message. For further information on the shared memory optimization, refer to the description of **MP_SHARED_MEMORY** in *IBM Parallel Environment Runtime Edition: MPI Programming Guide*.

The IBM PE/MPI library has a limit of seven 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 any call defined by the MPI standard.

Applications using nonblocking collective calls often perform best when they run in interrupt mode.

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator are started in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

MPE_I routines have not been enhanced to use MPI-2 extensions. This routine, and all **MPE_I** nonblocking collectives are semantically equivalent to MPI-1.

Use of **MPE_I** nonblocking collective communications rules out setting environment variable **MP_SINGLE_THREAD**, or the command-line flag **-single_thread** to **yes**.

Errors

Invalid counts
: **count** < 0

Invalid datatypes

Type not committed

Invalid communicator

Invalid communicator type
: must be intra-communicator

Unequal message lengths

MPI_IN_PLACE not valid

MPI not initialized

MPI already finalized

Develop mode error (returned in the WAIT) if:

Inconsistent message length

Match of blocking and non-blocking collectives

Related information

- MPI_ALLTOALL

MPE_IALLTOALLV, MPE_Ialltoallv

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' or USE MPI
MPE_ALLTOALLV(CHOICE SENDBUF,INTEGER SENDCOUNTS(*),
    INTEGER SDISPLS(*),INTEGER SENDTYPE,CHOICE RECVBUF,
    INTEGER RECVCOUNTS(*),INTEGER RDISPLS(*),INTEGER RECVTYPE,
    INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Description

This subroutine 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.

Parameters

sendbuf
The starting address of the send buffer (choice) (IN)

sendcounts
An integer array (of length group size) specifying the number of elements to send to each task (IN)

sdispls
An 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
The data type of the send buffer elements (handle) (IN)

recvbuf
The address of the receive buffer (choice) (OUT)

recvcounts
An integer array (of length group size) specifying the number of elements that can be received from each task (IN)

rdispls
An 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
The data type of the receive buffer elements (handle) (IN)

comm
The communicator (handle) (IN)

request
The communication request (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

The **MPE_I** nonblocking collectives provided by IBM PE MPI were well-suited to the early implementation of MPI on AIX, which used signals for asynchronous progress. They are not well-suited to a threads-based implementation of MPI. The **MPE_I** nonblocking collectives will remain supported by the MPI library, but **the use of these nonstandard subroutines, in new or restructured applications, is now deprecated**.

The MPE prefix used with this subroutine 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 than the nonblocking versions.

Early versions of IBM PE MPI allowed the mixing of **MPE_I** (nonblocking) and **MPI_** (blocking) calls in a single collective operation. With IBM PE Version 4, there is a new shared memory based optimization for certain MPI collective operations, available in 64-bit executables and enabled by default. The shared memory optimization is not suitable for nonblocking collectives, so with this optimization enabled, affected collective operations that mix blocking and nonblocking calls will deadlock. **MP_EUIDEVELOP** mode has been enhanced to detect this mix and issue an error message. For further information on the shared memory optimization, refer to the description of **MP_SHARED_MEMORY** in *IBM Parallel Environment Runtime Edition: MPI Programming Guide*.

The IBM PE/MPI library has a limit of seven 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 any call defined by the MPI standard.

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator are started in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

MPE_I routines have not been enhanced to use MPI-2 extensions. This routine, and all **MPE_I** nonblocking collectives are semantically equivalent to MPI-1.

Use of **MPE_I** nonblocking collective communications rules out setting environment variable **MP_SINGLE_THREAD**, or the command-line flag **-single_thread** to **yes**.

Errors

Invalid counts
: **count** < 0

Invalid datatypes

Type not committed

Invalid communicator

Invalid communicator type
must be intra-communicator

A send and receive have unequal message lengths

MPI_IN_PLACE not valid

MPI not initialized

MPI already finalized

Develop mode error (returned in the WAIT) if:

Match of blocking and non-blocking collectives

Related information

- MPI_ALLTOALLV

MPE_IBARRIER, MPE_Ibarrier

Performs a nonblocking barrier operation.

C synopsis

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

Fortran synopsis

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

Description

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

Parameters

comm
A communicator (handle) (IN)

request
The communication request (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

The **MPE_I** nonblocking collectives provided by IBM PE MPI were well-suited to the early implementation of MPI on AIX, which used signals for asynchronous progress. They are not well-suited to a threads-based implementation of MPI. The **MPE_I** nonblocking collectives will remain supported by the MPI library, but **the use of these nonstandard subroutines, in new or restructured applications, is now deprecated**.

The MPE prefix used with this subroutine 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 than 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.

Early versions of IBM PE MPI allowed the mixing of **MPE_I** (nonblocking) and **MPI_** (blocking) calls in a single collective operation. With IBM PE Version 4, there is a new shared memory based optimization for certain MPI collective operations, available in 64-bit executables and enabled by default. The shared memory optimization is not suitable for nonblocking collectives, so with this optimization enabled, affected collective operations that mix blocking and nonblocking calls will

deadlock. **MP_EUIDEVELOP** mode has been enhanced to detect this mix and issue an error message. For further information on the shared memory optimization, refer to the description of **MP_SHARED_MEMORY** in *IBM Parallel Environment Runtime Edition: MPI Programming Guide*.

The IBM PE/MPI library has a limit of seven 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 any call defined by the MPI standard.

Applications using nonblocking collective calls often perform best when they run in interrupt mode.

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator are started in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

MPE_I routines have not been enhanced to use MPI-2 extensions. This routine, and all **MPE_I** nonblocking collectives are semantically equivalent to MPI-1.

Use of **MPE_I** nonblocking collective communications rules out setting environment variable **MP_SINGLE_THREAD**, or the command-line flag **-single_thread** to **yes**.

Errors

Invalid communicator

Invalid communicator type
must be intra-communicator

MPI not initialized

MPI already finalized

Develop mode error (returned in the WAIT) if:

Match of blocking and non-blocking collectives

Related information

- MPI_BARRIER

MPE_IBCAST, MPE_Ibcast

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' or USE MPI
MPE_IBCAST(CHOICE BUFFER,INTEGER COUNT,INTEGER DATATYPE,INTEGER ROOT,
    INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Description

This subroutine 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.

Parameters

buffer
The starting address of the buffer (choice) (INOUT)

count
The number of elements in the buffer (integer) (IN)

datatype
The data type of the buffer elements (handle) (IN)

root
The rank of the root task (integer) (IN)

comm
The communicator (handle) (IN)

request
The communication request (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

The **MPE_I** nonblocking collectives provided by IBM PE MPI were well-suited to the early implementation of MPI on AIX, which used signals for asynchronous progress. They are not well-suited to a threads-based implementation of MPI. The **MPE_I** nonblocking collectives will remain supported by the MPI library, but **the use of these nonstandard subroutines, in new or restructured applications, is now deprecated**.

The MPE prefix used with this subroutine 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 than the nonblocking versions.

Early versions of IBM PE MPI allowed the mixing of **MPE_I** (nonblocking) and **MPI_** (blocking) calls in a single collective operation. With IBM PE Version 4, there is a new shared memory based optimization for certain MPI collective operations, available in 64-bit executables and enabled by default. The shared memory optimization is not suitable for nonblocking collectives, so with this optimization enabled, affected collective operations that mix blocking and nonblocking calls will deadlock. **MP_EUIDEVELOP** mode has been enhanced to detect this mix and issue an error message. For further information on the shared memory optimization, refer to the description of **MP_SHARED_MEMORY** in *IBM Parallel Environment Runtime Edition: MPI Programming Guide*.

The IBM PE/MPI library has a limit of seven 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 any call defined by the MPI standard.

Applications using nonblocking collective calls often perform best when they run in interrupt mode.

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator are started in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

MPE_I routines have not been enhanced to use MPI-2 extensions. This routine, and all **MPE_I** nonblocking collectives are semantically equivalent to MPI-1.

Use of **MPE_I** nonblocking collective communications rules out setting environment variable **MP_SINGLE_THREAD**, or the command-line flag **-single_thread** to **yes**.

Errors

Error Conditions:

Invalid communicator

Invalid communicator type
must be intra-communicator

Invalid count
count < 0

Invalid datatype

Type not committed

Invalid root
root < 0 or **root** >= groupsize

Unequal message length

MPI_IN_PLACE not valid

MPI not initialized

MPI already finalized

Develop mode error (returned in the WAIT) if:

Inconsistent message length

Inconsistent root

Match of blocking and non-blocking collectives

Related information

- MPI_BCAST

MPE_IGATHER, MPE_Igather

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' or USE MPI
MPE_IGATHER(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,
    CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER ROOT,
    INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Description

This subroutine 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.

Parameters

sendbuf
The starting address of the send buffer (choice) (IN)

sendcount
The number of elements in the send buffer (integer) (IN)

sendtype
The data type of the send buffer elements (integer) (IN)

recvbuf
The address of the receive buffer (choice, significant only at **root**) (OUT)

recvcount
The number of elements for any single receive (integer, significant only at **root**) (IN)

recvtype
The data type of the receive buffer elements (handle, significant at **root**) (IN)

root
The rank of the receiving task (integer) (IN)

comm
The communicator (handle) (IN)

request
The communication request (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

The **MPE_I** nonblocking collectives provided by IBM PE MPI were well-suited to the early implementation of MPI on AIX, which used signals for asynchronous progress. They are not well-suited to a threads-based implementation of MPI. The

MPE_I nonblocking collectives will remain supported by the MPI library, but **the use of these nonstandard subroutines, in new or restructured applications, is now deprecated**.

The MPE prefix used with this subroutine 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 than the nonblocking versions.

Early versions of IBM PE MPI allowed the mixing of **MPE_I** (nonblocking) and **MPI_** (blocking) calls in a single collective operation. With IBM PE Version 4, there is a new shared memory based optimization for certain MPI collective operations, available in 64-bit executables and enabled by default. The shared memory optimization is not suitable for nonblocking collectives, so with this optimization enabled, affected collective operations that mix blocking and nonblocking calls will deadlock. **MP_EUIDEVELOP** mode has been enhanced to detect this mix and issue an error message. For further information on the shared memory optimization, refer to the description of **MP_SHARED_MEMORY** in *IBM Parallel Environment Runtime Edition: MPI Programming Guide*.

The IBM PE/MPI library has a limit of seven 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 any call defined by the MPI standard.

Applications using nonblocking collective calls often perform best when they run in interrupt mode.

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator are started in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

MPE_I routines have not been enhanced to use MPI-2 extensions. This routine, and all **MPE_I** nonblocking collectives are semantically equivalent to MPI-1.

Use of **MPE_I** nonblocking collective communications rules out setting environment variable **MP_SINGLE_THREAD**, or the command-line flag **-single_thread** to **yes**.

Errors

Invalid communicator

Invalid communicator type
must be intra-communicator

Invalid counts
count < 0

Invalid datatypes

Type not committed

Invalid root
root < 0 or **root** >= groupsize

Unequal message lengths

MPI_IN_PLACE not valid

MPI not initialized

MPI already finalized

Develop mode error (returned in the WAIT) if:

Inconsistent message length

Inconsistent root

Match of blocking and non-blocking collectives

Related information

- MPI_GATHER

MPE_IGATHERV, MPE_Igatherv

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' or USE MPI
MPE_IGATHERV(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,
    CHOICE RECVBUF,INTEGER RECVCOUNTS(*),INTEGER DISPLS(*),
    INTEGER RECVTYPE,INTEGER ROOT,INTEGER COMM,INTEGER REQUEST,
    INTEGER IERROR)
```

Description

This subroutine 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.

Parameters

sendbuf
The starting address of the send buffer (choice) (IN)

sendcount
The number of elements to be sent (integer) (IN)

sendtype
The data type of the send buffer elements (handle) (IN)

recvbuf
The address of the receive buffer (choice, significant only at **root**) (OUT)

recvcounts
An integer array (of length group size) that contains the number of elements received from each task (significant only at **root**) (IN)

displs
An 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
The data type of the receive buffer elements (handle, significant only at **root**) (IN)

root
The rank of the receiving task (integer) (IN)

comm
The communicator (handle) (IN)

request
The communication request (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

The **MPE_I** nonblocking collectives provided by IBM PE MPI were well-suited to the early implementation of MPI on AIX, which used signals for asynchronous progress. They are not well-suited to a threads-based implementation of MPI. The **MPE_I** nonblocking collectives will remain supported by the MPI library, but **the use of these nonstandard subroutines, in new or restructured applications, is now deprecated**.

The MPE prefix used with this subroutine 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 than the nonblocking versions.

Early versions of IBM PE MPI allowed the mixing of **MPE_I** (nonblocking) and **MPI_** (blocking) calls in a single collective operation. With IBM PE Version 4, there is a new shared memory based optimization for certain MPI collective operations, available in 64-bit executables and enabled by default. The shared memory optimization is not suitable for nonblocking collectives, so with this optimization enabled, affected collective operations that mix blocking and nonblocking calls will deadlock. **MP_EUIDEVELOP** mode has been enhanced to detect this mix and issue an error message. For further information on the shared memory optimization, refer to the description of **MP_SHARED_MEMORY** in *IBM Parallel Environment Runtime Edition: MPI Programming Guide*.

The IBM PE/MPI library has a limit of seven 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 any call defined by the MPI standard.

Applications using nonblocking collective calls often perform best when they run in interrupt mode.

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator are started in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

MPE_I routines have not been enhanced to use MPI-2 extensions. This routine, and all **MPE_I** nonblocking collectives are semantically equivalent to MPI-1.

Use of **MPE_I** nonblocking collective communications rules out setting environment variable **MP_SINGLE_THREAD**, or the command-line flag **-single_thread** to **yes**.

Errors

Invalid communicator

Invalid communicator type
must be intra-communicator

Invalid counts

Invalid datatypes

Type not committed

Invalid root
root < 0 or **root** >= groupsize

A send and receive have unequal message lengths

MPI_IN_PLACE not valid

MPI not initialized

MPI already finalized

Develop mode error (returned in the WAIT) if:

Inconsistent root

Match of blocking and non-blocking collectives

Related information

- MPI_GATHERV

MPE_IREDUCE, MPE_Ireduce

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' or USE MPI
MPE_IREDUCE(CHOICE SENDBUF,CHOICE RECVBUF,INTEGER COUNT,
    INTEGER DATATYPE,INTEGER OP,INTEGER ROOT,INTEGER COMM,
    INTEGER REQUEST,INTEGER IERROR)
```

Description

This subroutine 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.

Parameters

sendbuf
The address of the send buffer (choice) (IN)

recvbuf
The address of the receive buffer (choice, significant only at root) (OUT)

count
The number of elements in the send buffer (integer) (IN)

datatype
The data type of elements of the send buffer (handle) (IN)

op The reduction operation (handle) (IN)

root
The rank of the root task (integer) (IN)

comm
The communicator (handle) (IN)

request
The communication request (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

The **MPE_I** nonblocking collectives provided by IBM PE MPI were well-suited to the early implementation of MPI on AIX, which used signals for asynchronous progress. They are not well-suited to a threads-based implementation of MPI. The **MPE_I** nonblocking collectives will remain supported by the MPI library, but **the use of these nonstandard subroutines, in new or restructured applications, is now deprecated**.

The MPE prefix used with this subroutine 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 than the nonblocking versions.

Early versions of IBM PE MPI allowed the mixing of **MPE_I** (nonblocking) and **MPI_** (blocking) calls in a single collective operation. With IBM PE Version 4, there is a new shared memory based optimization for certain MPI collective operations, available in 64-bit executables and enabled by default. The shared memory optimization is not suitable for nonblocking collectives, so with this optimization enabled, affected collective operations that mix blocking and nonblocking calls will deadlock. **MP_EUIDEVELOP** mode has been enhanced to detect this mix and issue an error message. For further information on the shared memory optimization, refer to the description of **MP_SHARED_MEMORY** in *IBM Parallel Environment Runtime Edition: MPI Programming Guide*.

The IBM PE/MPI library has a limit of seven 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 any call defined by the MPI standard.

Applications using nonblocking collective calls often perform best when they run in interrupt mode.

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator are started in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

MPE_I routines have not been enhanced to use MPI-2 extensions. This routine, and all **MPE_I** nonblocking collectives are semantically equivalent to MPI-1.

Use of **MPE_I** nonblocking collective communications rules out setting environment variable **MP_SINGLE_THREAD**, or the command-line flag **-single_thread** to **yes**.

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 intra-communicator

Unequal message lengths

MPI_IN_PLACE not valid

MPI not initialized

MPI already finalized

Develop mode error (returned in the WAIT) if:

Inconsistent datatype

Inconsistent message length

Inconsistent op

Inconsistent root

Match of blocking and non-blocking collectives

Related information

- MPI_REDUCE

MPE_IREDUCE_SCATTER, MPE_Ireduce_scatter

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' or USE MPI
MPE_IREDUCE_SCATTER(CHOICE SENDBUF,CHOICE RECVBUF,
     INTEGER RECVCOUNTS(*),INTEGER DATATYPE,INTEGER OP,
     INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Description

This subroutine 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.

Parameters

sendbuf
The starting address of the send buffer (choice) (IN)

recvbuf
The starting address of the receive buffer (choice) (OUT)

recvcounts
An integer array specifying the number of elements in result distributed to each task. Must be identical on all calling tasks. (IN)

datatype
The data type of elements in the input buffer (handle) (IN)

op The reduction operation (handle) (IN)

comm
The communicator (handle) (IN)

request
The communication request (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

The **MPE_I** nonblocking collectives provided by IBM PE MPI were well-suited to the early implementation of MPI on AIX, which used signals for asynchronous progress. They are not well-suited to a threads-based implementation of MPI. The **MPE_I** nonblocking collectives will remain supported by the MPI library, but **the use of these nonstandard subroutines, in new or restructured applications, is now deprecated**.

The MPE prefix used with this subroutine 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 than the nonblocking versions.

Early versions of IBM PE MPI allowed the mixing of **MPE_I** (nonblocking) and **MPI_** (blocking) calls in a single collective operation. With IBM PE Version 4, there is a new shared memory based optimization for certain MPI collective operations, available in 64-bit executables and enabled by default. The shared memory optimization is not suitable for nonblocking collectives, so with this optimization enabled, affected collective operations that mix blocking and nonblocking calls will deadlock. **MP_EUIDEVELOP** mode has been enhanced to detect this mix and issue an error message. For further information on the shared memory optimization, refer to the description of **MP_SHARED_MEMORY** in *IBM Parallel Environment Runtime Edition: MPI Programming Guide*.

The IBM PE/MPI library has a limit of seven 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 any call defined by the MPI standard.

Applications using nonblocking collective calls often perform best when they run in interrupt mode.

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator are started in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

MPE_I routines have not been enhanced to use MPI-2 extensions. This routine, and all **MPE_I** nonblocking collectives are semantically equivalent to MPI-1.

Use of **MPE_I** nonblocking collective communications rules out setting environment variable **MP_SINGLE_THREAD**, or the command-line flag **-single_thread** to **yes**.

Errors

Invalid recvcounts
: **recvcounts(i)** < 0

Invalid datatype

Type not committed

Invalid op

Invalid communicator

Invalid communicator type
: must be intra-communicator

Unequal message lengths

MPI_IN_PLACE not valid

MPI not initialized

MPI already finalized

Develop mode error (returned in the WAIT) if:

Inconsistent datatype

Inconsistent op

Match of blocking and non-blocking collectives

Related information

- MPI_REDUCE_SCATTER

MPE_ISCAN, MPE_Iscan

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' or USE MPI
MPE_ISCAN(CHOICE SENDBUF,CHOICE RECVBUF,INTEGER COUNT,
    INTEGER DATATYPE,INTEGER OP,INTEGER COMM,INTEGER REQUEST,
    INTEGER IERROR)
```

Description

This subroutine 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.

Parameters

sendbuf
The starting address of the send buffer (choice) (IN)

recvbuf
The starting address of the receive buffer (choice) (OUT)

count
The number of elements in **sendbuf** (integer) (IN)

datatype
The data type of elements in **sendbuf** (handle) (IN)

op The reduction operation (handle) (IN)

comm
The communicator (IN)

request
The communication request (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

The **MPE_I** nonblocking collectives provided by IBM PE MPI were well-suited to the early implementation of MPI on AIX, which used signals for asynchronous progress. They are not well-suited to a threads-based implementation of MPI. The **MPE_I** nonblocking collectives will remain supported by the MPI library, but **the use of these nonstandard subroutines, in new or restructured applications, is now deprecated**.

The MPE prefix used with this subroutine 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 than the nonblocking versions.

Early versions of IBM PE MPI allowed the mixing of **MPE_I** (nonblocking) and **MPI_** (blocking) calls in a single collective operation. With IBM PE Version 4, there is a new shared memory based optimization for certain MPI collective operations, available in 64-bit executables and enabled by default. The shared memory optimization is not suitable for nonblocking collectives, so with this optimization enabled, affected collective operations that mix blocking and nonblocking calls will deadlock. **MP_EUIDEVELOP** mode has been enhanced to detect this mix and issue an error message. For further information on the shared memory optimization, refer to the description of **MP_SHARED_MEMORY** in *IBM Parallel Environment Runtime Edition: MPI Programming Guide*.

The IBM PE/MPI library has a limit of seven 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 any call defined by the MPI standard.

Applications using nonblocking collective calls often perform best when they run in interrupt mode.

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator are started in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

MPE_I routines have not been enhanced to use MPI-2 extensions. This routine, and all **MPE_I** nonblocking collectives are semantically equivalent to MPI-1.

Use of **MPE_I** nonblocking collective communications rules out setting environment variable **MP_SINGLE_THREAD**, or the command-line flag **-single_thread** to **yes**.

Errors

Invalid count
: **count** < 0

Invalid datatype

Type not committed

Invalid op

Invalid communicator

Invalid communicator type
: must be intra-communicator

Unequal message lengths

MPI_IN_PLACE not valid

MPI not initialized

MPI already finalized

Develop mode error (returned in the WAIT) if:

Inconsistent datatype

Inconsistent message length

Inconsistent op

Match of blocking and non-blocking collectives

Related information

- MPI_SCAN

MPE_ISCATTER, MPE_Iscatter

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' or USE MPI
MPE_ISCATTER(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,
    CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER ROOT,
    INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Description

This subroutine 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.

Parameters

sendbuf
The address of the send buffer (choice, significant only at **root**) (IN)

sendcount
The number of elements to be sent to each task (integer, significant only at **root**) (IN)

sendtype
The data type of the send buffer elements (handle, significant only at **root**) (IN)

recvbuf
The address of the receive buffer (choice) (OUT)

recvcount
The number of elements in the receive buffer (integer) (IN)

recvtype
The data type of the receive buffer elements (handle) (IN)

root
The rank of the sending task (integer) (IN)

comm
The communicator (handle) (IN)

request
The communication request (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

The **MPE_I** nonblocking collectives provided by IBM PE MPI were well-suited to the early implementation of MPI on AIX, which used signals for asynchronous progress. They are not well-suited to a threads-based implementation of MPI. The

MPE_I nonblocking collectives will remain supported by the MPI library, but **the use of these nonstandard subroutines, in new or restructured applications, is now deprecated**.

The MPE prefix used with this subroutine 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 than the nonblocking versions.

Note: The IBM PE Fortran 90 type checking module (**mpi.mod**) was compiled without the **-qmixed** compiler option. As a result, routines that use the **mpi.mod** module must not be compiled with this option.

The IBM PE/MPI library has a limit of seven 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 any call defined by the MPI standard.

Applications using nonblocking collective calls often perform best when they run in interrupt mode.

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator are started in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

MPE_I routines have not been enhanced to use MPI-2 extensions. This routine, and all **MPE_I** nonblocking collectives are semantically equivalent to MPI-1.

Use of **MPE_I** nonblocking collective communications rules out setting environment variable **MP_SINGLE_THREAD**, or the command-line flag **-single_thread** to **yes**.

Errors

Invalid communicator

Invalid communicator type
must be intra-communicator

Invalid counts
count < 0

Invalid datatypes

Type not committed

Invalid root
root < 0 or **root** >= groupsize

Unequal message lengths

MPI_IN_PLACE not valid

MPI not initialized

MPI already finalized

Develop mode error (returned in the WAIT) if:

Inconsistent message length

Inconsistent root

Match of blocking and non-blocking collectives

Related information

- MPI_SCATTER

MPE_ISCATTERV, MPE_Iscatterv

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' or USE MPI
MPE_ISCATTERV(CHOICE SENDBUF,INTEGER SENDCOUNTS(*),INTEGER DISPLS(*),
    INTEGER SENDTYPE,CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,
    INTEGER ROOT,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Description

This subroutine 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.

Parameters

sendbuf
The address of the send buffer (choice, significant only at **root**) (IN)

sendcounts
An integer array (of length group size) that contains the number of elements to send to each task (significant only at **root**) (IN)

displs
An 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
The data type of the send buffer elements (handle, significant only at **root**) (IN)

recvbuf
The address of the receive buffer (choice) (OUT)

recvcount
The number of elements in the receive buffer (integer) (IN)

recvtype
The data type of the receive buffer elements (handle) (IN)

root
The rank of the sending task (integer) (IN)

comm
The communicator (handle) (IN)

request
The communication request (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

The **MPE_I** nonblocking collectives provided by IBM PE MPI were well-suited to the early implementation of MPI on AIX, which used signals for asynchronous progress. They are not well-suited to a threads-based implementation of MPI. The **MPE_I** nonblocking collectives will remain supported by the MPI library, but **the use of these nonstandard subroutines, in new or restructured applications, is now deprecated**.

The MPE prefix used with this subroutine 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 than the nonblocking versions.

Early versions of IBM PE MPI allowed the mixing of **MPE_I** (nonblocking) and **MPI_** (blocking) calls in a single collective operation. With IBM PE Version 4, there is a new shared memory based optimization for certain MPI collective operations, available in 64-bit executables and enabled by default. The shared memory optimization is not suitable for nonblocking collectives, so with this optimization enabled, affected collective operations that mix blocking and nonblocking calls will deadlock. **MP_EUIDEVELOP** mode has been enhanced to detect this mix and issue an error message. For further information on the shared memory optimization, refer to the description of **MP_SHARED_MEMORY** in *IBM Parallel Environment Runtime Edition: MPI Programming Guide*.

The IBM PE/MPI library has a limit of seven 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 any call defined by the MPI standard.

Applications using nonblocking collective calls often perform best when they run in interrupt mode.

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator are started in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

MPE_I routines have not been enhanced to use MPI-2 extensions. This routine, and all **MPE_I** nonblocking collectives are semantically equivalent to MPI-1.

Use of **MPE_I** nonblocking collective communications rules out setting environment variable **MP_SINGLE_THREAD**, or the command-line flag **-single_thread** to **yes**.

Errors

Invalid communicator

Invalid communicator type
must be intra-communicator

Invalid counts
count < 0

Invalid datatypes

Type not committed

Invalid root
root < 0 or **root** >= groupsize

Unequal message lengths

MPI_IN_PLACE not valid

MPI not initialized

MPI already finalized

Develop mode error (returned in the WAIT) if:

Inconsistent root

Match of blocking and non-blocking collectives

Related information

- MPI_SCATTERV

Chapter 3. MPI subroutines and functions

There are a number of MPI subroutines and functions that are available for parallel programming. Each of these subroutines and functions is defined in the MPI standard.

Codes that use these subroutines and functions can be ported to another MPI implementation through re-compilation of the source code. Review Chapter 1, "A sample MPI subroutine," on page 1 before proceeding to better understand how the subroutine and function descriptions are structured.

Do not match blocking (MPI) and nonblocking (MPE_I) collectives in the same 64-bit application. If you suspect a hang may be due to such mixing, turn on DEVELOP mode by setting the environment variable **MP_EUIDEVELOP** to **yes**, and look for error messages. If you receive a message about a mismatch, either run with **MP_SHARED_MEMORY** set to **no**, or change the application to no longer match blocking and nonblocking collectives.

For more information about matching blocking and nonblocking collectives in the same application, see the chapter *Programming considerations for user application in POE* of *IBM Parallel Environment Runtime Edition: MPI Programming Guide*.

MPI_ABORT, MPI_Abort

Forces all tasks of an MPI job to terminate.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::Comm::Abort(int errorcode);
```

Fortran synopsis

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

Description

This subroutine 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.

Parameters

comm
The communicator of the tasks to abort. (IN)

errorcode
The error code returned to the invoking environment. (IN)

IERROR
The Fortran return code. This is always the last argument.

Notes

MPI_ABORT causes *all* tasks to exit immediately.

In an environment that uses dynamic process management, there may be multiple disjoint worlds, each represented by its own MPI_COMM_WORLD. We say that world A and B are connected if there is any task in world A that is able to communicate with a task in world B. If some task of world A can communicate with some task of world B, and some task of world B can communicate with some task of world C, then worlds A and C are also connected, even though there is currently no task of A that can communicate directly with a task of C. If there is also a world D, in which no task can communicate with any task of A, B, or C, then D is not connected to ABC. In this sense, tasks of connected worlds are considered connected tasks whether they are able to communicate directly or not.

An MPI_Abort call is fatal to all connected tasks. It is transparent to tasks that are not connected (that is, tasks of worlds that are not connected).

Errors

MPI already finalized

MPI not initialized

MPI_ACCUMULATE, MPI_Accumulate

Accumulates, according to the specified reduction operation, the contents of the origin buffer to the specified target buffer.

C synopsis

```
#include <mpi.h>
int MPI_Accumulate (void *origin_addr, int origin_count,
                    MPI_Datatype origin_datatype, int target_rank,
                    MPI_Aint target_disp, int target_count,
                    MPI_Datatype target_datatype, MPI_Op op,
                    MPI_Win win);
```

C++ synopsis

```
#include mpi.h
void MPI::Win::Accumulate(const void* origin_addr, int origin_count,
     const MPI::Datatype& origin_datatype,
     int target_rank, MPI::Aint target_disp,
     int target_count, const MPI::Datatype& target_datatype,
     const MPI::Op& op) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_ACCUMULATE (CHOICE ORIGIN_ADDR, INTEGER ORIGIN_COUNT,
                INTEGER ORIGIN_DATATYPE, INTEGER TARGET_RANK,
                INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP, INTEGER TARGET_COUNT,
                INTEGER TARGET_DATATYPE, INTEGER OP,
                INTEGER WIN, INTEGER IERROR)
```

Description

This subroutine accumulates the contents of the origin buffer (as defined by *origin_addr*, *origin_count*, and *origin_datatype*) to the buffer specified by arguments *target_count* and *target_datatype*, at offset *target_disp*, in the target window specified by *target_rank* and *win*, using the operation *op*. MPI_ACCUMULATE is similar to MPI_PUT, except that data is combined into (rather than overwritten in) the target area.

This is a list of the predefined reduction operations that can be used. User-defined functions cannot be used. For example, if *op* is MPI_SUM, each element of the origin buffer is added to the corresponding element in the target, replacing the former value in the target.

Operation
Definition

MPI_BAND
Bitwise AND

MPI_BOR
Bitwise OR

MPI_BXOR
Bitwise XOR

MPI_LAND
Logical AND

MPI_LOR
Logical OR

MPI_LXOR
Logical XOR

MPI_MAX
Maximum value

MPI_MAXLOC
Maximum value and location

MPI_MIN
Minimum value

MPI_MINLOC
Minimum value and location

MPI_PROD
Product

MPI_REPLACE
f(a,b) = b (The current value in the target memory is replaced by the value supplied by the origin.)

MPI_SUM
Sum

Each data type argument must be a predefined data type or a derived data type, where all basic components are of the same predefined data type. Both data type arguments must be constructed from the same predefined data type. The operation *op* applies to elements of that predefined type. *target_datatype* must not specify overlapping entries, and the target buffer must fit in the target window.

A new predefined operation, MPI_REPLACE, corresponds to the associative function *f(a,b) = b*. That is, the current value in the target memory is replaced by the value supplied by the origin.

Concurrent MPI_ACCUMULATEs with MPI_REPLACE differs from concurrent MPI_PUT in that MPI_REPLACE guarantees each update will be atomic at element by element granularity.

Parameters

origin_addr
The initial address of the origin buffer (choice) (IN)

origin_count
The number of entries in origin buffer (nonnegative integer) (IN)

origin_datatype
The data type of each entry in the origin buffer (handle) (IN)

target_rank
The rank of the target (nonnegative integer) (IN)

target_disp
The displacement from the start of the window to the target buffer (nonnegative integer) (IN)

target_count
The number of entries in the target buffer (nonnegative integer) (IN)

target_datatype
The data type of each entry in the target buffer (handle) (IN)

op The reduction operation (handle) (IN)

win
The window object used for communication (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_ACCUMULATE does not require that data move from origin to target until some synchronization occurs. IBM PE MPI may try to combine multiple puts to a target within an epoch into a single data transfer. The user must not modify the source buffer or make any assumption about the contents of the destination buffer until after a synchronization operation has closed the epoch.

On some systems, there may be reasons to use special memory for one-sided communication buffers. MPI_ALLOC_MEM may be the preferred way to allocate buffers on these systems. With IBM PE MPI, there is no advantage to using MPI_ALLOC_MEM, but you can use it to improve the portability of your MPI code.

Errors

Invalid origin count (*count*)

Invalid origin datatype (*handle*)

Invalid target rank (*rank*)

Invalid target displacement (*value*)

Invalid target count (*count*)

Invalid target datatype (*handle*)

Invalid window handle (*handle*)

Target outside access group

Origin buffer too small (*size*)

Target buffer ends outside target window

Target buffer starts outside target window

RMA communication call outside access epoch

RMA communication call in progress

RMA synchronization call in progress

Origin datatype inappropriate for accumulate

Target datatype inappropriate for accumulate

Incompatible origin and target datatypes

Invalid reduction operation (*op*)

Related information

- MPI_GET
- MPI_PUT

MPI_ADD_ERROR_CLASS, MPI_Add_error_class

Creates a new error class and returns the value for it.

C synopsis

```
#include <mpi.h>
int MPI_Add_error_class(int *errorclass);
```

C++ synopsis

```
#include mpi.h
int MPI::Add_error_class();
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_ADD_ERROR_CLASS(INTEGER ERRORCLASS, INTEGER IERROR)
```

Description

This subroutine creates a new error class and returns the value for it so that the user classes do not conflict with any existing codes or classes. See subroutine "MPI_ERROR_CLASS, MPI_Error_class" on page 175 for a list of the predefined IBM PE MPI error classes.

Parameters

errorclass
The value for the new error class (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

Because a call to MPI_ADD_ERROR_CLASS is local, the same error class may not be returned on all tasks that make this call. Thus, it is not safe to assume that registering a new error class or code on a set of tasks at the same time will yield the same error class or code on all of the tasks. Only if all calls to create an error class or code occur in the same order on each task of MPI_COMM_WORLD will the values be globally consistent. The value of MPI_ERR_LASTCODE is not affected by new user-defined error codes and classes, as it is a constant value. Instead, a predefined attribute key MPI_LASTUSEDCODE is associated with MPI_COMM_WORLD. The attribute value corresponding to this key is the current maximum error class including the user-defined ones. This is a local value and may be different on different tasks. The value returned by this key is always greater than or equal to MPI_ERR_LASTCODE.

The value returned by the key MPI_LASTUSEDCODE will not change unless the user calls a function to explicitly add an error class or code. In a multi-threaded environment, the user must take extra care in assuming this value has not changed. Note that error codes and error classes are not necessarily dense. A user may not assume that each error class below MPI_LASTUSEDCODE is valid. An error is returned if the user tries to set the predefined MPI_LASTUSEDCODE using MPI_COMM_SET_ATTR.

Errors

Fatal errors:

MPI already finalized

MPI not initialized

Related information

- MPI_ADD_ERROR_CODE
- MPI_ADD_ERROR_STRING
- MPI_ERROR_CLASS
- MPI_ERROR_STRING

MPI_ADD_ERROR_CODE, MPI_Add_error_code

Creates a new error code associated with *errorclass* and returns its value in *errorcode*.

C synopsis

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

C++ synopsis

```
#include mpi.h
int MPI::Add_error_code(int errorclass);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_ADD_ERROR_CODE(INTEGER ERRORCLASS, INTEGER ERRORCODE, INTEGER IERROR)
```

Description

This subroutine creates a new error code associated with *errorclass* and returns its value in *errorcode* so that there are no conflicts with existing codes or classes.

Parameters

errorclass
The error class (integer) (IN)

errorcode
The new error code associated with *errorclass* (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

Because a call to MPI_ADD_ERROR_CLASS is local, the same error class may not be returned on all tasks that make this call. Thus, it is not safe to assume that registering a new error class or code on a set of tasks at the same time will yield the same error class or code on all of the tasks. Only if all calls to create an error class or code occur in the same order on each task of MPI_COMM_WORLD will the values be globally consistent. The value of MPI_ERR_LASTCODE is not affected by new user-defined error codes and classes, as it is a constant value. Instead, a predefined attribute key MPI_LASTUSEDCODE is associated with MPI_COMM_WORLD. The attribute value corresponding to this key is the current maximum error class including the user-defined ones. This is a local value and may be different on different tasks. The value returned by this key is always greater than or equal to MPI_ERR_LASTCODE.

The value returned by the key MPI_LASTUSEDCODE will not change unless the user calls a function to explicitly add an error class or code. In a multi-threaded environment, the user must take extra care in assuming this value has not changed. Note that error codes and error classes are not necessarily dense. A user may not assume that each error class below MPI_LASTUSEDCODE is valid. An error is returned if the user tries to set the predefined MPI_LASTUSEDCODE using MPI_COMM_SET_ATTR.

Errors

Fatal errors:

Invalid error class

MPI already finalized

MPI not initialized

Related information

- MPI_ADD_ERROR_CLASS
- MPI_ADD_ERROR_STRING
- MPI_ERROR_CLASS
- MPI_ERROR_STRING

MPI_ADD_ERROR_STRING, MPI_Add_error_string

Associates an error string with an error code or class.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::Add_error_string(int errorcode, const char* string);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_ADD_ERROR_STRING(INTEGER ERRORCODE, CHARACTER*(*) STRING, INTEGER IERROR)
```

Description

This subroutine associates an error string with an error code or class. The string length must be no more than the value specified by MPI_MAX_ERROR_STRING (128 characters).

Parameters

errorcode
The error code or class (integer) (IN)

string
The text corresponding to *errorcode* (string) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

The length of the string does not include the null terminator in C or C++. Trailing blanks are deleted in Fortran. Calling MPI_ADD_ERROR_STRING for an error code that already has a string will replace the old string with the new string. It is erroneous to call MPI_ADD_ERROR_STRING for an error code or class with a value that is less than or equal to the value specified by MPI_ERR_LASTCODE. In other words, error strings on IBM PE MPI-defined errors cannot be replaced. If MPI_ERROR_STRING is called when no string has been set, it returns a empty string (all spaces in Fortran or "" in C and C++).

Errors

Fatal errors:

Error string too long

Improper error message change

Invalid error code

MPI already finalized

MPI not initialized

Related information

- MPI_ADD_ERROR_CLASS
- MPI_ADD_ERROR_STRING
- MPI_ERROR_CLASS
- MPI_ERROR_STRING

MPI_ADDRESS, MPI_Address

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' or USE MPI
MPI_ADDRESS(CHOICE LOCATION,INTEGER ADDRESS,INTEGER IERROR)
```

Description

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

Parameters

location
The location in caller memory (choice) (IN)

address
The address of location (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_GET_ADDRESS supersedes MPI_ADDRESS.

The Fortran MPI_ADDRESS binding is not valid for 64-bit Fortran programs because it is not possible to predict when an address will fit in 32 bits.

MPI_ADDRESS is equivalent to **address= (MPI_Aint) location** in C, but this subroutine is portable to processors with less straightforward addressing.

Errors

MPI not initialized

MPI already finalized

Related information

- MPI_TYPE_HINDEXED
- MPI_TYPE_INDEXED
- MPI_TYPE_STRUCT

MPI_ALLGATHER, MPI_Allgather

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Comm::Allgather(const void* sendbuf, int sendcount,
     const MPI::Datatype& sendtype, void* recvbuf,
     int recvcount, const MPI::Datatype& recvtype)
     const;
```

Fortran synopsis

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

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.

The *in place* option for intra-communicators is specified by passing the value MPI_IN_PLACE to *sendbuf* at all tasks. The *sendcount* and *sendtype* arguments are ignored. The input data of each task is assumed to be in the area where that task would receive its own contribution to the receive buffer. Specifically, the outcome of a call to MPI_ALLGATHER in the *in place* case is as if all tasks issued *n* calls to:

```
MPI_GATHER(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, recvbuf, recvcount, recvtype,
    root, comm)
```

for: *root* = 0 to *n-1*.

If *comm* is an inter-communicator, each task in group A contributes a data item. These items are concatenated and the result is stored at each task in group B. Conversely, the concatenation of the contributions of the tasks in group B is stored at each task in group A. The send buffer arguments in group A must be consistent with the receive buffer arguments in group B, and vice versa.

MPI_IN_PLACE is not supported for inter-communicators.

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator occur in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

Parameters

sendbuf
The starting address of the send buffer (choice) (IN)

sendcount
The number of elements in the send buffer (integer) (IN)

sendtype
The data type of the send buffer elements (handle) (IN)

recvbuf
The address of the receive buffer (choice) (OUT)

recvcount
The number of elements received from any task (integer) (IN)

recvtype
The data type of the receive buffer elements (handle) (IN)

comm
The communicator (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Errors

Fatal errors:

Invalid communicator

Invalid counts
count < **0**

Invalid datatypes

Type not committed

Unequal message lengths

Invalid use of MPI_IN_PLACE

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

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Comm::Allgatherv(const void* sendbuf, int sendcount,
      const MPI::Datatype& sendtype, void* recvbuf,
      const int recvcounts[], const int displs[],
      const MPI::Datatype& recvtype) const;
```

Fortran synopsis

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

Description

This subroutine 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.

The *in place* option for intra-communicators is specified by passing the value MPI_IN_PLACE to *sendbuf* at all tasks. The *sendcount* and *sendtype* arguments are ignored. The input data of each task is assumed to be in the area where that task would receive its own contribution to the receive buffer. Specifically, the outcome of a call to MPI_ALLGATHERV in the *in place* case is as if all tasks issued *n* calls to:

```
MPI_GATHERV(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, recvbuf, recvcount, recvtype,
     root, comm)
```

for: *root* = 0 to *n- 1*.

If *comm* is an inter-communicator, each task in group A contributes a data item. These items are concatenated and the result is stored at each task in group B. Conversely, the concatenation of the contributions of the tasks in group B is stored at each task in group A. The send buffer arguments in group A must be consistent with the receive buffer arguments in group B, and vice versa.

MPI_IN_PLACE is not supported for inter-communicators.

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

IBM Parallel Environment Runtime Edition: MPI Programming Guide for more information on programming with MPI in a threads environment.

Parameters

sendbuf
The starting address of the send buffer (choice) (IN)

sendcount
The number of elements in the send buffer (integer) (IN)

sendtype
The data type of the send buffer elements (handle) (IN)

recvbuf
The address of the receive buffer (choice) (OUT)

recvcounts
An integer array (of length *groupsize*) that contains the number of elements received from each task (IN)

displs
An integer array (of length *groupsize*). Entry *i* specifies the displacement (relative to *recvbuf*) at which to place the incoming data from task *i* (IN)

recvtype
The data type of the receive buffer elements (handle) (IN)

comm
The communicator (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Errors

Fatal errors:

Invalid communicator

Invalid counts
count < **0**

Invalid datatypes

Type not committed

Unequal message lengths

Invalid use of MPI_IN_PLACE

MPI not initialized

MPI already finalized

Develop mode error if:

None

Related information

- MPE_IALLGATHERV
- MPI_ALLGATHER

MPI_ALLOC_MEM, MPI_Alloc_mem

Allocates storage and returns a pointer to it.

C synopsis

```
#include <mpi.h>
int MPI_Alloc_mem (MPI_Aint size, MPI_Info info, void *baseptr);
```

C++ synopsis

```
#include mpi.h
void* MPI::Alloc_mem(MPI::Aint size, const MPI::Info& info);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_ALLOC_MEM(INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, INTEGER INFO,
              INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR, INTEGER IERROR)
```

Description

This subroutine allocates at least *size* bytes of storage and returns a pointer to it in the *baseptr* argument. The block of allocated storage is aligned so that it may be used for any type of data.

The *info* argument may be used in some implementations to provide directives that control the desired location of the allocated memory. Such a directive does not affect the semantics of the call. Valid *info* values are implementation-dependent. IBM PE MPI does not recognize any hints for MPI_ALLOC_MEM. A null directive value of *info* = **MPI_INFO_NULL** is always valid.

Parameters

size
The size of the memory segment in bytes (nonnegative integer) (IN)

info
The Info argument (handle) (IN)

baseptr
The pointer to the beginning of the memory segment allocated (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

If the requested amount of memory is not available, the error handler associated with MPI_COMM_WORLD is invoked. By default, this is MPI_ERRORS_ARE_FATAL.

Errors

Fatal errors:

Out of memory (MPI_ERR_NO_MEM)

Invalid info (MPI_ERR_INFO)

MPI not initialized (MPI_ERR_OTHER)

MPI already finalized (MPI_ERR_OTHER)

Related information

- MPI_FREE_MEM
- MPI_WIN_CREATE

MPI_ALLREDUCE, MPI_Allreduce

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Comm::Allreduce(const void* sendbuf, void* recvbuf, int count,
     const MPI::Datatype& datatype, const MPI::Op& op)
     const;
```

Fortran synopsis

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

Description

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

The *in place* option for intra-communicators is specified by passing the value MPI_IN_PLACE to the argument *sendbuf* at the root. In this case, the input data is taken at each task from the receive buffer, where it will be replaced by the output data.

If *comm* is an inter-communicator, the result of the reduction of the data provided by tasks in group A is stored at each task in group B, and vice versa. Both groups should provide the same count value.

MPI_IN_PLACE is not supported for inter-communicators.

The parameter *op* may be a predefined reduction operation or a user-defined function, created using MPI_OP_CREATE. This is a list of predefined reduction operations:

Operation
Definition

MPI_BAND
Bitwise AND

MPI_BOR
Bitwise OR

MPI_BXOR
Bitwise XOR

MPI_LAND
Logical AND

MPI_LOR
Logical OR

MPI_LXOR
Logical XOR

MPI_MAX
Maximum value

MPI_MAXLOC
Maximum value and location

MPI_MIN
Minimum value

MPI_MINLOC
Minimum value and location

MPI_PROD
Product

MPI_SUM
Sum

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator occur in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

Parameters

sendbuf
The starting address of the send buffer (choice) (IN)

recvbuf
The starting address of the receive buffer (choice) (OUT)

count
The number of elements in the send buffer (integer) (IN)

datatype
The data type of elements in the send buffer (handle) (IN)

op The reduction operation (handle) (IN)

comm
The communicator (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for information about reduction functions.

The MPI standard urges MPI implementations to use the same evaluation order for reductions every time, even if this negatively affects performance. IBM PE MPI adjusts its reduce algorithms for the optimal performance on a given task distribution. The MPI standard suggests, but does not mandate, this sacrifice of performance. IBM PE MPI maintains a balance between performance and the MPI standard's recommendation. IBM PE MPI does not promise that any two runs with the same tack count will give the same answer, in the least significant bits, for

floating point reductions. Changes to evaluation order may produce different rounding effects. However, IBM PE MPI does promise that two calls to MPI_REDUCE (or MPI_ALLREDUCE) on the same communicator with the same inputs, or two runs that use the same task count and the same distribution across nodes, will always give identical results.

In the 64-bit library, this function uses a shared memory optimization among the tasks on a node. This optimization is discussed in the chapter *Using shared memory* of *IBM Parallel Environment Runtime Edition: MPI Programming Guide*, and is enabled by default. This optimization is not available to 32-bit programs.

Errors

Fatal errors:

Invalid count
count < **0**

Invalid datatype

Type not committed

Invalid op

Invalid communicator

Unequal message lengths

Invalid use of MPI_IN_PLACE

MPI not initialized

MPI already finalized

Develop mode error if:

Inconsistent op

Inconsistent datatype

Inconsistent message length

Related information

- MPE_IALLREDUCE
- MPI_OP_CREATE
- MPI_REDUCE
- MPI_REDUCE_SCATTER

MPI_ALLTOALL, MPI_Alltoall

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Comm::Alltoall(const void* sendbuf, int sendcount,
    const MPI::Datatype& sendtype, void* recvbuf,
    int recvcount, const MPI::Datatype& recvtype)
    const;
```

Fortran synopsis

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

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.

The *in place* option for intracommunicators is specified by passing **MPI_IN_PLACE** to the argument *sendbuf* at all processes. In such a case, *sendcount* and *sendtype* are ignored. The data to be sent is taken from the *recvbuf* and replaced by the received data. Data sent and received must have the same type map as specified by *recvcount* and *recvtype*.

For large **MPI_ALLTOALL** instances, allocating both send and receive buffers may consume too much memory. The *in place* option effectively halves the application memory consumption and is useful in situations where the data to be sent will not be used by the sending process after the **MPI_ALLTOALL** exchange.

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

When MPI_ALLTOALL is run on an inter-communicator, the number of data items sent from tasks in group A to tasks in group B does not need to be equal to the number of items sent in the reverse direction. In particular, you can have unidirectional communication by specifying *sendcount* = **0** in the reverse direction.

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator occur in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

Parameters

sendbuf
The starting address of the send buffer (choice) (IN)

sendcount
The number of elements sent to each task (integer) (IN)

sendtype
The data type of the send buffer elements (handle) (IN)

recvbuf
The address of the receive buffer (choice) (OUT)

recvcount
The number of elements received from any task (integer) (IN)

recvtype
The data type of the receive buffer elements (handle) (IN)

comm
The communicator (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Errors

Fatal errors:

Unequal lengths

Invalid counts
count < **0**

Invalid datatypes

Type not committed

Invalid communicator

Unequal message lengths

Invalid use of MPI_IN_PLACE

MPI not initialized

MPI already finalized

Develop mode error if:

Inconsistent message lengths

Related information

- MPE_IALLTOALL
- MPI_ALLTOALLV

MPI_ALLTOALLV, MPI_Alltoallv

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Comm::Alltoallv(const void* sendbuf, const int sendcounts[],
     const int sdispls[], const MPI::Datatype& sendtype,
     void* recvbuf, const int recvcounts[],
     const int rdispls[], const MPI::Datatype& recvtype)
           const;
```

Fortran synopsis

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

Description

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

This subroutine 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 *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.

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

Specifying the *in place* option (which must be given on all processes) implies that the same amount and type of data is sent and received between any two processes

in the group of the communicator. Different pairs of processes can exchange different amounts of data. Users must ensure that *recvcounts* [j] and *recvtype* on process i match *recvcounts*[i] and *recvtype* on process j. This symmetric exchange can be useful in applications where the data to be sent will not be used by the sending process after the **MPI_ALLTOALLV** exchange.

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

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator occur in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

Parameters

sendbuf
The starting address of the send buffer (choice) (IN)

sendcounts
An integer array (of length *groupsize*) specifying the number of elements to send to each task (IN)

sdispls
An integer array (of length *groupsize*). Entry *j* specifies the displacement relative to *sendbuf* from which to take the outgoing data destined for task *j*. (IN)

sendtype
The data type of the send buffer elements (handle) (IN)

recvbuf
The address of the receive buffer (choice) (OUT)

recvcounts
An integer array (of length *groupsize*) specifying the number of elements to be received from each task (IN)

rdispls
An integer array (of length *groupsize*). Entry *i* specifies the displacement relative to *recvbuf* at which to place the incoming data from task *i*. (IN)

recvtype
The data type of the receive buffer elements (handle) (IN)

comm
The communicator (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Errors

Fatal errors:

Invalid counts
count < **0**

Invalid datatypes

Type not committed

Invalid communicator

A send and receive have unequal message lengths

Invalid use of MPI_IN_PLACE

MPI not initialized

MPI already finalized

Related information

- MPE_IALLTOALLV
- MPI_ALLTOALL

MPI_ALLTOALLW, MPI_Alltoallw

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

C synopsis

```
#include <mpi.h>
int MPI_Alltoallw(void* sendbuf, int sendcounts[], int sdispls[],
    MPI_Datatype sendtypes[], void *recvbuf, int recvcounts[],
    int rdispls[], MPI_Datatype recvtypes[], MPI_Comm comm);
```

C++ synopsis

```
#include mpi.h
void MPI::Comm::Alltoallw(const void *sendbuf, const int sendcounts[],
     const int sdispls[], const MPI::Datatype sendtypes[],
     void *recvbuf, const int recvcounts[], const int rdispls[],
     const MPI::Datatype recvtypes[]) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_ALLTOALLW(CHOICE SENDBUF(*), INTEGER SENDCOUNTS(*), INTEGER SDISPLS(*),
       INTEGER SENDTYPES(*), CHOICE RECVBUF, INTEGER RECVCOUNTS(*),
       INTEGER RDISPLS(*), INTEGER RECVTYPES(*), INTEGER COMM, INTEGER IERROR)
```

Description

This subroutine is an extension of MPI_ALLTOALLV. It allows separate specification of count, displacement and data type. In addition, to allow maximum flexibility, the displacement of blocks within the send and receive buffers is specified in bytes.

The *j*th block sent from task *i* is received by task *j* and is placed in the *i*th block of *recvbuf*. These blocks need not all have the same size.

The type signature associated with *sendcounts[j], sendtypes[j]* at task *i* must be equal to the type signature associated with *recvcounts[i], recvtypes[i]* 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.

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

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

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator occur in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more

information on programming with MPI in a threads environment.

Parameters

sendbuf
The starting address of the send buffer (choice) (IN)

sendcounts
An integer array (of length *groupsize*) specifying the number of elements to send to each task (IN)

sdispls
An integer array (of length *groupsize*). Entry *j* specifies the displacement in bytes (relative to *sendbuf*) from which to take the outgoing data destined for task *j*. (IN)

sendtypes
The array of data types (of length *groupsize*). Entry *j* specifies the type of data to send to task *j*. (handle) (IN)

recvbuf
The address of the receive buffer (choice) (OUT)

recvcounts
An integer array (of length *groupsize*) specifying the number of elements to be received from each task (IN)

rdispls
An integer array (of length *groupsize*). Entry *i* specifies the displacement in bytes (relative to *recvbuf*) at which to place the incoming data from task *i*. (IN)

recvtypes
The array of data types (of length *groupsize*). Entry *i* specifies the type of data received from task *i*. (handle) (IN)

comm
The communicator (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

In the bindings for this subroutine, the send displacement and receive displacements are arrays of integers. This may limit the usability of this subroutine in certain 64-bit applications. It is possible that the MPI Forum will define a replacement for MPI_ALLTOALLW and deprecate this binding. The replacement subroutine will use arrays of **address_size** integers. The MPI_ALLTOALLW subroutine with the present binding will remain available.

Errors

Fatal errors:

Invalid counts
count < **0**

Invalid datatypes

Type not committed

Invalid communicator

A send and receive have unequal message lengths

Invalid use of MPI_IN_PLACE

MPI not initialized

MPI already finalized

Related information

- MPI_ALLTOALLV

MPI_ATTR_DELETE, MPI_Attr_delete

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' or USE MPI
MPI_ATTR_DELETE(INTEGER COMM,INTEGER KEYVAL,INTEGER IERROR)
```

Description

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

Parameters

comm
The communicator that the attribute is attached (handle) (IN)

keyval
The key value of the deleted attribute (integer) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_COMM_DELETE_ATTR supersedes MPI_ATTR_DELETE.

MPI_ATTR_DELETE does not inter-operate with MPI_COMM_DELETE_ATTR. The Fortran bindings for MPI-1 caching functions presume that an attribute is an INTEGER. The MPI-2 caching bindings use INTEGER (KIND=MPI_ADDRESS_KIND). In an MPI implementation that uses 64-bit addresses and 32-bit INTEGERS, the two formats would be incompatible.

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

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' or USE MPI
MPI_ATTR_GET(INTEGER COMM,INTEGER KEYVAL,INTEGER ATTRIBUTE_VAL,
      LOGICAL FLAG,INTEGER IERROR)
```

Description

This subroutine 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**.

Parameters

comm
The communicator to which attribute is attached (handle) (IN)

keyval
The key value (integer) (IN)

attribute_val
The attribute value unless *flag* = **false** (OUT)

flag
Set to **true** if an attribute value was extracted and *false* if no attribute is associated with the key. (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_COMM_GET_ATTR supersedes MPI_ATTR_GET.

MPI_ATTR_GET does not interoperate with MPI_COMM_SET_ATTR. The Fortran bindings for MPI-1 caching functions presume that an attribute is an INTEGER. The MPI-2 caching bindings use INTEGER (KIND=MPI_ADDRESS_KIND). In an MPI implementation that uses 64-bit addresses and 32-bit INTEGERS, the two formats would be incompatible. You should not mix the MPI-1 and MPI-2 functions when managing attributes on communicators in Fortran.

The implementation of MPI_ATTR_GET and MPI_ATTR_PUT 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 that 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 make sure 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 that 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 can choose to encode all attribute information in this integer or maintain some kind of database in which the integer can index. Either of these approaches will port to other MPI implementations.

Many of the Fortran compilers include an additional feature that allows some of the same functions a C programmer would use. These compilers support the POINTER type, often referred to as a *Cray pointer*. XL Fortran is one of the compilers that supports the POINTER type. For more information, see *IBM XL Fortran Compiler Reference*.

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

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' or USE MPI
MPI_ATTR_PUT(INTEGER COMM,INTEGER KEYVAL,INTEGER ATTRIBUTE_VAL,
      INTEGER IERROR)
```

Description

This subroutine 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.

Parameters

comm
The communicator to which attribute will be attached (handle) (IN)

keyval
The key value as returned by MPI_KEYVAL_CREATE (integer) (IN)

attribute_val
The attribute value (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_COMM_SET_ATTR supersedes MPI_ATTR_PUT.

MPI_ATTR_PUT does not interoperate with MPI_COMM_GET_ATTR. The Fortran bindings for MPI-1 caching functions presume that an attribute is an INTEGER. The MPI-2 caching bindings use INTEGER (KIND=MPI_ADDRESS_KIND). In an MPI implementation that uses 64-bit addresses and 32-bit INTEGERS, the two formats would be incompatible. You should not mix the MPI-1 and MPI-2 functions when managing attributes on communicators in Fortran.

The implementation of 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 that 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 make sure 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 that 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 can choose to encode all attribute information in this integer or maintain some kind of database in which the integer can index. Either of these approaches will port to other MPI implementations.

Many of the Fortran compilers include an additional feature that allows some of the same functions a C programmer would use. These compilers support the POINTER type, often referred to as a *Cray pointer*. XL Fortran is one of the compilers that supports the POINTER type. For more information, see *IBM XL Fortran Compiler Reference*.

Errors

A delete_fn did not return MPI_SUCCESS

Invalid communicator

Invalid keyval
keyval is undefined.

Predefined keyval
You cannot modify predefined attributes.

MPI not initialized

MPI already finalized

Related information

- MPI_COMM_COPY_ATTR_FUNCTION
- MPI_COMM_CREATE_KEYVAL
- MPI_COMM_DELETE_ATTR
- MPI_COMM_DELETE_ATTR_FUNCTION
- MPI_COMM_GET_ATTR

MPI_BARRIER, MPI_Barrier

Blocks each task until all tasks have called it.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::Comm::Barrier() const;
```

Fortran synopsis

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

Description

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

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator occur in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

Parameters

comm
A communicator (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

comm can be an inter-communicator or an intra-communicator. If *comm* is an inter-communicator, the barrier is performed across all tasks in the inter-communicator. In this case, all tasks in the local group of the inter-communicator can exit the barrier when all of the tasks in the remote group have entered the barrier.

In the 64-bit library, this function uses a shared memory optimization among the tasks on a node. This optimization is discussed in the chapter *Using shared memory* of *IBM Parallel Environment Runtime Edition: MPI Programming Guide*, and is enabled by default. This optimization is not available to 32-bit programs.

Errors

Fatal errors:

Invalid communicator

MPI not initialized

MPI already finalized

Related information

- MPE_IBARRIER

MPI_BCAST, MPI_Bcast

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Comm::Bcast(void* buffer, int count, const MPI::Datatype& datatype,
        int root) const;
```

Fortran synopsis

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

Description

This subroutine broadcasts a message from *root* to all tasks in *comm*. The contents of *root*'s communication buffer are 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.

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

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator occur in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

Parameters

buffer
The starting address of the buffer (choice) (INOUT)

count
The number of elements in the buffer (integer) (IN)

datatype
The data type of the buffer elements (handle) (IN)

root
The rank of the root task (integer) (IN)

comm
The communicator (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

In the 64-bit library, this function uses a shared memory optimization among the tasks on a node. This optimization is discussed in the chapter *Using shared memory* of *IBM Parallel Environment Runtime Edition: MPI Programming Guide*, and is enabled by default. This optimization is not available to 32-bit programs.

Errors

Fatal errors:

Invalid communicator

Invalid count
count < **0**

Invalid datatype

Type not committed

Invalid root

For an intra-communicator: *root* < **0** or *root* >= *groupsize*

For an inter-communicator: *root* < **0** and is neither MPI_ROOT nor MPI_PROC_NULL, or *root* >= *groupsize* of the remote group

Unequal message lengths

Invalid use of MPI_IN_PLACE

MPI not initialized

MPI already finalized

Develop mode error if:

Inconsistent root

Inconsistent message length

Related information

- MPE_IBCAST

MPI_BSEND, MPI_Bsend

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Comm::Bsend(const void* buf, int count, const MPI::Datatype& datatype,
        int dest, int tag) const;
```

Fortran synopsis

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

Description

This subroutine is a blocking buffered mode send operation. It 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.

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.

Parameters

buf
The initial address of the send buffer (choice) (IN)

count
The number of elements in the send buffer (integer) (IN)

datatype
The data type of each send buffer element (handle) (IN)

dest
The rank of destination (integer) (IN)

tag
The message tag (positive integer) (IN)

comm
The communicator (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

Make sure you have enough buffer space available. An error occurs if the message must be buffered and there is there is not enough buffer space. The amount of buffer space needed to be safe depends on the expected peak of pending messages. The sum of the sizes of all of the pending messages at that point plus (**MPI_BSEND_OVERHEAD***number_of_messages*) should be sufficient.

Avoid using **MPI_BSEND** if possible. It adds overhead because it requires an extra memory-to-memory copy of the outgoing data. If **MPI_BSEND** is used, the associated receive operations may perform better with **MPI_CSS_INTERRUPT** enabled.

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_BUFFER_ATTACH
- MPI_BUFFER_DETACH
- MPI_IBSEND
- MPI_SEND

MPI_BSEND_INIT, MPI_Bsend_init

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);
```

C++ synopsis

```
#include mpi.h
MPI::Prequest MPI::Comm::Bsend_init(const void* buf, int count,
        const MPI::Datatype& datatype,
        int dest, int tag) const;
```

Fortran synopsis

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

Description

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

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

Parameters

buf
The initial address of the send buffer (choice) (IN)

count
The number of elements to be sent (integer) (IN)

datatype
The type of each element (handle) (IN)

dest
The rank of the destination task (integer) (IN)

tag
The message tag (positive integer) (IN)

comm
The communicator (handle) (IN)

request
The communication request (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

Make sure you have enough buffer space available. An error occurs if the message must be buffered and there is there is not enough buffer space. The amount of buffer space needed to be safe depends on the expected peak of pending messages.

The sum of the sizes of all of the pending messages at that point plus (**MPI_BSEND_INIT_OVERHEAD***number_of_messages*) should be sufficient.

Avoid using **MPI_BSEND_INIT** if possible. It adds overhead because it requires an extra memory-to-memory copy of the outgoing data. If **MPI_BSEND_INIT** is used, the associated receive operations may perform better with **MPI_CSS_INTERRUPT** enabled.

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_IBSEND
- MPI_START

MPI_BUFFER_ATTACH, MPI_Buffer_attach

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Attach_buffer(void* buffer, int size);
```

Fortran synopsis

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

Description

This subroutine 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.

Parameters

buffer
The initial buffer address (choice) (IN)

size
The buffer size in bytes (integer) (IN)

IERROR
The Fortran return code. It is always the last argument.

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_BSEND
- MPI_BUFFER_DETACH
- MPI_IBSEND

MPI_BUFFER_DETACH, MPI_Buffer_detach

Detaches the current buffer.

C synopsis

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

C++ synopsis

```
#include mpi.h
int MPI::Detach_buffer(void*& buffer);
```

Fortran synopsis

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

Description

This subroutine 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 implicit 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.

Parameters

buffer
The initial buffer address (choice) (OUT)

size
The buffer size in bytes (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

It is important to detach an attached buffer *before* it is deallocated. Otherwise, unpredictable errors are likely.

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 temporary buffer as the *buffer* argument.

If a buffer was attached, IBM PE MPI returns the address of the freed buffer in the first word of the *buffer* argument. If the *size* being returned is 0 to 4 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 temporary buffer 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_BSEND
- MPI_BUFFER_ATTACH
- MPI_IBSEND

MPI_CANCEL, MPI_Cancel

Marks a nonblocking request for cancellation.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::Request::Cancel(void) const;
```

Fortran synopsis

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

Description

This subroutine 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 communications operation succeeds, but not both. If a send is marked for cancellation, 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.

Parameters

request
A communication request (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

Nonblocking collective communication requests cannot be cancelled.
MPI_CANCEL may be called on nonblocking 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 Grequest cancel function returned an error

MPI not initialized

MPI already finalized

Related information

- MPI_TEST_CANCELLED
- MPI_WAIT

MPI_CART_COORDS, MPI_Cart_coords

Translates task rank in a communicator into Cartesian task coordinates.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::Cartcomm::Get_coords(int rank, int maxdims,
          int coords[]) const;
```

Fortran synopsis

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

Description

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

Parameters

comm
A communicator with Cartesian topology (handle) (IN)

rank
The rank of a task within group *comm* (integer) (IN)

maxdims
The length of array **coords** in the calling program (integer) (IN)

coords
An integer array specifying the Cartesian coordinates of a task. (OUT)

IERROR
The Fortran return code. It is always the last argument.

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_CREATE
- MPI_CART_RANK

MPI_CART_CREATE, MPI_Cart_create

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);
```

C++ synopsis

```
#include mpi.h
MPI::Cartcomm MPI::Intracomm::Create_cart(int ndims, const int dims[],
       const bool periods[], bool reorder) const;
```

Fortran synopsis

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

Description

This subroutine creates a new communicator that contains 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 are required by the grid, some tasks are returned and *comm_cart* = MPI_COMM_NULL. *comm_old* must be an intra-communicator.

Parameters

comm_old
The input communicator (handle) (IN)

ndims
The number of Cartesian dimensions in the grid (integer) (IN)

dims
An integer array of size *ndims* specifying the number of tasks in each dimension (IN)

periods
A logical array of size *ndims* specifying if the grid is periodic or not in each dimension (IN)

reorder
Set to **true**, ranking may be reordered. Set to **false**, rank in *comm_cart* must be the same as in *comm_old*. (logical) (IN)

comm_cart
A communicator with new Cartesian topology (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

Early versions of MPI on AIX and most other MPI implementations (on either AIX or Linux) that are available today ignore *reorder*, as the MPI standard allows.

If you have a program that works with *reorder* = **false** and fails with *reorder* = **true**, examine your code for communication on *comm_cart* using ranks from *comm_old*.

Errors

MPI not initialized

Conflicting collective operations on communicator

MPI already finalized

Invalid communicator

Invalid communicator type
must be intra-communicator

Invalid ndims
ndims < **0** or *ndims* > *groupsize*

Invalid dimension

Related information

- MPI_CART_SUB
- MPI_GRAPH_CREATE

MPI_CART_GET, MPI_Cart_get

Retrieves Cartesian topology information from a communicator.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::Cartcomm::Get_topo(int maxdims, int dims[],
        bool periods[], int coords[]) const;
```

Fortran synopsis

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

Description

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

Parameters

comm
A communicator with Cartesian topology (handle) (IN)

maxdims
The length of *dims, periods,* and *coords* in the calling program (integer) (IN)

dims
The number of tasks for each Cartesian dimension (array of integer) (OUT)

periods
A logical array specifying if each Cartesian dimension is periodic or not. (OUT)

coords
The coordinates of the calling task in the Cartesian structure (array of integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

MPI not initialized

MPI already finalized

Invalid communicator

No topology

Invalid topology type
Type must be Cartesian.

Invalid array size
maxdims < 0

Related information

- MPI_CART_CREATE
- MPI_CARTDIM_GET

MPI_CART_MAP, MPI_Cart_map

Computes placement of tasks on the physical processor.

C synopsis

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

C++ synopsis

```
#include mpi.h
int MPI::Cartcomm::Map(int ndims, const int dims[],
         const bool periods[]) const;
```

Fortran synopsis

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

Description

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

Parameters

comm
The input communicator (handle) (IN)

ndims
The number of dimensions of the Cartesian structure (integer) (IN)

dims
An integer array of size *ndims* specifying the number of tasks in each coordinate direction (IN)

periods
A logical array of size *ndims* specifying the periodicity in each coordinate direction (IN)

newrank
The reordered rank or MPI_UNDEFINED if the calling task does not belong to the grid (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

The rank determined by MPI_CART_MAP depends on the distribution of task per node. The value may or may not match rank in MPI_COMM_WORLD.

Errors

MPI not initialized

MPI already finalized

Invalid communicator

Invalid communicator type
Communication type must be intra-communicator.

Invalid ndims
ndims < 1

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

Translates task coordinates into a task rank.

C synopsis

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

C++ synopsis

```
#include mpi.h
int MPI::Cartcomm::Get_cart_rank(const int coords[]) const;
```

Fortran synopsis

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

Description

This subroutine 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 automatically shifted back to the interval **0** <= *coords(i)* < *dims(i)*. Out-of-range coordinates are erroneous for nonperiodic dimensions.

Parameters

comm
A communicator with Cartesian topology (handle) (IN)

coords
An integer array of size *ndims* specifying the Cartesian coordinates of a task (IN)

rank
An integer specifying the rank of specified task (OUT)

IERROR
The Fortran return code. It is always the last argument.

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
Type must be Cartesian.

Invalid coordinates
Refer to the **Description** section of this manual page.

Related information

- MPI_CART_COORDS
- MPI_CART_CREATE

MPI_CART_SHIFT, MPI_Cart_shift

Returns shifted source and destination ranks for a task.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::Cartcomm::Shift(int direction, int disp, int &rank_source,
     int &rank_dest) const;
```

Fortran synopsis

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

Description

This subroutine 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 nonperiodic, off-end shifts result in the value MPI_PROC_NULL being returned for *rank_source* or *rank_dest* or both.

Parameters

comm
A communicator with Cartesian topology (handle) (IN)

direction
The coordinate dimension of shift (integer) (IN)

disp
The displacement (> 0 = upward shift, < 0 = downward shift) (integer) (IN)

rank_source
The rank of the source task (integer) (OUT)

rank_dest
The rank of the destination task (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

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
Type must be Cartesian.

No topology

Related information

- MPI_CART_COORDS
- MPI_CART_CREATE
- MPI_CART_RANK

MPI_CART_SUB, MPI_Cart_sub

Partitions a Cartesian communicator into lower-dimensional subgroups.

C synopsis

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

C++ synopsis

```
#include mpi.h
MPI::Cartcomm MPI::Cartcomm::Sub(const bool remain_dims[]) const;
```

Fortran synopsis

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

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
- 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, suppose MPI_CART_CREATE (..., *comm*) defined a $2 \times 3 \times 4$ grid and *remain_dims* = (true, false, true). 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), the call to:

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

creates six nonoverlapping communicators, each with four tasks in a one-dimensional Cartesian topology.

Parameters

comm
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
The communicator containing the subgrid that includes the calling task (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

MPI not initialized

MPI already finalized

Invalid communicator

Invalid topology
Type must be Cartesian.

No topology

Related information

- MPI_CART_CREATE
- MPI_COMM_SPLIT

MPI_CARTDIM_GET, MPI_Cartdim_get

Retrieves the number of Cartesian dimensions from a communicator.

C synopsis

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

C++ synopsis

```
#include mpi.h
int MPI::Cartcomm::Get_dim() const;
```

Fortran synopsis

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

Description

This subroutine retrieves the number of dimensions in a Cartesian topology.

Parameters

comm
A communicator with Cartesian topology (handle) (IN)

ndims
An integer specifying the number of dimensions of the Cartesian topology (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid communicator

No topology

Invalid topology type
Type must be Cartesian.

MPI not initialized

MPI already finalized

Related information

- MPI_CART_CREATE
- MPI_CART_GET

MPI_CLOSE_PORT, MPI_Close_port

Releases the network address.

C synopsis

```
#include <mpi.h>
int MPI_Close_port(char *port_name);
```

C++ synopsis

```
#include <mpi.h>
void MPI::Close_port(const char* port_name);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_CLOSE_PORT(PORT_NAME, IERROR)
CHARACTER*(*) PORT_NAME
INTEGER IERROR
```

Description

This subroutine releases the network address represented by *port_name*.

Parameters

port_name
A port (string) (IN)

Related information

- **MPI_OPEN_PORT**

MPI_COMM_ACCEPT, MPI_Comm_accept

Establishes communication with a client.

C synopsis

```
#include <mpi.h>
int MPI_Comm_accept(char *port_name, MPI_Info info, int root,
MPI_Comm comm, MPI_Comm *newcomm);
```

C++ synopsis

```
#include <mpi.h>
int MPI::Intercomm MPI::Intracomm::Accept(const char* port_name,
const MPI::Info& info, int root) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
CHARACTER*(*) PORT_NAME
INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
```

Description

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

The *port_name* argument must have been established through a call to **MPI_OPEN_PORT**.

Parameters

port_name
The port name (string, used only on **root**) (IN)

info
An info is an object containing {key,value} pairs. IBM PE MPI **MPI_COMM_ACCEPT** does not recognize any info keys. **MPI_INFO_NULL** is always valid (IN)

root
The rank in **comm** of the root node (integer) (IN)

comm
The intracommunicator over which the call is collective (handle) (IN)

newcomm
The intercommunicator with client as remote group (handle) (OUT)

Errors

Named port does not exist.

Invalid rank (remote leader that called MPI_COMM_CONNECT)

Related information

- **MPI_COMM_CONNECT**

MPI_Comm_c2f

Translates a C communicator handle into a Fortran handle to the same communicator.

C synopsis

```
#include <mpi.h>
MPI_Fint MPI_Comm_c2f(MPI_Comm comm);
```

Description

This function does not have C++ or Fortran bindings. MPI_Comm_c2f translates a C communicator handle into a Fortran handle to the same communicator. This function maps a null handle into a null handle and a handle that is not valid into a handle that is not valid. The converted handle is returned as the function's value. There is no error detection or return code.

Parameters

comm
A communicator (handle) (IN)

Related information

- MPI_Comm_f2c

MPI_COMM_CALL_ERRHANDLER, MPI_Comm_call_errhandler

Calls the error handler assigned to the communicator with the error code supplied.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::Comm::Call_errhandler(int errorcode) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_COMM_CALL_ERRHANDLER(INTEGER COMM, INTEGER ERRORCODE, INTEGER IERROR)
```

Description

This subroutine calls the error handler assigned to the communicator with the error code supplied.

Parameters

comm
The communicator with the error handler (handle) (IN)

errorcode
The error code (integer) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_COMM_CALL_ERRHANDLER returns MPI_SUCCESS in C and C++ and the same value in IERROR if the error handler was successfully called (assuming the error handler itself is not fatal).

The default error handler for communicators is MPI_ERRORS_ARE_FATAL. Thus, calling MPI_COMM_CALL_ERRHANDLER will terminate the job if the default error handler has not been changed for this communicator or on the parent before the communicator was created. When a predefined error handler is used on *comm*, the error message printed by IBM PE MPI will indicate the error code that is passed in. You cannot force IBM PE MPI to issue a specific predefined error by passing its error code to this subroutine.

Error handlers should not be called recursively with MPI_COMM_CALL_ERRHANDLER. Doing this can create a situation where an infinite recursion is created. This can occur if MPI_COMM_CALL_ERRHANDLER is called inside an error handler.

Error codes and classes are associated with a task, so they can be used in any error handler. An error handler should be prepared to deal with any error code it is given. Furthermore, it is good practice to call an error handler only with the appropriate error codes. For example, communicator errors would normally be sent to the communicator error handler.

Errors

Invalid communicator

Invalid error code

MPI not initialized

MPI already finalized

Related information

- MPI_COMM_CREATE_ERRHANDLER
- MPI_COMM_GET_ERRHANDLER
- MPI_COMM_SET_ERRHANDLER
- MPI_ERRHANDLER_FREE

MPI::Comm::Clone

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

C++ synopsis

```
#include mpi.h
MPI::Cartcomm& MPI::Cartcomm::Clone() const;

#include mpi.h
MPI::Graphcomm& MPI::Graphcomm::Clone() const;

#include mpi.h
MPI::Intercomm& MPI::Intercomm::Clone() const;

#include mpi.h
MPI::Intracomm& MPI::Intracomm::Clone() const;
```

Description

This subroutine is a pure virtual function. For the derived communicator classes, MPI::Comm::Clone() behaves like Dup(), except that it returns a new object by reference.

Parameters

comm
The communicator (handle) (IN)

newcomm
The copy of *comm* (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

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_COMM_DUP

MPI_COMM_COMPARE, MPI_Comm_compare

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);
```

C++ synopsis

```
#include mpi.h
int MPI::Comm::Compare(const MPI::Comm& comm1, const MPI::Comm& comm2);
```

Fortran synopsis

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

Description

This subroutine compares the groups and contexts of two communicators. This 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 intercommunication).

MPI_UNEQUAL
Results if MPI_IDENT, MPI_CONGRUENT, or MPI_SIMILAR do not result.

Parameters

comm1
The first communicator (handle) (IN)

comm2
The second communicator (handle) (IN)

result
An integer specifying the result. The defined values are: MPI_IDENT, MPI_CONGRUENT, MPI_SIMILAR, and MPI_UNEQUAL. (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid communicators

MPI not initialized

MPI already finalized

Related information

- MPI_GROUP_COMPARE

MPI_COMM_CONNECT, MPI_Comm_connect

Establishes communication with a server.

C synopsis

```
#include <mpi.h>
int MPI_Comm_connect(char *port_name, MPI_Info info, int root,
MPI_Comm comm, MPI_Comm *newcomm);
```

C++ synopsis

```
#include <mpi.h>
MPI::Intercomm MPI::Intracomm::Connect(const char* port_name,
const MPI::Info& info, int root) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_COMM_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
CHARACTER*(*) PORT_NAME
INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
```

Description

This subroutine establishes communication with a server specified by *port_name*. It is collective over the calling communicator and returns an intercommunicator in which the remote group participated in an **MPI_COMM_ACCEPT**.

If the named port does not exist (or has been closed), **MPI_COMM_CONNECT** raises an error of class **MPI_ERR_PORT**. If the port exists, but does not have a pending **MPI_COMM_ACCEPT**, the connection attempt times out after 10 minutes or will succeed when the server calls **MPI_COMM_ACCEPT**. In the case of a timeout, **MPI_COMM_CONNECT** raises an error of class **MPI_ERR_PORT**.

MPI makes no guarantee of fairness in servicing connection attempts. That is, connection attempts are not necessarily satisfied in the order in which they were initiated, and competition from other connection attempts may prevent a particular connection attempt from being satisfied.

Parameters

port_name
The network address (string, used only on **root**) (IN)

info
An info is an object containing {key,value} pairs. IBM PE MPI **MPI_COMM_CONNECT** does not recognize any info keys. **MPI_INFO_NULL** is always valid (IN)

root
The rank in *comm* of the root node (integer) (IN)

comm
The intracommunicator over which the call is collective (handle) (IN)

newcomm
The intercommunicator with server as remote group (handle) (OUT)

Errors

Named port does not exist

Invalid rank (remote leader that called MPI_COMM_ACCEPT)

Related information

- **MPI_COMM_ACCEPT**

MPI_COMM_CREATE, MPI_Comm_create

Creates a new communicator with a given group.

C synopsis

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

C++ synopsis

```
#include mpi.h
MPI::Intercomm MPI::Intercomm::Create(const MPI::Group& group) const;
```

```
#include mpi.h
MPI::Intracomm MPI::Intracomm::Create(const MPI::Group& group) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_COMM_CREATE(INTEGER COMM_IN, INTEGER GROUP, INTEGER COMM_OUT,
      INTEGER IERROR)
```

Description

MPI_COMM_CREATE is a collective operation that is invoked by all tasks in the group associated with *comm_in*. This subroutine creates a new communicator *comm_out* with the communication group defined by *group* and a new context. Cached information is not propagated from *comm_in* to *comm_out*.

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_in*. The call is invoked by all tasks in *comm_in* even if they do not belong to the new group.

If *comm_in* is an inter-communicator, the output communicator is also an inter-communicator where the local group consists only of those tasks contained in *group*. The *group* argument should contain only those tasks in the local group of the input inter-communicator that are to be a part of *comm_out*. If either group does not specify at least one task in the local group of the inter-communicator, or if the calling task is not included in the group, MPI_COMM_NULL is returned.

Parameters

comm_in
The original communicator (handle) (IN)

group
A group of tasks that will be in the new communicator (handle) (IN)

comm_out
The new communicator (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

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 **comm_out** in subsequent calls to MPI_COMM_CREATE or other communicator constructors to further subdivide a computation into parallel sub-computations.

Errors

Fatal errors:

Conflicting collective operations on communicator

Invalid communicator

Invalid group
group is not a subset of the group associated with *comm_in*.

MPI not initialized

MPI already finalized

Related information

- MPI_COMM_DUP
- MPI_COMM_SPLIT

MPI_COMM_CREATE_ERRHANDLER, MPI_Comm_create_errhandler

Creates an error handler that can be attached to communicators.

C synopsis

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

C++ synopsis

```
#include mpi.h
static MPI::Errhandler MPI::Comm::Create_errhandler,
    (MPI::Comm::Errhandler_fn* function);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_COMM_CREATE_ERRHANDLER(EXTERNAL FUNCTION, INTEGER ERRHANDLER,
                           INTEGER IERROR)
```

Description

In C, the user subroutine should be a function of type **MPI_Comm_errhandler_fn**, which is defined as:

```
typedef void MPI_Comm_errhandler_fn(MPI_Comm *, int *, ...);
```

The first argument is the communicator in use, the second is the error code to be returned.

In C++, the user subroutine should be of the form:

```
typedef void MPI::Comm::Errhandler_fn(MPI::Comm &, int *, ...);
```

In Fortran, the user subroutine should be of the form:

```
SUBROUTINE COMM_ERRHANDLER_FN(COMM, ERROR_CODE, ...)
INTEGER COMM, ERROR_CODE
```

Parameters

function
The user-defined error handling procedure (function) (IN)

errhandler
The MPI error handler (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_COMM_CREATE_ERRHANDLER supersedes MPI_ERRHANDLER_CREATE.

The MPI standard specifies a **varargs** error handler prototype. A correct user error handler would be coded as:

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

IBM PE 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. The extra *errhandler* arguments can be accessed by using the C **varargs** (or **stdargs**) facility, but programs that do so will not port cleanly to other MPI implementations that might have different additional arguments.

The effective prototype for an error handler in IBM PE MPI is:

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

The additional arguments are:

routine_name
: the name of the MPI routine in which the error occurred

flag **true** if *badval* is meaningful, otherwise **false**

badval
: the non-valid integer or long 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_COMM_CALL_ERRHANDLER
- MPI_COMM_GET_ERRHANDLER
- MPI_COMM_SET_ERRHANDLER
- MPI_ERRHANDLER_CREATE
- MPI_ERRHANDLER_FREE

MPI_COMM_CREATE_KEYVAL, MPI_Comm_create_keyval

Creates a new attribute key for a communicator.

C synopsis

```
#include <mpi.h>
int MPI_Comm_create_keyval (MPI_Comm_copy_attr_function *comm_copy_attr_fn,
                            MPI_Comm_delete_attr_function *comm_delete_attr_fn,
                            int *comm_keyval, void *extra_state);
```

C++ synopsis

```
#include mpi.h
int MPI::Comm::Create_keyval(MPI::Comm::Copy_attr_function* comm_copy_attr_fn,
        MPI::Comm::Delete_attr_function* comm_delete_attr_fn,
        void* extra_state);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_COMM_CREATE_KEYVAL(EXTERNAL COMM_COPY_ATTR_FN, EXTERNAL COMM_DELETE_ATTR_FN,
                       INTEGER COMM_KEYVAL, INTEGER(KIND=MPI_ADDRESS_KIND)
                       EXTRA_STATE, INTEGER IERROR)
```

Description

This subroutine creates a new attribute key for a communicator and returns a handle to it in the *comm_keyval* argument. A key is unique in a task and is opaque to the user. Once created, a key can be used to associate an attribute with a communicator and access it within the local task.

The argument *comm_copy_attr_fn* can be specified as MPI_COMM_NULL_COPY_FN or MPI_COMM_DUP_FN in C, C++, or Fortran. The MPI_COMM_NULL_COPY_FN function returns *flag* = **0** and MPI_SUCCESS. MPI_COMM_DUP_FN is a simple copy function that sets *flag* = **1**, returns the value of *attribute_val_in* in *attribute_val_out*, and returns MPI_SUCCESS.

The argument *comm_delete_attr_fn* can be specified as MPI_COMM_NULL_DELETE_FN in C, C++, or Fortran. The MPI_COMM_NULL_DELETE_FN function, which supersedes MPI_NULL_DELETE_FN, returns MPI_SUCCESS.

The C callback functions are:

```
typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm, int comm_keyval,
                                        void *extra_state, void *attribute_val_in,
                                        void *attribute_val_out, int *flag);
```

and

```
typedef int MPI_Comm_delete_attr_function(MPI_Comm comm, int comm_keyval,
                                          void *attribute_val, void *extra_state);
```

The Fortran callback functions are:

```
SUBROUTINE COMM_COPY_ATTR_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
                             ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
INTEGER OLDCOMM, COMM_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT
LOGICAL FLAG
```

and

```
SUBROUTINE COMM_DELETE_ATTR_FN(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
                               EXTRA_STATE, IERROR)
INTEGER COMM, COMM_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

The C++ callback functions are:

```
typedef int MPI::Comm::Copy_attr_function(const MPI::Comm& oldcomm,
int comm_keyval, void* extra_state, void* attribute_val_in,
void* attribute_val_out, bool& flag);
```

and

```
typedef int MPI::Comm::Delete_attr_function(MPI::Comm& comm, int comm_keyval,
                                            void* attribute_val, void* extra_state);
```

The *attribute_val_in* parameter is the value of the attribute. The *attribute_val_out* parameter is the address of the value, so the function can set a new value. The *attribute_val_out* parameter is logically a **void****, but it is prototyped as **void***, to avoid the need for complex casting.

Parameters

extra_state
The extra state for callback functions (IN)

comm_copy_attr_fn
The copy callback function for *comm_keyval* (IN)

comm_delete_attr_fn
The delete callback function for *comm_keyval* (IN)

comm_keyval
The key value for future access (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_COMM_CREATE_KEYVAL supersedes MPI_KEYVAL_CREATE.

MPI_COMM_CREATE_KEYVAL does not inter-operate with MPI_KEYVAL_CREATE. The Fortran bindings for MPI-1 caching functions presume that an attribute is an INTEGER. The MPI-2 caching bindings use INTEGER (KIND=MPI_ADDRESS_KIND). In an MPI implementation that uses 64-bit addresses and 32-bit INTEGERS, the two formats would be incompatible.

Errors

MPI not initialized

MPI already finalized

Related information

- MPI_COMM_FREE_KEYVAL
- MPI_KEYVAL_CREATE

MPI_COMM_DELETE_ATTR, MPI_Comm_delete_attr

Removes an attribute value from a communicator.

C synopsis

```
#include <mpi.h>
int MPI_Comm_delete_attr (MPI_Comm comm, int comm_keyval);
```

C++ synopsis

```
#include mpi.h
void MPI::Comm::Delete_attr(int comm_keyval);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_COMM_DELETE_ATTR(INTEGER COMM, INTEGER COMM_KEYVAL, INTEGER IERROR)
```

Description

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

Parameters

comm
The communicator from which the attribute is deleted (handle) (INOUT)

comm_keyval
The key value (integer) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_COMM_DELETE_ATTR supersedes MPI_ATTR_DELETE.

MPI_COMM_DELETE_ATTR does not inter-operate with MPI_ATTR_DELETE. The Fortran bindings for MPI-1 caching functions presume that an attribute is an INTEGER. The MPI-2 caching bindings use INTEGER (KIND=MPI_ADDRESS_KIND). In an MPI implementation that uses 64-bit addresses and 32-bit INTEGERS, the two formats would be incompatible.

Errors

Fatal errors:

MPI not initialized

MPI already finalized

Wrong keytype (MPI_ERR_ARG) attribute key is not a communicator key

Related information

- MPI_ATTR_DELETE
- MPI_COMM_CREATE_KEYVAL
- MPI_COMM_GET_ATTR
- MPI_COMM_SET_ATTR

MPI_COMM_DISCONNECT, MPI_Comm_disconnect

Waits for all pending communication on *comm* to complete internally, deallocates the communicator object, and sets the handle to **MPI_COMM_NULL**.

C synopsis

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

C++ synopsis

```
#include <mpi.h>
void MPI::Comm::Disconnect();
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_COMM_DISCONNECT(COMM, IERROR)
INTEGER COMM, IERROR
```

Description

This subroutine waits for all pending communication on *comm* to complete internally, deallocates the communicator object, and sets the handle to **MPI_COMM_NULL**. It is a collective operation.

MPI_COMM_DISCONNECT cannot be called with the **MPI_COMM_WORLD** or **MPI_COMM_SELF** communicators. It can be called only if all communication is matched and locally complete, so that buffered data can be delivered to its destination. This requirement is the same as for **MPI_FINALIZE**.

MPI_COMM_DISCONNECT has the same action as **MPI_COMM_FREE**, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected tasks.

A fatal error in any task affects all tasks that are currently connected to the failing task, and is harmless to tasks that are not currently connected.
MPI_COMM_DISCONNECT waits for all communication to complete to be able to guarantee this behavior.

Parameters

comm
The communicator (handle) (INOUT)

Notes

To disconnect from other tasks, you might need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the local task and the remote tasks. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before the local task is completely independent of the remote tasks.

Errors

Pending collective communication operations when freeing communicator

Pending point-to-point communication operations when freeing communicator

MPI_Comm_disconnect cannot be called on MPI_COMM_WORLD or MPI_COMM_SELF

Related information

- **MPI_FINALIZE**
- **MPI_COMM_FREE**
- **MPI_WIN_FREE**
- **MPI_FILE_CLOSE**

MPI_COMM_DUP, MPI_Comm_dup

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);
```

C++ synopsis

```
#include mpi.h
MPI::Cartcomm MPI::Cartcomm::Dup() const;
```

```
#include mpi.h
MPI::Graphcomm MPI::Graphcomm::Dup() const;
```

```
#include mpi.h
MPI::Intercomm MPI::Intercomm::Dup() const;
```

```
#include mpi.h
MPI::Intracomm MPI::Intracomm::Dup() const;
```

Fortran synopsis

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

Description

MPI_COMM_DUP is a collective operation that is invoked by the group associated with *comm*. This subroutine 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 subroutine applies to both intra-communicators and inter-communicators.

Parameters

comm
The communicator (handle) (IN)

newcomm
The copy of *comm* (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

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 subroutine 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::Comm::Clone
- MPI_KEYVAL_CREATE

MPI_Comm_f2c

Returns a C handle to a communicator.

C synopsis

```
#include <mpi.h>
MPI_Comm MPI_Comm_f2c(MPI_Fint comm);
```

Description

This function does not have C++ or Fortran bindings. MPI_Comm_f2c returns a C handle to a communicator. If *comm* is a valid Fortran handle to a communicator, MPI_Comm_f2c returns a valid C handle to that same communicator. If *comm* is set to the Fortran value MPI_COMM_NULL, MPI_Comm_f2c returns the equivalent null C handle. If *comm* is not a valid Fortran handle, MPI_Comm_f2c returns a C handle that is not valid. The converted handle is returned as the function's value. There is no error detection or return code.

Parameters

comm
The communicator (handle) (IN)

Related information

- MPI_Comm_c2f

MPI_COMM_FREE, MPI_Comm_free

Marks a communicator for deallocation.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::Comm::Free(void);
```

Fortran synopsis

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

Description

This collective operation marks either an intracommunicator or an intercommunicator 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.

Note: When using IBM PE MPI's dynamic process management features, it may be necessary for the application to use MPI_COMM_DISCONNECT instead of MPI_COMM_FREE. MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes.

Parameters

comm
The communicator to be freed (handle) (INOUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

A delete_fn did not return MPI_SUCCESS.

Invalid communicator

MPI not initialized

MPI already finalized

Related information

- MPI_KEYVAL_CREATE

MPI_COMM_FREE_KEYVAL, MPI_Comm_free_keyval

Marks a communicator attribute key for deallocation.

C synopsis

```
#include <mpi.h>
int MPI_Comm_free_keyval (int *comm_keyval);
```

C++ synopsis

```
#include mpi.h
void MPI::Comm::Free_keyval(int& comm_keyval);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_COMM_FREE_KEYVAL(INTEGER COMM_KEYVAL, INTEGER IERROR)
```

Description

This subroutine 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_COMM_DELETE_ATTR to free one attribute instance. To free all attribute instances associated with a communicator, use MPI_COMM_FREE.

Parameters

comm_keyval
The key value (integer) (INOUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_COMM_FREE_KEYVAL supersedes MPI_KEYVAL_FREE.

MPI_COMM_FREE_KEYVAL does not inter-operate with MPI_KEYVAL_FREE. The Fortran bindings for MPI-1 caching functions presume that an attribute is an INTEGER. The MPI-2 caching bindings use INTEGER (KIND=MPI_ADDRESS_KIND). In an MPI implementation that uses 64-bit addresses and 32-bit INTEGERS, the two formats would be incompatible.

Errors

Fatal errors:

MPI not initialized

MPI already finalized

Wrong keytype (MPI_ERR_ARG) attribute key is not a communicator key

Related information

- MPI_COMM_CREATE_KEYVAL
- MPI_KEYVAL_FREE

MPI_COMM_GET_ATTR, MPI_Comm_get_attr

Retrieves the communicator attribute value identified by the key.

C synopsis

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

C++ synopsis

```
#include mpi.h
bool MPI::Comm::Get_attr(int comm_keyval, void* attribute_val) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_COMM_GET_ATTR(INTEGER COMM, INTEGER COMM_KEYVAL, INTEGER(KIND=MPI_ADDRESS_KIND)
                  ATTRIBUTE_VAL, LOGICAL FLAG, INTEGER IERROR)
```

Description

This subroutine 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* set to **false**.

Parameters

comm
The communicator to which the attribute is attached (handle) (IN)

comm_keyval
The key value (integer) (IN)

attribute_val
The attribute value, unless *flag* is **false** (OUT)

flag
Set to **false** if there is no attribute associated with the key (logical) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_COMM_GET_ATTR supersedes MPI_ATTR_GET.

MPI_COMM_GET_ATTR does not interoperate with MPI_ATTR_PUT. The Fortran bindings for MPI-1 caching functions presume that an attribute is an INTEGER. The MPI-2 caching bindings use INTEGER (KIND=MPI_ADDRESS_KIND). In an MPI implementation that uses 64-bit addresses and 32-bit INTEGERS, the two formats would be incompatible. You should not mix the MPI-1 and MPI-2 functions when managing attributes on communicators in Fortran.

The implementation of MPI_COMM_SET_ATTR and MPI_COMM_GET_ATTR 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 that holds arbitrary attribute information. Before calling MPI_COMM_SET_ATTR, you allocate

some storage for the attribute structure and then call MPI_COMM_SET_ATTR to record the address of this structure. You must make sure 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_COMM_GET_ATTR. Both MPI_COMM_SET_ATTR and MPI_COMM_GET_ATTR take a **void*** parameter, but this does not imply that the same parameter is passed to either one.

In Fortran:

MPI_COMM_SET_ATTR records an address-size integer and MPI_COMM_GET_ATTR returns the address-size integer. As the programmer, you can choose to encode all attribute information in this integer or maintain some kind of database in which the integer can index. Either of these approaches will port to other MPI implementations.

Many of the Fortran compilers include an additional feature that allows some of the same functions a C programmer would use. These compilers support the POINTER type, often referred to as a *Cray pointer*. XL Fortran is one of the compilers that supports the POINTER type. For more information, see *IBM XL Fortran Compiler Reference*

Errors

Fatal errors:

MPI not initialized

MPI already finalized

Wrong keytype (MPI_ERR_ARG) attribute key is not a communicator key

Related information

- MPI_ATTR_GET
- MPI_COMM_DELETE_ATTR
- MPI_COMM_SET_ATTR

MPI_COMM_GET_ERRHANDLER, MPI_Comm_get_errhandler

Retrieves the error handler currently associated with a communicator.

C synopsis

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

C++ synopsis

```
#include mpi.h
MPI::Errhandler MPI::Comm::Get_errhandler() const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_COMM_GET_ERRHANDLER(INTEGER COMM, INTEGER ERRHANDLER, INTEGER IERROR)
```

Description

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

Parameters

comm
The communicator (handle) (IN)

errhandler
The error handler that is currently associated with the communicator (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_COMM_GET_ERRHANDLER supersedes MPI_ERRHANDLER_GET.

Errors

Fatal errors:

Invalid communicator

MPI not initialized

MPI already finalized

Related information

- MPI_COMM_CALL_ERRHANDLER
- MPI_COMM_CREATE_ERRHANDLER
- MPI_COMM_SET_ERRHANDLER
- MPI_ERRHANDLER_FREE

MPI_COMM_GET_NAME, MPI_Comm_get_name

Returns the name that was last associated with a communicator.

C synopsis

```
#include <mpi.h>
int MPI_Comm_get_name (MPI_Comm comm, char *comm_name, int *resultlen);
```

C++ synopsis

```
#include mpi.h
void MPI::Comm::Get_name(char* comm_name, int& resultlen) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_COMM_GET_NAME(INTEGER COMM, CHARACTER*(*) COMM_NAME, INTEGER RESULTLEN,
    INTEGER IERROR)
```

Description

This subroutine returns the name that was last associated with the specified communicator. The name can be set and retrieved from any language. The same name is returned independent of the language used. The name should be allocated so it can hold a resulting string that is the length of MPI_MAX_OBJECT_NAME. For IBM PE MPI, the value of MPI_MAX_OBJECT_NAME is 256. MPI_COMM_GET_NAME returns a copy of the set name in *comm_name*.

Parameters

comm
The communicator with the name to be returned (handle) (IN)

comm_name
The name previously stored on the communicator, or an empty string if no such name exists (string) (OUT)

resultlen
The length of the returned name (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

If you did not associate a name with a communicator, or if an error occurs, MPI_COMM_GET_NAME returns an empty string (all spaces in Fortran or "" in C and C++). The two predefined communicators have predefined names associated with them. Thus, the names of MPI_COMM_SELF and MPI_COMM_WORLD have the default of MPI_COMM_SELF and MPI_COMM_WORLD. When a task originates from a spawn, the *parent* communicator is given the default name, MPI_COMM_PARENT. The fact that the system may have assigned a default name to a communicator does not prevent you from setting a name on the same communicator. Doing this removes the old name and assigns the new one.

It is safe simply to print the string returned by MPI_COMM_GET_NAME, as it is always a valid string even if there was no name.

Errors

Fatal errors:

Invalid communicator

MPI already finalized

MPI not initialized

Related information

- MPI::Comm::Clone
- MPI_COMM_DUP
- MPI_COMM_SET_NAME

MPI_COMM_GET_PARENT, MPI_Comm_get_parent

Returns the parent intercommunicator of the current task, if the task was started with **MPI_COMM_SPAWN** or **MPI_COMM_SPAWN_MULTIPLE**.

C synopsis

```
#include <mpi.h>
int MPI_Comm_get_parent(MPI_Comm *parent);
```

C++ synopsis

```
#include <mpi.h>
static MPI::Intercomm MPI::Comm::Get_parent();
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_COMM_GET_PARENT(PARENT, IERROR)
INTEGER PARENT, IERROR
```

Description

This subroutine returns the parent intercommunicator of the current task, if the task was started with **MPI_COMM_SPAWN** or **MPI_COMM_SPAWN_MULTIPLE**. This parent intercommunicator is created implicitly inside of **MPI_INIT** and is the same intercommunicator that is returned by **MPI_SPAWN** in the parents.

If the task was not spawned, **MPI_COMM_GET_PARENT** returns **MPI_COMM_NULL**. After the parent communicator is freed or disconnected, **MPI_COMM_GET_PARENT** returns **MPI_COMM_NULL**.

Parameters

parent
The parent intercommunicator handle (OUT)

Notes

Calling **MPI_COMM_GET_PARENT** a second time returns the same handle. The calls are not reference counted, so calling **MPI_COMM_DISCONNECT** or **MPI_COMM_FREE** on the handle returned by any call to **MPI_COMM_GET_PARENT** destroys the parent intercommunicator and leaves any other references dangling. Calling **MPI_COMM_FREE** on the parent intercommunicator is not useful.

Related information

- **MPI_COMM_SPAWN**
- **MPI_COMM_SPAWN_MULTIPLE**

MPI_COMM_GROUP, MPI_Comm_group

Returns the group handle associated with a communicator.

C synopsis

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

C++ synopsis

```
#include mpi.h
MPI::Group MPI::Comm::Get_group() const;
```

Fortran synopsis

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

Description

This subroutine returns the group handle associated with a communicator.

Parameters

comm
The communicator (handle) (IN)

group
The group corresponding to *comm* (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

If *comm* is an inter-communicator, *group* is set to the local group. To determine the remote group of an inter-communicator, use MPI_COMM_REMOTE_GROUP.

Errors

Invalid communicator

MPI not initialized

MPI already finalized

Related information

- MPI_COMM_REMOTE_GROUP

MPI_COMM_JOIN, MPI_Comm_join

Creates an intercommunicator from the union of two MPI tasks that are connected by a socket.

C synopsis

```
#include <mpi.h>
int MPI_Comm_join(int fd, MPI_Comm *intercomm);
```

C++ synopsis

```
#include <mpi.h>
static MPI::Intercomm MPI::Comm::Join(const int fd);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_COMM_JOIN(FD, INTERCOMM, IERROR)
INTEGER FD, INTERCOMM, IERROR
```

Description

This subroutine creates an intercommunicator from the union of two MPI tasks that are connected by a socket. **MPI_COMM_JOIN** will succeed if both tasks were started under the same **poe** invocation, and will fail if they were not.

Parameters

fd A socket file descriptor (IN)

intercomm
A new intercommunicator (handle) (OUT)

Errors

Invalid rank (remote task that called MPI_COMM_JOIN)

Related information

- **MPI_COMM_DISCONNECT**

MPI_COMM_RANK, MPI_Comm_rank

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);
```

C++ synopsis

```
#include mpi.h
int MPI::Comm::Get_rank() const;
```

Fortran synopsis

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

Description

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

You can use this subroutine 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 to *size-1* , where *size* is the output parameter of MPI_COMM_SIZE.

If *comm* is an inter-communicator, *rank* is the rank of the local task in the local group.

Parameters

comm
The communicator (handle) (IN)

rank
An integer specifying the rank of the calling task in group of *comm* (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid communicator

MPI not initialized

MPI already finalized

Related information

- MPI_GROUP_RANK

MPI_COMM_REMOTE_GROUP, MPI_Comm_remote_group

Returns the handle of the remote group of an inter-communicator.

C synopsis

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

C++ synopsis

```
#include mpi.h
MPI::Group MPI::Intercomm::Get_remote_group() const;
```

Fortran synopsis

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

Description

This subroutine is a local operation that returns the handle of the remote group of an inter-communicator.

Parameters

comm
The inter-communicator (handle) (IN)

group
The remote group corresponding to *comm*. (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

To determine the local group of an inter-communicator, use MPI_COMM_GROUP.

Errors

Invalid communicator

Invalid communicator type
Communication type must be inter-communicator.

MPI not initialized

MPI already finalized

Related information

- MPI_COMM_GROUP

MPI_COMM_REMOTE_SIZE, MPI_Comm_remote_size

Returns the size of the remote group of an inter-communicator.

C synopsis

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

C++ synopsis

```
#include mpi.h
int MPI::Intercomm::Get_remote_size() const;
```

Fortran synopsis

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

Description

This subroutine is a local operation that returns the size of the remote group of an inter-communicator.

Parameters

comm
The inter-communicator (handle) (IN)

size
An integer specifying the number of tasks in the remote group of *comm*. (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

To determine the size of the local group of an inter-communicator, use MPI_COMM_SIZE.

Errors

Invalid communicator

Invalid communicator type
Communication type must be inter-communicator.

MPI not initialized

MPI already finalized

Related information

- MPI_COMM_SIZE

MPI_COMM_SET_ATTR, MPI_Comm_set_attr

Attaches the communicator attribute value to the communicator and associates it with the key.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::Comm::Set_attr(int comm_keyval, const void* attribute_val) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_COMM_SET_ATTR(INTEGER COMM, INTEGER COMM_KEYVAL, INTEGER(KIND=MPI_ADDRESS_KIND)
                  ATTRIBUTE_VAL, INTEGER IERROR)
```

Description

This subroutine stores the attribute value for retrieval by MPI_COMM_GET_ATTR. 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.

Parameters

comm
The communicator to which the attribute will be attached (handle) (INOUT)

comm_keyval
The key value (integer) (IN)

attribute_val
The attribute value (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_COMM_SET_ATTR supersedes MPI_ATTR_PUT.

MPI_COMM_SET_ATTR does not interoperate with MPI_ATTR_GET. The Fortran bindings for MPI-1 caching functions presume that an attribute is an INTEGER. The MPI-2 caching bindings use INTEGER (KIND=MPI_ADDRESS_KIND). In an MPI implementation that uses 64-bit addresses and 32-bit INTEGERS, the two formats would be incompatible. You should not mix the MPI-1 and MPI-2 functions when managing attributes on communicators in Fortran.

The implementation of MPI_COMM_SET_ATTR and MPI_COMM_GET_ATTR 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 that holds arbitrary attribute information. Before calling MPI_COMM_SET_ATTR, you allocate some storage for the attribute structure and then call MPI_COMM_SET_ATTR to record the address of this structure. You must make sure 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_COMM_GET_ATTR. Both MPI_COMM_SET_ATTR and MPI_COMM_GET_ATTR take a **void*** parameter, but this does not imply that the same parameter is passed to either one.

In Fortran:

MPI_COMM_SET_ATTR records an address-size integer and MPI_COMM_GET_ATTR returns the address-size integer. As the programmer, you can choose to encode all attribute information in this integer or maintain some kind of database in which the integer can index. Either of these approaches will port to other MPI implementations.

Many of the Fortran compilers include an additional feature that allows some of the same functions a C programmer would use. These compilers support the POINTER type, often referred to as a *Cray pointer*. XL Fortran is one of the compilers that supports the POINTER type. For more information, see *IBM XL Fortran Compiler Reference*

Errors

Fatal errors:

MPI not initialized

MPI already finalized

Wrong keytype (MPI_ERR_ARG) attribute key is not a communicator key

Related information

- MPI_ATTR_PUT
- MPI_COMM_DELETE_ATTR
- MPI_COMM_GET_ATTR

MPI_COMM_SET_ERRHANDLER, MPI_Comm_set_errhandler

Attaches a new error handler to a communicator.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::Comm::Set_errhandler(const MPI::Errhandler& errhandler);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_COMM_SET_ERRHANDLER(INTEGER COMM, INTEGER ERRHANDLER, INTEGER IERROR)
```

Description

This subroutine attaches a new error handler to a communicator. The error handler must be either a predefined error handler, or an error handler created by a call to MPI_COMM_CREATE_ERRHANDLER. The previously-attached error handler is replaced.

Parameters

comm
The communicator (handle) (INOUT)

errhandler
The new error handler for the communicator (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_COMM_SET_ERRHANDLER supersedes MPI_ERRHANDLER_SET.

For information about a predefined error handler for C++, see *IBM Parallel Environment Runtime Edition: MPI Programming Guide*.

Errors

Invalid communicator

Invalid error handler

MPI not initialized

MPI already finalized

Related information

- MPI_COMM_CALL_ERRHANDLER
- MPI_COMM_CREATE_ERRHANDLER
- MPI_COMM_GET_ERRHANDLER
- MPI_ERRHANDLER_FREE

MPI_COMM_SET_NAME, MPI_Comm_set_name

Associates a name string with a communicator.

C synopsis

```
#include <mpi.h>
int MPI_Comm_set_name (MPI_Comm comm, char *comm_name);
```

C++ synopsis

```
#include mpi.h
void MPI::Comm::Set_name(const char* comm_name);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_COMM_SET_NAME(INTEGER COMM, CHARACTER*(*) COMM_NAME, INTEGER IERROR)
```

Description

This subroutine lets you associate a name string with a communicator. The name is intended for use as an identifier, so when the communicator is copied or duplicated, the name does not propagate.

The character string that is passed to MPI_COMM_SET_NAME is copied to space managed by the MPI library (so it can be freed by the caller immediately after the call, or allocated on the stack). Leading spaces in the name are significant, but trailing spaces are not.

Parameters

comm
The communicator with the identifier to be set (handle) (INOUT)

comm_name
The character string that is saved as the communicator's name (string) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_COMM_SET_NAME is a local (noncollective) operation, which affects only the name of the communicator as specified in the task that made the MPI_COMM_SET_NAME call. There is no requirement that the same (or any) name be assigned to a communicator in every task where that communicator exists. However, to avoid confusion, it is a good idea to give the same name to a communicator in all of the tasks where it exists.

The length of the name that can be stored is limited to the value of MPI_MAX_OBJECT_NAME in Fortran and MPI_MAX_OBJECT_NAME-1 in C and C++ to allow for the null terminator. An attempt to use a longer name is not an error, but will result in truncation of the name. For IBM PE MPI, the value of MPI_MAX_OBJECT_NAME is 256.

Associating a name with a communicator has no effect on the semantics of an MPI program, and (necessarily) increases the store requirement of the program, because the names must be saved. Therefore, there is no requirement that you use this function to associate names with communicators. However, debugging and

profiling MPI applications can be made easier if names are associated with communicators, as the debugger or profiler should then be able to present information in a less cryptic manner.

Errors

Fatal errors:

Invalid communicator

MPI already finalized

MPI not initialized

Related information

- MPI::Comm::Clone
- MPI_COMM_DUP
- MPI_COMM_GET_NAME

MPI_COMM_SIZE, MPI_Comm_size

Returns the size of the group associated with a communicator.

C synopsis

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

C++ synopsis

```
#include mpi.h
int MPI::Comm::Get_size() const;
```

Fortran synopsis

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

Description

This subroutine returns the size of the group associated with a communicator.

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

You can use this subroutine 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 output parameter of MPI_COMM_SIZE. The rank and size information can then be used to partition work across the available tasks.

Parameters

comm
The communicator (handle) (IN)

size
An integer specifying the number of tasks in the group of *comm* (OUT)

IERROR
The Fortran return code. It is always the last argument.

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_COMM_GROUP
- MPI_COMM_RANK
- MPI_COMM_REMOTE_SIZE

- MPI_GROUP_FREE
- MPI_GROUP_SIZE

MPI_COMM_SPAWN, MPI_Comm_spawn

Starts a number of MPI tasks and establishes communication with them.

C synopsis

```
#include <mpi.h>
int MPI_Comm_spawn (char *command, char *argv[], int maxprocs, MPI_Info
info, int root, MPI_Comm comm, MPI_Comm *intercomm,
int array_of_errcodes[]);
```

C++ synopsis

```
#include <mpi.h>
int MPI::Intercomm MPI::Intracomm::Spawn(const char* command,
const char* argv[], int maxprocs, const MPI::Info& info,
int root, int array_of_errcodes[]) const;
```

```
#include <mpi.h>
int MPI::Intercomm MPI::Intracomm::Spawn(const char* command,
const char* argv[], int maxprocs, const MPI::Info& info,
int root) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,
ARRAY_OF_ERRCODES)
CHARACTER*(*) COMMAND, ARGV(*)
INTEGER INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*),
IERROR
```

Description

This subroutine starts a number of MPI tasks, establishes communication with them, and returns an intercommunicator.

MPI_COMM_SPAWN tries to start *maxprocs* identical copies of the MPI program specified by *command*, establishing communication with them and returning an intercommunicator. The spawned tasks are referred to as children. The children have their own **MPI_COMM_WORLD**, which is separate from that of the parents. **MPI_COMM_SPAWN** is collective over *comm*, and also may not return until MPI_INIT has been called in the children. Similarly, MPI_INIT in the children may not return until all parents have called **MPI_COMM_SPAWN**. In this sense, **MPI_COMM_SPAWN** in the parents and **MPI_INIT** in the children form a collective operation over the union of parent and child tasks. The intercommunicator returned by **MPI_COMM_SPAWN** contains the parent tasks in the local group and the child tasks in the remote group. The ordering of tasks in the local and remote groups is the same as the ordering of the group of the *comm* in the parents and of **MPI_COMM_WORLD** of the children, respectively. This intercommunicator can be obtained in the children through the function **MPI_COMM_GET_PARENT**.

A spawn call with a default behavior is called *hard*. A spawn call for which fewer than *maxprocs* tasks may be returned is called *soft*. IBM PE MPI supports a restricted form of soft spawn in which either 0 or *maxprocs* tasks are spawned. This allows the application to retry with a smaller maxprocs or to retry in the expectation that resources will become available. This is useful because the MPI Standard provides for freeing of resources, but does not provide a mechanism for knowing exactly when those resources become available for reuse.

To use the soft spawn support of IBM PE MPI, the application must pass in an **MPI_Info** object with a valid key and value. The key is **IBM_soft_spawn** and the value for soft spawn is **true**. Setting **IBM_soft_spawn** to a value other than **true** is ignored and the result is a hard spawn. Passing **MPI_INFO_NULL** as the **info** argument results in a hard spawn.

Parameters

command
The name of the program to be spawned (string, significant only at root) (IN)

argv
The arguments to the command (array of strings, significant only at root) (IN)

maxprocs
The maximum number of tasks to start (integer, significant only at root) (IN)

info
a set of key-value pairs that tell the runtime system where and how to start the tasks (handle, significant only at root) (IN)

root
The rank of the task in which previous arguments are examined (IN)

comm
The intracommunicator that contains the group of spawning tasks (IN)

intercomm
The intercommunicator between the original group (referred to as *master*) and the newly spawned group (referred to as *worker*) (OUT)

array_of_errcodes
Specifies one error code per task (array of integer) (OUT)

Errors

Invalid errcodes buffer (NULL)

Invalid maxprocs value

Invalid command buffer (NULL)

Invalid rank (remote leader of the spawned group)

Not enough resources to spawn tasks (when spawn is hard)

Error in spawning tasks

Related information

- **MPI_INIT**
- **MPI_COMM_GET_PARENT**
- **MPI_COMM_SPAWN_MULTIPLE**

MPI_COMM_SPAWN_MULTIPLE, MPI_Comm_spawn_multiple

Spawns multiple binaries, or the same binary with multiple sets of arguments, establishes communication with them, and returns an intercommunicator.

C synopsis

```
#include <mpi.h>
int MPI_Comm_spawn_multiple(int count, char *array_of_commands[],
char **array_of_argv[], int array_of_maxprocs[],
MPI_Info array_of_info[], int root, MPI_Comm comm,
MPI_Comm *intercomm, int array_of_errcodes[]);
```

C++ synopsis

```
#include <mpi.h>
int MPI::Intercomm MPI::Intracomm::Spawn_multiple(int count,
const char* array_of_commands[], const char** array_of_argv[],
const int array_of_maxprocs[],
const MPI::Info array_of_info[], int root,
int array_of_errcodes[]);
```

```
#include <mpi.h>
int MPI::Intercomm MPI::Intracomm::Spawn_multiple(int count,
const char* array_of_commands[], const char** array_of_argv[],
const int array_of_maxprocs[],
const MPI::Info array_of_info[], int root)
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,
ARRAY_OF_ERRCODES, IERROR)
INTEGER COUNT, ARRAY_OF_INFO(*), ARRAY_OF_MAXPROCS(*), ROOT, COMM,
INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR
CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)
```

Description

While **MPI_COMM_SPAWN** is sufficient for most cases, it does not allow the spawning of multiple binaries, or of the same binary with multiple sets of arguments. For each command specified in the array_of_commands, this subroutine starts its corresponding **maxprocs** (from **array_of_maxprocs**) instances of that command, and establishes communication with it. This subroutine returns an intercommunicator with all spawned tasks in the remote group.

MPI_COMM_SPAWN_MULTIPLE is identical to **MPI_COMM_SPAWN** except that there are multiple executable specifications. The first argument, **count**, gives the number of specifications. Each of the next four arguments are simply arrays of the corresponding arguments in **MPI_COMM_SPAWN**.

For specifying soft spawn, each element of the **array_of_info** should have the key **IBM_soft_spawn**, each with the value **true**.

Parameters

count
The number of commands (positive integer, significant to MPI only at root) (IN)

array_of_commands
The programs to be run (array of strings, significant only at root) (IN)

array_of_argv
The arguments to commands (array of strings, significant only at root) (IN)

array_of_maxprocs
The maximum number of tasks to start for each command (array of integer, significant only at root) (IN)

array_of_info
The info objects that tell the runtime system where and how to start tasks. The same info handle can be passed as each element in **array_of_info** (array of handles, significant only at root) (IN).

root
The rank of the task in which previous arguments are examined (IN)

comm
The intracommunicator that contains a group of spawning tasks (IN)

intercomm
The intercommunicator between the original group and the newly spawned group (OUT)

array_errcodes
Specifies one error code per task (array of integer) (OUT)

Errors

Invalid count

Invalid info (the key and value in all the elements of array_of_info are the same)

Invalid errcodes buffer (NULL)

Invalid maxprocs value

Invalid command buffer (NULL)

Invalid rank (remote leader of the spawned group)

Not enough resources to spawn tasks

Error in spawning task

Related information

- **MPI_INIT**
- **MPI_COMM_GET_PARENT**
- **MPI_COMM_SPAWN**

MPI_COMM_SPLIT, MPI_Comm_split

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

C synopsis

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

C++ synopsis

```
#include mpi.h
MPI::Intercomm MPI::Intercomm::Split(int color, int key) const;
```

```
#include mpi.h
MPI::Intracomm MPI::Intracomm::Split(int color, int key) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_COMM_SPLIT(INTEGER COMM_IN, INTEGER COLOR, INTEGER KEY,
    INTEGER COMM_OUT, INTEGER IERROR)
```

Description

MPI_COMM_SPLIT is a collective operation that partitions the group associated with *comm_in* 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 *comm_out*. If a task supplies the color value MPI_UNDEFINED, *comm_out* returns MPI_COMM_NULL. Even though this is a collective operation, each task is allowed to provide different values for *color* and *key*.

The value of *color* must be greater than or equal to 0.

Parameters

comm_in
The original communicator (handle) (IN)

color
An integer specifying control of subset assignment (IN)

key
An integer specifying control of rank assignment (IN)

comm_out
The new communicator (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

The result of MPI_COMM_SPLIT on an inter-communicator is that those tasks on one side of the inter-communicator with the same color as those tasks on the other side of the inter-communicator combine to create a new inter-communicator. The *key* argument describes the relative rank of tasks on each side of the inter-communicator. For those colors that are specified only on one side of the inter-communicator, MPI_COMM_NULL is returned. MPI_COMM_NULL is also returned to those tasks that specify MPI_UNDEFINED as the color.

Errors

Fatal errors:

Conflicting collective operations on communicator

Invalid color
color < 0

Invalid communicator

MPI not initialized

MPI already finalized

Related information

- MPI_CART_SUB

MPI_COMM_TEST_INTER, MPI_Comm_test_inter

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

C synopsis

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

C++ synopsis

```
#include mpi.h
bool MPI::Comm::Is_inter() const;
```

Fortran synopsis

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

Description

This subroutine is used to determine if a communicator is an inter- or intra-communicator.

If *comm* is an inter-communicator, the call returns **true**. If *comm* is an intra-communicator, the call returns **false**.

Parameters

comm
The communicator (handle) (IN)

flag
The communicator type (logical) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

Though many subroutines accept either an inter-communicator or an intra-communicator, the usage and semantic can be quite different.

Errors

Invalid communicator

MPI not initialized

MPI already finalized

MPI_DIMS_CREATE, MPI_Dims_create

Defines a Cartesian grid to balance tasks.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::Compute_dims(int nnodes, int ndims, int dims[]);
```

Fortran synopsis

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

Description

This subroutine 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]* is equal to 0 are modified by the call.

Parameters

nnodes
An integer specifying the number of nodes in a grid (IN)

ndims
An integer specifying the number of Cartesian dimensions (IN)

dims
An integer array of size *ndims* that specifies the number of nodes in each dimension. (INOUT)

IERROR
The Fortran return code. It is always the last argument.

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 nonzero entries of *dims*

Related information

- MPI_CART_CREATE

MPI_DIST_GRAPH_CREATE, MPI_Dist_graph_create

Returns a handle to a new communicator to which the distributed graph topology information is attached.

C synopsis

```
#include <mpi.h>
int MPI_Dist_graph_create(MPI_Comm comm_old, int n,
     int sources[], int degrees[], int destinations[],
     int weights[], MPI_Info info, int reorder,
     MPI_Comm *comm_dist_graph)
```

C++ synopsis

```
#include mpi.h
void MPI::Distgraphcomm MPI::Intracomm::Dist_graph_create(int n,
const int sources[], const int degrees[],
const int destinations[], const int weights[],
const MPI::Info& info, bool reorder) const

#include mpi.h
void MPI::Distgraphcomm MPI::Intracomm::Dist_graph_create(int n,
     const int sources[], const int degrees[],
     const int destinations[], const MPI::Info& info, bool reorder)
     const
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_DIST_GRAPH_CREATE(INTEGER COMM_OLD, INTEGER N, INTEGER SOURCES,
         INTEGER DEGREES, INTEGER DESTINATIONS, INTEGER WEIGHTS, INTEGER INFO,
         INTEGER REORDER, LOGICAL REORDER, INTEGER COMM_DIST_GRAPH,
         INTEGER IERROR)
```

Description

This subroutine returns a handle to a new communicator to which the distributed graph topology information is attached. Concretely, each process calls the constructor with a set of directed (source,destination) communication edges. Every process passes an array of *n* source nodes in the sources array. For each source node, a non-negative number of destination nodes is specified in the degrees array. The destination nodes are stored in the corresponding consecutive segment of the destinations array. More precisely, if the i-th node in *sources* is **s**, this specifies *degrees* [i] edges (s,d) with **d** of the j-th such edge stored in *destinations*[degrees[0]+...+degrees[i-l]+j]. The weight of this edge is stored in *weights*[degrees[0]+...+degrees[i-l]+j]. Both the *sources* and the *destinations* arrays may contain the same node more than once, and the order in which nodes are listed as destinations or sources is not significant. Similarly, different processes may specify edges with the same source and destination nodes. Source and destination nodes must process ranks of *comm_old*. Different processes may specify different numbers of source and destination nodes, as well as different source to destination edges. This allows a fully-distributed specification of the communication graph. Isolated processes (processes with no outgoing or incoming edges, that is, processes that do not occur as source or destination node in the graph specification) are allowed.

The call creates a new communicator *comm_dist_graph*, of distributed graph topology type, to which topology information has been attached. The number of processes in *comm_dist_graph* is identical to the number of processes in *comm_old*. The call to **MPI_Dist_graph_create** is collective.

If *reorder* = **false**, all processes will have the same rank in *comm_dist_graph* as in *comm_old*. If *reorder* = **true**, then the MPI library is free to remap to other processes (of *comm_old*) in order to improve communication on the edges of the communication graph.

Parameters

comm_old
Input communicator (handle) (IN)

n Number of source nodes for which this process specifies edges (non-negative integer) (IN)

sources
Array containing the n source nodes for which this process specifies edges (array of non-negative integers) (IN)

degrees
Array specifying the number of destinations for each source node (array of non-negative integers) (IN)

destinations
Destination nodes for the source nodes in the source node array (array of non-negative integers) (IN)

weights
Weights for source to destination edges (array of non-negative integers) (IN)

info
Hints on optimization and interpretation of *weights* (handle) (IN)

reorder
The process may be reordered (true) or not (false) (logical) (IN)

comm_dist_graph
Communicator with distributed graph topology (handle) (OUT)

Errors

Negative array length given

Negative number of degrees for source

Negative degree value

Invalid neighbor rank

Inconsistent use of MPI_UNWEIGHTED

Invalid output communicator

Input communicator not intracomm

Related information

- MPI_DIST_GRAPH_CREATE

MPI_DIST_GRAPH_CREATE_ADJACENT, MPI_Dist_graph_create_adjacent

Returns a handle to a new communicator to which the distributed graph topology information is attached.

C synopsis

```
#include <mpi.h>
int MPI_Dist_graph_create_adjacent(MPI_Comm comm_old,
int indegree, int sources[], int sourceweights[],
int outdegree, int destinations[], int destweights[],
MPI_Info info, int reorder, MPI_Comm *comm_dist_graph);
```

C++ synopsis

```
#include mpi.h
void MPI::Distgraphcomm MPI::Intracomm::Dist_graph_create_adjacent(int
     indegree, const int sources[], const int sourceweights[],
     int outdegree, const int destinations[], const int destweights[],
     const MPI::Info& info, bool reorder);
     const (binding deprecated, see section 15.2 of the
     MPI 2.2 Standard)

#include mpi.h
void MPI::Distgraphcomm MPI::Intracomm::Dist_graph_create_adjacent(int indegree,
     const int sources[], int outdegree, const int destinations[],
     const MPI::Info& info, bool reorder)
     const binding deprecated, see section 15.2 of the MPI 2.2 Standard)
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_DIST_GRAPH_CREATE_ADJACENT(INTEGER COMM_OLD, INTEGER INDEGREE,
       INTEGER SOURCES, INTEGER SOURCEWEIGHTS, INTEGER OUTDEGREE,
       INTEGER DESTINATIONS, INTEGER DESTWEIGHTS, INTEGER INFO,
       INTEGER REORDER, LOGICAL REORDER, INTEGER COMM_DIST_GRAPH,
       INTEGER IERROR)
```

Description

This subroutine returns a handle to a new communicator to which the distributed graph topology information is attached. Each process passes all information about the edges to its neighbors in the virtual distributed graph topology. The calling processes must ensure that each edge of the graph is described in the source and in the destination process with the same weights.

Parameters

comm_old
Input communicator (handle) (IN)

indegree
Size of *sources* and *sourceweights* arrays (non-negative integer) (IN)

sources
Ranks of processes for which the calling process is a destination (array of non-negative integers) (IN)

sourceweights
Weights of the edges into the calling process (array of non-negative integers) (IN)

outdegree
Size of destinations and destweights arrays (non-negative integer) (IN)

destinations
Ranks of processes for which the calling process is a source (array of non-negative integers) (IN)

destweights
Weights of the edges out of the calling process (array of non-negative integers) (IN)

info
Hints on optimization and interpretation of *weights* (handle) (IN)

reorder
The ranks may be reordered (true) or not (false) (logical) (IN)

comm_dist_graph
Communicator with distributed graph topology (handle) (OUT)

Errors

Negative number of degrees

Negative degree value

Inconsistent use of MPI_UNWEIGHTED

Invalid output communicator

Input communicator not intracomm

No topology

Related information

- MPI_DIST_GRAPH_CREATE

MPI_DIST_GRAPH_NEIGHBORS, MPI_Dist_graph_neighbors

Provides adjacency information for a distributed graph topology.

C synopsis

```
#include <mpi.h>
int MPI_Dist_graph_neigbors(MPI_Comm comm, int maxindegree,
     int sources[], int sourceweights[], int maxoutdegree,
     int destinations[], int destweights[])
```

C++ synopsis

```
#include mpi.h
void MPI::Distgraphcomm::Get_dist_neighbors(int maxindegree,
     int sources[], int sourceweights[], int maxoutdegree,
     int destinations[], int destweights[])
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_DIST_GRAPH_NEIGHBORS(INTEGER COMM, INTEGER MAXINDEGREE,
        INTEGER SOURCES, INTEGER SOURCEWEIGHTS, INTEGER MAXOUTDEGREE,
        INTEGER DESTINATIONS, INTEGER DESTWEIGHTS, INTEGER IERROR)
```

Description

The **MPI_DIST_GRAPH_NEIGHBORS_COUNT** and **MPI_DIST_GRAPH_NEIGHBORS** calls are local. The number of edges into and out of the process returned by **MPI_DIST_GRAPH_NEIGHBORS_COUNT** are the total number of such edges given in the call to **MPI_DIST_GRAPH_CREATE_ADJACENT** or **MPI_DIST_GRAPH_CREATE** (potentially by processes other than the calling process, in the case of **MPI_DIST_GRAPH_CREATE**). Multiple defined edges are all counted and returned by **MPI_DIST_GRAPH_NEIGHBORS** in some order.

If **MPI_UNWEIGHTED** is supplied for *sourceweights* or *destweights*, or both, or if **MPI_UNWEIGHTED** was supplied during the construction of the graph, then no weight information is returned in that array or those arrays. The only requirement on the order of values in *sources* and *destinations* is that two calls to the routine with the same input argument *comm* will return the same sequence of edges. If *maxindegree* or maxoutdegree is smaller than the numbers returned by **MPI_DIST_GRAPH_NEIGHBORS_COUNT**, only the first part of the full list is returned.

Note that the order of returned edges does not need to be identical to the order that was provided in the creation of *comm* for the case that **MPI_DIST_GRAPH_CREATE_ADJACENT** was used.

Parameters

comm
Communicator with distributed graph topology (handle) (IN)

maxindegree
Size of *sources* and *sourceweights* arrays (non-negative integer) (IN)

sources
Processes for which the calling process is a destination (array of non-negative integers) (OUT)

sourceweights
Weights of the edges into the calling process (array of non-negative integers) (OUT)

maxoutdegree
Size of *destinations* and *destweights* arrays (non-negative integer) (IN)

destinations
Processes for which the calling process is a source (array of non-negative integers) (OUT)

destweights
Weights of the edges out of the calling process (array of non-negative integers) (OUT)

Errors

Invalid input communicator

Input communicator not a DIST_GRAPH comm

Negative length given for MAXINDEGREE

Negative length given for MAXOUTDEGREE

Related information

- MPI_DIST_GRAPH_CREATE
- MPI_DIST_GRAPH_CREATE_ADJACENT
- MPI_DIST_GRAPH_NEIGHBORS_COUNT

MPI_DIST_GRAPH_NEIGHBORS_COUNT, MPI_Dist_graph_neighbors_count

Provides adjacency information for a distributed graph topology.

C synopsis

```
#include <mpi.h>
int MPI_Dist_graph_neigbors_count(MPI_Comm comm, int *indegree,
     int *outdegree, int *weighted)
```

C++ synopsis

```
#include mpi.h
void MPI::Distgraphcomm::Get_dist_neighbors_count(int rank,
     int indegree[], int outdegree[], bool& weighted) const
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_DIST_GRAPH_NEIGHBORS_COUNT(INTEGER COMM, INTEGER INDEGREE,
      INTEGER OUTDEGREE, INTEGER WEIGHTED, LOGICAL WEIGHTED,
      INTEGER IERROR)
```

Description

Provides adjacency information for a distributed graph topology.

Parameters

comm
Communicator with distributed graph topology (handle) (IN)

indegree
Number of edges into this process (non-negative integer) (OUT)

outdegree
Number of edges out of this process (non-negative integer) (OUT)

weighted
False if **MPI_UNWEIGHTED** was supplied during creation, true otherwise (logical) (OUT)

Errors

Invalid input communicator

Input communicator not a DIST_GRAPH comm

Related information

- MPI_DIST_GRAPH_CREATE
- MPI_DIST_GRAPH_CREATE_ADJACENT
- MPI_DIST_GRAPH_NEIGHBORS

MPI_Errhandler_c2f

Translates a C error handler into a Fortran handle to the same error handler.

C synopsis

```
#include <mpi.h>
MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler);
```

Description

This function does not have C++ or Fortran bindings. MPI_Errhandler_c2f translates a C error handler into a Fortran handle to the same error handler. The converted handle is returned as the function's value. There is no error detection or return code.

Parameters

errhandler
The error handler (handle) (IN)

Related information

- MPI_Errhandler_f2c

MPI_ERRHANDLER_CREATE, MPI_Errhandler_create

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' or USE MPI
MPI_ERRHANDLER_CREATE(EXTERNAL FUNCTION,INTEGER ERRHANDLER,
      INTEGER IERROR)
```

Description

This subroutine 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.

Parameters

function
A user-defined error handling procedure (IN)

errhandler
An MPI error handler (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

The MPI standard specifies a **varargs** error handler prototype. A correct user error handler would be coded as:

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

IBM PE 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. The extra *errhandler* arguments can be accessed by using the C **varargs** (or **stdargs**) facility, but programs that do so will not port cleanly to other MPI implementations that might have different additional arguments.

The effective prototype for an error handler in IBM PE MPI is:

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

The additional arguments are:

routine_name
The name of the MPI routine in which the error occurred

flag Set to **true** if *badval* is meaningful, otherwise set to **false**.

badval
The incorrect integer or long 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

MPI not initialized

MPI already finalized

Null function not allowed
function cannot be NULL.

Related information

- MPI_ERRHANDLER_FREE
- MPI_ERRHANDLER_GET
- MPI_ERRHANDLER_SET

MPI_Errhandler_f2c

Returns a C handle to an error handler.

C synopsis

```
#include <mpi.h>
MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler);
```

Description

This function does not have C++ or Fortran bindings. MPI_Errhandler_f2c returns a C handle to an error handler. If *errhandler* is a valid Fortran handle to an error handler, MPI_Errhandler_f2c returns a valid C handle to that same error handler. If *errhandler* is not a valid Fortran handle, MPI_Errhandler_f2c returns a non-valid C handle. The converted handle is returned as the function's value. There is no error detection or return code.

Parameters

errhandler
The error handler (handle) (IN)

Related information

- MPI_Errhandler_c2f

MPI_ERRHANDLER_FREE, MPI_Errhandler_free

Marks an error handler for deallocation.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::Errhandler::Free();
```

Fortran synopsis

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

Description

This subroutine marks *errhandler* for deallocation and sets it (*errhandler*) to MPI_ERRHANDLER_NULL. Actual deallocation occurs when all communicators associated with the error handler have been deallocated or have had new error handlers attached.

Parameters

errhandler
An MPI error handler (handle) (INOUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid error handler

MPI not initialized

MPI already finalized

Related information

- MPI_ERRHANDLER_CREATE

MPI_ERRHANDLER_GET, MPI_Errhandler_get

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' or USE MPI
MPI_ERRHANDLER_GET(INTEGER COMM,INTEGER ERRHANDLER,INTEGER IERROR)
```

Description

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

Parameters

comm
A communicator (handle) (IN)

errhandler
The MPI error handler currently associated with *comm* (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid communicator

MPI not initialized

MPI already finalized

Related information

- MPI_ERRHANDLER_CREATE
- MPI_ERRHANDLER_SET

MPI_ERRHANDLER_SET, MPI_Errhandler_set

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' or USE MPI
MPI_ERRHANDLER_SET(INTEGER COMM, INTEGER ERRHANDLER, INTEGER IERROR)
```

Description

This subroutine 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.

Parameters

comm
A communicator (handle) (IN)

errhandler
A new MPI error handler for *comm* (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

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 nonstandard 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 be used only 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 does not 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 either a file error handler (created with MPI_FILE_CREATE_ERRHANDLER) or a window error handler (created with MPI_WIN_CREATE_ERRHANDLER). The 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_CREATE
- MPI_ERRHANDLER_GET

MPI_ERROR_CLASS, MPI_Error_class

Returns the error class for the corresponding error code.

C synopsis

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

C++ synopsis

```
#include mpi.h
int MPI::Get_error_class(int errorcode);
```

Fortran synopsis

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

Description

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

This is a list of the predefined error classes.

Error class
Description

MPI_ERR_ACCESS
permission denied

MPI_ERR_AMODE
error related to the *amode* passed to MPI_FILE_OPEN

MPI_ERR_ARG
non-valid argument

MPI_ERR_ASSERT
non-valid *assert* argument

MPI_ERR_BAD_FILE
non-valid file name (the path name is too long, for example)

MPI_ERR_BASE
non-valid *base* argument

MPI_ERR_BUFFER
non-valid buffer pointer

MPI_ERR_COMM
non-valid communicator

MPI_ERR_CONVERSION
An error occurred in a user-supplied data conversion function.

MPI_ERR_COUNT
non-valid count argument

MPI_ERR_DIMS
non-valid dimension argument

MPI_ERR_DISP
non-valid *disp* argument

MPI_ERR_DUP_DATAREP
Conversion functions could not be registered because a previously-defined data representation was passed to MPI_REGISTER_DATAREP.

MPI_ERR_FILE
non-valid file handle

MPI_ERR_FILE_EXISTS
file exists

MPI_ERR_FILE_IN_USE
File operation could not be completed because the file is currently opened by some task.

MPI_ERR_GROUP
non-valid group

MPI_ERR_IN_STATUS
error code is in status

MPI_ERR_INFO
Info object is not valid

MPI_ERR_INFO_NOKEY
Info key is not defined

MPI_ERR_INFO_VALUE
info value is not valid

MPI_ERR_INTERN
internal MPI error

MPI_ERR_IO
other I/O error

MPI_ERR_LASTCODE
last standard error code

MPI_ERR_LOCKTYPE
non-valid *locktype* argument

MPI_ERR_NO_SPACE
Not enough space

MPI_ERR_NO_SUCH_FILE
File does not exist

MPI_ERR_NOT_SAME
Collective argument is not identical on all tasks.

MPI_ERR_OP
non-valid operation

MPI_ERR_OTHER
known error not provided

MPI_ERR_PENDING
pending request

MPI_ERR_QUOTA
quota exceeded

MPI_ERR_RANK
non-valid rank

MPI_ERR_READ_ONLY
read-only file or file system

MPI_ERR_REQUEST
non-valid request (handle)

MPI_ERR_RMA_CONFLICT
conflicting accesses to window

MPI_ERR_RMA_SYNC
incorrect synchronization of RMA calls

MPI_ERR_ROOT
non-valid root

MPI_ERR_SIZE
non-valid *size* argument

MPI_ERR_TAG
non-valid tag argument

MPI_ERR_TOPOLOGY
non-valid topology

MPI_ERR_TRUNCATE
Message truncated on receive.

MPI_ERR_TYPE
non-valid data type argument

MPI_ERR_UNKNOWN
unknown error

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 only sequential access.

MPI_ERR_WIN
non-valid *win* argument

MPI_SUCCESS

Parameters

errorcode
The predefined or user-created error code returned by an MPI subroutine (IN)

errorclass
The predefined or user-defined error class for *errorcode* (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

For IBM PE MPI, see the *IBM Parallel Environment Runtime Edition: Messages*, which provides a list 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.

In general, the subroutine return code and the error message associated with it provide more specific information than the error class does.

This subroutine can also return new error classes that are defined by a user application. The meaning of such classes is determined entirely by the user who creates them. User-defined error classes will be found only on user-created error codes.

Errors

MPI not initialized

MPI already finalized

Related information

- MPI_ADD_ERROR_CLASS
- MPI_ADD_ERROR_CODE
- MPI_ERROR_STRING

MPI_ERROR_STRING, MPI_Error_string

Returns the error string for a given error code.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::Get_error_string(int errorcode, char* string, int& resultlen);
```

Fortran synopsis

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

Description

This subroutine 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**.

This subroutine returns an empty string (all spaces in Fortran, "" in C and C++) for any user-defined error code or error class, unless the the user provides a string using MPI_ADD_ERROR_STRING.

Parameters

errorcode
The error code returned by an MPI routine (IN)

string
The error message for the **errorcode** (OUT)

resultlen
The character length of **string** (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid error code
The *errorcode* is not defined.

MPI not initialized

MPI already finalized

Related information

- MPI_ADD_ERROR_STRING
- MPI_ERROR_CLASS

MPI_EXSCAN, MPI_Exscan

Performs a prefix reduction on data distributed across the group.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::Intracomm::Exscan(const void* sendbuf, void* recvbuf, int count,
                            const MPI::Datatype& datatype, const MPI::Op& op) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_EXSCAN(CHOICE SENDBUF, CHOICE RECVBUF, INTEGER COUNT,
    INTEGER DATATYPE, INTEGER OP, INTEGER COMM, INTEGER IERROR)
```

Description

Use this subroutine to perform a prefix reduction operation on data distributed across a group. The value in *recvbuf* on the task with rank 0 is undefined, and *recvbuf* is not significant on task 0. The value in *recvbuf* on the task with rank 1 is defined as the value in *sendbuf* on the task with rank 0. For tasks with rank $i > 1$, 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 to *i-1* inclusive. The type of operations supported, their semantics, and the constraints on send and receive buffers, are as for MPI_REDUCE.

The *in place* option for intracommunicators is specified by passing **MPI_IN_PLACE** in the *sendbuf* argument. In this case, the input data is taken from the receive buffer, and replaced by the output data. The receive buffer on rank 0 is not changed by this operation.

The parameter *op* may be a predefined reduction operation or a user-defined function, created using MPI_OP_CREATE. This is a list of predefined reduction operations:

Operation
Definition

MPI_BAND
Bitwise AND

MPI_BOR
Bitwise OR

MPI_BXOR
Bitwise XOR

MPI_LAND
Logical AND

MPI_LOR
Logical OR

MPI_LXOR
Logical XOR

MPI_MAX
Maximum value

MPI_MAXLOC
Maximum value and location

MPI_MIN
Minimum value

MPI_MINLOC
Minimum value and location

MPI_PROD
Product

MPI_SUM
Sum

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator occur in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

Parameters

sendbuf
The starting address of the send buffer (choice) (IN)

recvbuf
The starting address of the receive buffer (choice) (OUT)

count
The number of elements in the input buffer (integer) (IN)

datatype
The data type of elements in the input buffer (handle) (IN)

op The reduction operation (handle) (IN)

comm
The intra-communicator (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

As for MPI_SCAN, MPI does not specify which tasks can call the reduction operation, only that the result be correctly computed. In particular, note that the task with rank 1 need not call the MPI_Op, because all it needs to do is to receive the value from the task with rank 0. However, all tasks, even the tasks with ranks 0 and 1, must provide the same op.

Errors

Fatal errors:

Invalid count
count < **0**

Invalid datatype

Type not committed

Invalid op

Invalid communicator

Unequal message lengths

Invalid use of MPI_IN_PLACE

MPI not initialized

MPI already finalized

Develop mode error if:

Inconsistent op

Inconsistent datatype

Inconsistent message length

Related information

- MPI_REDUCE
- MPI_SCAN

MPI_File_c2f

Translates a C file handle into a Fortran handle to the same file.

C synopsis

```
#include <mpi.h>
MPI_Fint MPI_File_c2f(MPI_File file);
```

Description

This function does not have C++ or Fortran bindings. MPI_File_c2f translates a C file handle into a Fortran handle to the same file. This function maps a null handle into a null handle and a non-valid handle into a non-valid handle. The converted handle is returned as the function's value. There is no error detection or return code.

Parameters

file
The file (handle) (IN)

Related information

- MPI_File_f2c

MPI_FILE_CALL_ERRHANDLER, MPI_File_call_errhandler

Calls the error handler assigned to the file with the error code supplied.

C synopsis

```
#include <mpi.h>
int MPI_File_call_errhandler (MPI_File fh, int errorcode);
```

C++ synopsis

```
#include mpi.h
void MPI::File::Call_errhandler(int errorcode) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_CALL_ERRHANDLER(INTEGER FH, INTEGER ERRORCODE, INTEGER IERROR)
```

Description

This subroutine calls the error handler assigned to the file with the error code supplied.

Parameters

fh The file with the error handler (handle) (IN)

errorcode
The error code (integer) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_FILE_CALL_ERRHANDLER returns MPI_SUCCESS in C and C++ and the same value in IERROR if the error handler was successfully called (assuming the error handler itself is not fatal).

The default error handler for files is MPI_ERRORS_RETURN. Thus, calling MPI_FILE_CALL_ERRHANDLER will be transparent if the default error handler has not been changed for this file or on the parent before the file was created. When a predefined error handler is used on *fh*, the error message printed by IBM PE MPI is a specific IBM PE MPI error message that will indicate the error code that is passed in. You cannot force IBM PE MPI to issue a caller-chosen predefined error by passing its error code to this subroutine.

Error handlers should not be called recursively with MPI_FILE_CALL_ERRHANDLER. Doing this can create a situation where an infinite recursion is created. This can occur if MPI_FILE_CALL_ERRHANDLER is called inside an error handler.

Error codes and classes are associated with a task, so they can be used in any error handler. An error handler should be prepared to deal with any error code it is given. Furthermore, it is good practice to call an error handler only with the appropriate error codes. For example, file errors would normally be sent to the file error handler.

Errors

Invalid error code
The *errorcode* is not defined.

Invalid file handle

MPI not initialized

MPI already finalized

Related information

- MPI_ERRHANDLER_FREE
- MPI_FILE_CREATE_ERRHANDLER
- MPI_FILE_GET_ERRHANDLER
- MPI_FILE_SET_ERRHANDLER

MPI_FILE_CLOSE, MPI_File_close

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);
```

C++ synopsis

```
#include mpi.h
void MPI::File::Close();
```

Fortran synopsis

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

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.

Parameters

fh The file handle of the file to be closed (handle) (INOUT)

IERROR
The Fortran return code. It is always the last argument.

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_DELETE
- MPI_FILE_OPEN
- MPI_FINALIZE

MPI_FILE_CREATE_ERRHANDLER, MPI_File_create_errhandler

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);
```

C++ synopsis

```
#include mpi.h
static MPI::Errhandler MPI::File::Create_errhandler,
     (MPI::File::Errhandler_fn* function);
```

Fortran synopsis

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

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.

Parameters

function
A user defined file error handling procedure (IN)

errhandler
An MPI error handler (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

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,...){}
```

IBM PE 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. The extra *errhandler* arguments can be accessed by using the C **varargs** (or **stdargs**) facility, but programs that do so will not port cleanly to other MPI implementations that might have different additional arguments.

The effective prototype for an error handler in IBM PE MPI is:

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

The additional arguments are:

routine_name
The name of the MPI routine in which the error occurred.

flag Set to **true** if *badval* is meaningful, set to **false** if not.

badval
The incorrect 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_ERRHANDLER_FREE
- MPI_FILE_CALL_ERRHANDLER
- MPI_FILE_GET_ERRHANDLER
- MPI_FILE_SET_ERRHANDLER

MPI_FILE_DELETE, MPI_File_delete

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);
```

C++ synopsis

```
#include mpi.h
static void MPI::File::Delete(const char* filename, const MPI::Info& info);
```

Fortran synopsis

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

Description

This subroutine 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.

Parameters

filename
The name of the file to be deleted (string) (IN)

info
An Info object specifying file hints (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

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_f2c

Returns a C handle to a file.

C synopsis

```
#include <mpi.h>
MPI_File MPI_File_f2c(MPI_Fint file);
```

Description

This function does not have C++ or Fortran bindings. MPI_File_f2c returns a C handle to a file. If *file* is a valid Fortran handle to a file, MPI_File_f2c returns a valid C handle to that same file. If *file* is set to the Fortran value MPI_FILE_NULL, MPI_File_f2c returns the equivalent null C handle. If *file* is not a valid Fortran handle, MPI_File_f2c returns a non-valid C handle. The converted handle is returned as the function's value. There is no error detection or return code.

Parameters

file
The file (handle) (IN)

Related information

- MPI_File_c2f

MPI_FILE_GET_AMODE, MPI_File_get_amode

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);
```

C++ synopsis

```
#include mpi.h
int MPI::File::Get_amode() const;
```

Fortran synopsis

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

Description

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

Parameters

fh The file handle (handle) (IN)

amode
The file access mode used to open the file (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

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

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);
```

C++ synopsis

```
#include mpi.h
bool MPI::File::Get_atomicity() const;
```

Fortran synopsis

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

Description

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

Parameters

fh The file handle (handle) (IN)

flag
TRUE if atomic mode, FALSE if nonatomic mode (logical) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

The atomic mode is set to FALSE by default when the file is first 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_SET_ATOMICITY

MPI_FILE_GET_BYTE_OFFSET, MPI_File_get_byte_offset

Allows conversion of an offset.

C synopsis

```
#include <mpi.h>
int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset,
                             MPI_Offset *disp);
```

C++ synopsis

```
#include mpi.h
MPI::Offset MPI::File::Get_byte_offset(const MPI::Offset disp) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_GET_BYTE_OFFSET(INTEGER FH, INTEGER(KIND=MPI_OFFSET_KIND) OFFSET,
                         INTEGER(KIND=MPI_OFFSET_KIND) DISP, INTEGER IERROR)
```

Description

This subroutine allows conversion of an offset, expressed as a number of elementary data types from the file displacement and within the file view, to an absolute number of bytes from the beginning of the file.

Parameters

fh The file handle (handle) (IN)

offset
The offset (integer) (IN)

disp
The absolute byte position of *offset* (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

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 offset (MPI_ERR_FILE)
offset is not a valid offset.

Related information

- MPI_FILE_OPEN
- MPI_FILE_SET_VIEW

MPI_FILE_GET_ERRHANDLER, MPI_File_get_errhandler

Retrieves the error handler currently associated with a file handle.

C synopsis

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

C++ synopsis

```
#include mpi.h
MPI::Errhandler MPI::File::Get_errhandler() const;
```

Fortran synopsis

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

Description

If *fh* is MPI_FILE_NULL, 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, 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.

Parameters

fh A file handle or MPI_FILE_NULL (handle) (IN)

errhandler
The error handler currently associated with *fh* or the current default file error handler (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

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_ERRHANDLER_FREE
- MPI_FILE_CALL_ERRHANDLER
- MPI_FILE_CREATE_ERRHANDLER

- MPI_FILE_SET_ERRHANDLER

MPI_FILE_GET_GROUP, MPI_File_get_group

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);
```

C++ synopsis

```
#include mpi.h
MPI::Group MPI::File::Get_group() const;
```

Fortran synopsis

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

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* using MPI_GROUP_FREE.

Parameters

fh The file handle (handle) (IN)

group
The group that opened the file handle (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

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_GROUP_FREE

MPI_FILE_GET_INFO, MPI_File_get_info

Returns a new Info object.

C synopsis

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

C++ synopsis

```
#include mpi.h
MPI::Info MPI::File::Get_info() const;
```

Fortran synopsis

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

Description

This subroutine creates a new Info object containing the file hints in effect for the file referred to by **fh**, and returns its handle in *info_used*.

Use the MPI_INFO_FREE subroutine to free *info_used*.

Parameters

fh The file handle (handle) (IN)

info_used
The new Info object (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

You can specify file hints using the *info* parameter of these subroutines: MPI_FILE_OPEN, MPI_FILE_SET_INFO, and MPI_FILE_SET_VIEW.

If the user does not specify any file hints, MPI will assign default values to file hints it supports.

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_INFO

- MPI_FILE_SET_VIEW
- MPI_INFO_FREE

MPI_FILE_GET_POSITION, MPI_File_get_position

Returns the current position of the individual file pointer relative to the current file view.

C synopsis

```
#include <mpi.h>
int MPI_File_get_position(MPI_File fh,MPI_Offset *offset);
```

C++ synopsis

```
#include mpi.h
MPI::Offset MPI::File::Get_position() const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_GET_POSITION(INTEGER FH, INTEGER(KIND=MPI_OFFSET_KIND) OFFSET,
                      INTEGER IERROR)
```

Description

This subroutine returns, in *offset*, the current position of the individual file pointer relative to the current file view, in elementary data type units.

Parameters

fh The file handle (handle) (IN).

offset
The offset of the individual file pointer (integer) (OUT).

IERROR
The Fortran return code. It is always the last argument.

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.

MPI_FILE_GET_POSITION_SHARED, MPI_File_get_position_shared

Returns the current position of the shared file pointer relative to the current file view.

C synopsis

```
#include <mpi.h>
int MPI_File_get_position_shared(MPI_File fh,MPI_Offset *offset);
```

C++ synopsis

```
#include mpi.h
MPI::Offset MPI::File::Get_position_shared() const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_GET_POSITION_SHARED(INTEGER FH, INTEGER(KIND=MPI_OFFSET_KIND) OFFSET,
                             INTEGER IERROR)
```

Description

This subroutine returns, in *offset*, the current position of the shared file pointer relative to the current file view, in elementary data type units.

Parameters

fh The file handle (handle) (IN).

offset
The offset of the shared file pointer (integer) (OUT).

IERROR
The Fortran return code. It is always the last argument.

Notes

All tasks in the file group must use the same file view. MPI does not verify that file views are identical.

The position returned may already be inaccurate at the time the subroutine returns if other tasks are concurrently making calls that alter the shared file pointer. It is the user's responsibility to ensure that there are no race conditions between calls to this subroutine and other calls that may alter the shared file pointer.

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.

MPI_FILE_GET_SIZE, MPI_File_get_size

Retrieves the current file size.

C synopsis

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

C++ synopsis

```
#include mpi.h
MPI::Offset MPI::File::Get_size() const;
```

Fortran synopsis

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

Description

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

Parameters

fh The file handle (handle) (IN)

size
The size of the file in bytes (long long) (OUT)

IERROR
The Fortran return code. It is always the last argument.

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_IWRITE_AT
- MPI_FILE_SET_SIZE
- MPI_FILE_WRITE_AT
- MPI_FILE_WRITE_AT_ALL

MPI_FILE_GET_TYPE_EXTENT, MPI_File_get_type_extent

Retrieves the extent of a data type.

C synopsis

```
#include <mpi.h>
int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype,
                             MPI_Aint *extent);
```

C++ synopsis

```
#include mpi.h
MPI::Aint MPI::File::Get_type_extent(const MPI::Datatype& datatype) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_GET_TYPE_EXTENT (INTEGER FH, INTEGER DATATYPE,
                          INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT,
                          INTEGER IERROR)
```

Description

This subroutine retrieves (in *extent*) the extent of *datatype* in the current data representation associated with the open file referred to by *fh*.

Parameters

fh The file handle (handle) (IN)

datatype
The data type (handle) (IN)

extent
The data type extent (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

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.

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

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

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

Related information

- MPI_REGISTER_DATAREP

MPI_FILE_GET_VIEW, MPI_File_get_view

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);
```

C++ synopsis

```
#include mpi.h
void MPI::File::Get_view(MPI::Offset& disp,MPI::Datatype& etype,
    MPI::Datatype& filetype, char* datarep) const;
```

Fortran synopsis

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

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 data type 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 using MPI_TYPE_FREE. Freeing the MPI_Datatype reference returned by MPI_FILE_GET_VIEW invalidates only this reference.

Parameters

fh The file handle (handle) (IN)

disp
The displacement (long long) (OUT)

etype
The elementary data type (handle) (OUT).

filetype
The file type (handle) (OUT).

datarep
The data representation (string) (OUT).

IERROR
The Fortran return code. It is always the last argument.

Notes

1. 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.

2. An MPI type constructor, such as MPI_TYPE_CONTIGUOUS, creates a data type object within MPI and gives a handle for that object to the caller. This handle represents one reference to the object. In IBM PE MPI, the MPI data types obtained with calls to MPI_TYPE_GET_VIEW are new handles for the existing data type objects. The number of handles (references) given to the user is tracked by a reference counter in the object. MPI cannot discard a data type 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 aatype 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 bbtype. Because both handles identify the same data type 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 aatype and bbtype. 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, MPI_File_iread

Performs a nonblocking read operation.

C synopsis

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

C++ synopsis

```
#include mpi.h
MPI::Request MPI::File::Iread(void* buf, int count, const MPI::Datatype& datatype);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_IREAD (INTEGER FH, CHOICE BUF, INTEGER COUNT,
                INTEGER DATATYPE, INTEGER REQUEST, INTEGER IERROR)
```

Description

This subroutine is the nonblocking version of MPI_FILE_READ. It performs the same function as MPI_FILE_READ, except it returns immediately and stores a request handle in *request*. 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 with a completion subroutine call. Completion of the request guarantees that the read operation is complete.

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

If the completion subroutine is associated with multiple requests, it returns when all requests complete successfully or when the first I/O request fails. In the latter case, 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 determines the outcome of the entire completion subroutine, whether the error is on a file request or a communication request. In other words, the error handler associated with the first failing request is triggered.

Parameters

fh The file handle (handle) (INOUT).

buf
The initial address of the buffer (choice) (OUT).

count
The number of elements in the buffer (integer) (IN).

datatype
The data type of each buffer element (handle) (IN).

request
The request object (handle) (OUT).

IERROR
The Fortran return code. It is always the last argument.

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.

Passing MPI_STATUS_IGNORE for the completion subroutine's *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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

For more information, see "MPI_FILE_READ, MPI_File_read" on page 235.

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 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 data type.

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.

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

Errors returned by the completion subroutine (MPI error class):

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

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

Read conversion error (MPI_ERR_CONVERSION)
The conversion attempted during the read operation failed.

Related information

- MPI_CANCEL
- MPI_FILE_READ
- MPI_TEST
- MPI_WAIT

MPI_FILE_IREAD_AT, MPI_File_iread_at

Performs a nonblocking read operation using an explicit offset.

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);
```

C++ synopsis

```
#include mpi.h
MPI::Request MPI::File::Iread_at(MPI::Offset offset, void* buf,
     int count, const MPI::Datatype& datatype);
```

Fortran synopsis

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

Description

This subroutine is the nonblocking version of MPI_FILE_READ_AT. It performs the same function as MPI_FILE_READ_AT, except it returns immediately and stores a request handle in *request*. 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 with a completion subroutine call, such as MPI_TEST, MPI_WAIT, or one of the other MPI test or wait functions. 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 subroutine's *status* argument. If an error occurs during the read operation, the error is returned by the completion subroutine through its return value or in the appropriate element of the *array_of_statuses* argument.

If the completion subroutine is associated with multiple requests, it returns when all requests complete successfully or when the first I/O request fails. In the latter case, 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 determines the outcome of the entire completion subroutine, whether the error is on a file request or a communication request. In other words, the error handler associated with the first failing request is triggered.

Parameters

fh The file handle (handle) (IN).

offset
The file offset (long long) (IN).

buf
The initial address of buffer (choice) (OUT).

count
The number of elements in the buffer (integer) (IN).

datatype
The data type of each buffer element (handle) (IN).

request
The request object (handle) (OUT).

IERROR
The Fortran return code. It is always the last argument.

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 completion subroutine's *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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

For more information, see "MPI_FILE_READ_AT, MPI_File_read_at" on page 243.

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 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 data type.

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 not a valid offset.

Errors returned by the completion subroutine (MPI error class):

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

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

Read conversion error (MPI_ERR_CONVERSION)
The conversion attempted during the read operation failed.

Related information

- MPI_CANCEL
- MPI_FILE_READ_AT
- MPI_TEST
- MPI_WAIT

MPI_FILE_IREAD_SHARED, MPI_File_iread_shared

Performs a nonblocking read operation using the shared file pointer.

C synopsis

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

C++ synopsis

```
#include mpi.h
MPI::Request MPI::File::Iread_shared(void* buf, int count,
         const MPI::Datatype& datatype);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_IREAD_SHARED (INTEGER FH, CHOICE BUF, INTEGER COUNT,
                INTEGER DATATYPE, INTEGER REQUEST, INTEGER IERROR)
```

Description

This subroutine is the nonblocking version of MPI_FILE_READ_SHARED. It performs the same function as MPI_FILE_READ_SHARED, except it returns immediately and stores a request handle in *request*. 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 with a completion subroutine call, such as MPI_TEST, MPI_WAIT, or one of the other MPI test or wait functions. Completion of the request guarantees that the read operation is complete.

When MPI_FILE_IREAD_SHARED completes, the actual number of bytes read is stored in the completion subroutine'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 element of the *array_of_statuses* argument.

If the completion subroutine is associated with multiple requests, it returns when all requests complete successfully or when the first I/O request fails. In the latter case, 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 determines the outcome of the entire completion subroutine, whether the error is on a file request or a communication request. In other words, the error handler associated with the first failing request is triggered.

Parameters

fh The file handle (handle) (INOUT).

buf
The initial address of the buffer (choice) (OUT).

count
The number of elements in the buffer (integer) (IN).

datatype
The data type of each buffer element (handle) (IN).

request
The request object (handle) (OUT).

IERROR
The Fortran return code. It is always the last argument.

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.

Passing MPI_STATUS_IGNORE for the completion subroutine's *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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

For more information, see "MPI_FILE_READ_SHARED, MPI_File_read_shared" on page 259.

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 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 data type.

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

Uncommitted datatype (MPI_ERR_TYPE)
datatype must be committed.

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

Errors returned by the completion subroutine (MPI error class):

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

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

Read conversion error (MPI_ERR_CONVERSION)
The conversion attempted during the read operation failed.

Related information

- MPI_CANCEL
- MPI_FILE_READ_SHARED
- MPI_TEST
- MPI_WAIT

MPI_FILE_IWRITE, MPI_File_iwrite

Performs a nonblocking write operation.

C synopsis

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

C++ synopsis

```
#include mpi.h
MPI::Request MPI::File::Iwrite(const void* buf, int count,
          const MPI::Datatype& datatype);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_IWRITE(INTEGER FH,CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,
                INTEGER REQUEST,INTEGER IERROR)
```

Description

This subroutine is the nonblocking version of MPI_FILE_WRITE. It performs the same function as MPI_FILE_WRITE, except it returns immediately and stores a request handle in *request*. 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 with a completion subroutine call, such as MPI_TEST, MPI_WAIT, or one of the other MPI test or wait functions.

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

If the completion subroutine is associated with multiple requests, it returns when all requests complete successfully or when the first I/O request fails. In the latter case, 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 determines the outcome of the entire completion subroutine whether the error is on a file request or a communication request. In other words, the error handler associated with the first failing request is triggered.

Parameters

fh The file handle (handle) (INOUT).

buf
The initial address of the buffer (choice) (IN).

count
The number of elements in the buffer (integer) (IN).

datatype
The data type of each buffer element (handle) (IN).

request
The request object (handle) (OUT).

IERROR
The Fortran return code. It is always the last argument.

Notes

Completion of the request does not guarantee that the data has been written to the storage device (or devices). In particular, written data may still be present in system buffers. However, it guarantees that the memory buffer can be safely reused.

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.

Passing MPI_STATUS_IGNORE for the completion subroutine's *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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

For more information, see "MPI_FILE_WRITE, MPI_File_write" on page 276.

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 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 data type.

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.

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

Errors returned by the completion subroutine (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 lseek failed (MPI_ERR_IO)
An internal **lseek** operation failed.

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

Write conversion error (MPI_ERR_CONVERSION)
The conversion attempted during the write operation failed.

Related information

- MPI_CANCEL
- MPI_FILE_WRITE
- MPI_TEST
- MPI_WAIT

MPI_FILE_IWRITE_AT, MPI_File_iwrite_at

Performs a nonblocking write operation using an explicit offset.

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);
```

C++ synopsis

```
#include mpi.h
MPI::Request MPI::File::Iwrite_at(MPI::Offset offset, const void* buf,
      int count, const MPI::Datatype& datatype);
```

Fortran synopsis

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

Description

This subroutine is the nonblocking version of MPI_FILE_WRITE_AT. It performs the same function as MPI_FILE_WRITE_AT, except it returns immediately and stores a request handle in *request*. 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 with a completion subroutine call, such as MPI_TEST, MPI_WAIT, or one of the other MPI test or wait functions.

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

If the completion subroutine is associated with multiple requests, it returns when all requests complete successfully or when the first I/O request fails. In the latter case, 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 determines the outcome of the entire completion subroutine whether the error is on a file request or a communication request. In other words, the error handler associated with the first failing request is triggered.

Parameters

fh The file handle (handle) (INOUT).

offset
The file offset (long long) (IN).

buf
The initial address of buffer (choice) (IN).

count
The number of elements in buffer (integer) (IN).

datatype
The data type of each buffer element (handle) (IN).

request
The request object (handle) (OUT).

IERROR
The Fortran return code. It is always the last argument.

Notes

Completion of the request does not guarantee that the data has been written to the storage device (or devices). In particular, written data may still be present in system buffers. However, it guarantees that the memory buffer can be safely reused.

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 completion subroutine's *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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

For more information, see "MPI_FILE_WRITE_AT, MPI_File_write_at" on page 285.

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 data type.

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 not a valid offset.

Errors returned by the completion subroutine (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.

Write conversion error (MPI_ERR_CONVERSION)
The conversion attempted during the write operation failed.

Related information

- MPI_CANCEL
- MPI_TEST
- MPI_WAIT
- MPI_FILE_WRITE_AT

MPI_FILE_IWRITE_SHARED, MPI_File_iwrite_shared

Performs a nonblocking write operation using the shared file pointer.

C synopsis

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

C++ synopsis

```
#include mpi.h
MPI::Request MPI::File::Iwrite_shared(const void* buf, int count,
          const MPI::Datatype& datatype);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_IWRITE_SHARED (INTEGER FH, CHOICE BUF, INTEGER COUNT,
                INTEGER DATATYPE, INTEGER REQUEST, INTEGER IERROR)
```

Description

This subroutine is the nonblocking version of MPI_FILE_WRITE_SHARED. It performs the same function as MPI_FILE_WRITE_SHARED, except it returns immediately and stores a request handle in *request*. 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 with a completion subroutine call, such as MPI_TEST, MPI_WAIT, or one of the other MPI test or wait functions.

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

If the completion subroutine is associated with multiple requests, it returns when all requests complete successfully or when the first I/O request fails. In the latter case, 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 determines the outcome of the entire completion subroutine, whether the error is on a file request or a communication request. In other words, the error handler associated with the first failing request is triggered.

Parameters

fh The file handle (handle) (INOUT).

buf
The initial address of the buffer (choice) (IN).

count
The number of elements in the buffer (integer) (IN).

datatype
The data type of each buffer element (handle) (IN).

request
The request object (handle) (OUT).

IERROR
The Fortran return code. It is always the last argument.

Notes

Completion of the request does not guarantee that the data has been written to the storage device (or devices). In particular, written data may still be present in system buffers. However, it guarantees that the memory buffer can be safely reused.

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.

Passing MPI_STATUS_IGNORE for the completion subroutine's *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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

For more information, see "MPI_FILE_WRITE_SHARED, MPI_File_write_shared" on page 301.

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 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 data type.

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

Uncommitted datatype (MPI_ERR_TYPE)
datatype must be committed.

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

Errors returned by the completion subroutine (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 lseek failed (MPI_ERR_IO)
An internal **lseek** operation failed.

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

Write conversion error (MPI_ERR_CONVERSION)
The conversion attempted during the write operation failed.

Related information

- MPI_CANCEL
- MPI_FILE_WRITE_SHARED
- MPI_TEST
- MPI_WAIT

MPI_FILE_OPEN, MPI_File_open

Opens a file.

C synopsis

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

C++ synopsis

```
#include mpi.h
static MPI::File MPI::File::Open(const MPI::Intracomm& comm, const char* filename,
            int amode, const MPI::Info& info);
```

Fortran synopsis

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

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. Though most I/O can be done with the default file view, much of the optimization MPI-IO can provide depends on the effective use of appropriate user-defined file views.

MPI_FILE_OPEN is a collective operation. *comm* must be a valid intra-communicator. Values specified for *amode* by all participating tasks must be identical. Participating tasks must refer to the same file through their own instances of *filename*.

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

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

MPI_MODE_UNIQUE_OPEN allows IBM PE MPI-IO to use an optimization that is not possible when a file may be shared by other jobs. The optimization is more likely to help with read performance than with write performance. If it is known that the file will not be shared, try using MPI_MODE_UNIQUE_OPEN.

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 appear only once.

File hints can be associated with a file when it is being opened. **MPI_FILE_OPEN** ignores the hint value if it is not valid. Any Info *key, value* pair the user provides will either be accepted or ignored. There will never be an error returned or change in semantic as a result of a hint.

File Hints

This is a list of the supported *file hints* or **info** *keys*. There are restrictions on which file hints can be used simultaneously, and on when and under what circumstances a hint value can be set or used. In general, if a hint is specified in a circumstance where it is not supported, it will be ignored. Use the **MPI_FILE_GET_INFO** routine to verify the set of hints in effect for a file.

Hint name
Description

filename

- **Default value:** The file name specified by **MPI_FILE_OPEN**.
- **Valid values:** Not applicable
- **Subroutines you can use to set it:** This hint cannot be set with an Info object. The hint value is taken from the file name specified by the *filename* parameter of the **MPI_FILE_OPEN** subroutine.
- **Value consistency requirement:** Not applicable
- **Notes:** This hint can be retrieved only by the **MPI_FILE_GET_INFO** subroutine.

file_perm

- **Default value: 644** if specified by **MPI_FILE_OPEN** with a mode of **MPI_MODE_CREATE**; otherwise, the value reflects the access permissions associated with the file.
- **Valid values:** Octal values **000** through **777**
- **Subroutines you can use to set it: MPI_FILE_OPEN**
- **Value consistency requirement:** Consistent values are required at all participating tasks
- **Notes:**

  This hint can be specified in the Info object when calling **MPI_FILE_OPEN** with the mode **MPI_MODE_CREATE** enabled in order to set the access permissions of the file to be created.

  This hint can also be retrieved when the **MPI_FILE_GET_INFO** subroutine is called, and its value then represents the access permissions associated with the file.

  The hint value is expressed as a three-digit octal number, similar to the format used by the numeric mode of the **chmod** shell command. The value is the sum of the following values:

  400 permits read by owner

  200 permits write by owner

  100 permits execute by owner

  040 permits read by group

  020 permits write by group

010 permits execute by group

004 permits read by others

002 permits write by others

001 permits execute by others

IBM_io_buffer_size

- **Default value:** number of bytes corresponding to 16 file blocks
- **Valid values:** any positive value up to 128 MB. The size can be expressed either as a number of bytes, or as a number of kilobytes (KB), using the letter **K** or **k** as the suffix, or as a number of megabytes (MB), using the letter **M** or **m** as the suffix
- **Subroutines you can use to set it: MPI_FILE_OPEN**, or, if there is no pending I/O operation: **MPI_FILE_SET_INFO** or **MPI_FILE_SET_VIEW**
- **Value consistency requirement:** Consistent values are required at all participating tasks
- **Notes:** This hint specifies the size that is used to stripe the file across I/O agents in round-robin style. In general, one I/O agent is associated with each MPI task. However, if the **MP_IONODEFILE** environment variable or the **poe -ionodefile** command is used, one I/O agent is associated with each task running on any of the nodes specified in the file referred to by **MP_IONODEFILE** or **-ionodefile**.

  IBM PE MPI rounds up the number of bytes specified to an integral number of file blocks. The size of a file block is returned in the *st_blksize* field of the *struct stat* argument passed to the **stat** or **fstat** routine. For example, if *IBM_io_buffer_size* has a value of **23240**, all data access operations on a file that belongs to a GPFS™ file system with a block size of 16KB will be performed as follows: the first 32KB of the file will be handled by the first I/O agent, all data access operations to the next 32KB of the file will be handled by the second I/O agent, and so on.

  Increasing the *IBM_io_buffer_size* value can improve performance when using large files, where large refers to hundreds of megabytes, particularly if the program uses collective data access operations.

  This hint applies only when the *IBM_largeblock_io* hint has a value of **false**. When *IBM_largeblock_io* is enabled, data striping across I/O agents is not performed.

IBM_largeblock_io

- **Default value: false**
- **Valid values: switchable**, **true**, **false**
- **Subroutines you can use to set it: MPI_FILE_OPEN**, or, if there is no pending I/O operation: **MPI_FILE_SET_INFO** or **MPI_FILE_SET_VIEW**
- **Value consistency requirement:** Consistent values are required at all participating tasks
- **Notes:** Examples of applications that should benefit from using this hint are those in which each task accesses a large, contiguous chunk of the file, or in which the file is divided into distinct regions that are accessed by separate tasks. The hint value **switchable**, which can be specified only when calling **MPI_FILE_OPEN**, indicates that the hint value can be toggled between **true** and **false** until the file is closed. If the hint is specified as **switchable** on the call to **MPI_FILE_OPEN**, the hint value is set to **false** and can be toggled on calls to **MPI_FILE_SET_INFO** or **MPI_FILE_SET_VIEW**. If the hint is specified as **true** or **false** on the call

to **MPI_FILE_OPEN**, the hint value cannot be changed by either **MPI_FILE_SET_INFO** or **MPI_FILE_SET_VIEW**. This hint can be used only if all tasks are being used for I/O: either the **MP_IONODEFILE** environment variable is not set, or it specifies a file that lists all nodes on which the application is running. For JFS files, this hint can be set only if all tasks are running on the same node.

IBM_sparse_access

Lets you specify the future file access pattern of the application for the associated file. Specifically, you can specify whether the file access requests from participating tasks are sparse (the value is set to **true**) or dense (the value is set to **false**).

- **Default value: false**
- **Valid values: true**, **false**
- **Subroutines you can use to set it: MPI_FILE_OPEN**, **MPI_FILE_SET_INFO**, **MPI_FILE_SET_VIEW**
- **Value consistency requirement:** Consistent values are required at all participating tasks
- **Notes:** In cases where each single MPI collective read or write operation touches most of the sections in a fairly large region of a file, this hint will not help. In cases where the entire range of each collective read or write is relatively small or, if the range is large and only widely-separated bits of the file are touched, this hint may improve performance. In this context, *section* refers to either the default or explicitly set *IBM_io_buffer_size* and *large* begins somewhere near (*IBM_io_buffer_size* multiplied by sizeof(MPI_ COMM_WORLD)).

Parameters

comm
The communicator (handle) (IN)

filename
The name of the file to open (string) (IN)

amode
The file access mode (integer) (IN)

info
The Info object (handle) (IN)

fh The new file handle (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

When you open a file, the atomicity is set to **false**.

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

Parameter consistency checking is performed only 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.

MPI-IO in IBM PE MPI is targeted to the IBM General Parallel File System (GPFS) for production use. File access through MPI-IO normally requires that a single GPFS file system image be available across all tasks of an MPI job. IBM PE MPI with MPI-IO can be used for program development on any other file system that supports a POSIX interface (AFS™, JFS, or NFS) as long as all tasks run on a single node or workstation. This is not expected to be a useful model for production use of MPI-IO. IBM PE MPI can be used without all nodes on a single file system image by using the **MP_IONODEFILE** environment variable. See *IBM Parallel Environment Runtime Edition: Operation and Use* for information about **MP_IONODEFILE**.

When MPI-IO is used correctly, a file name will refer to the same file system at every task. 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.

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.

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

For AFS, 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 **MP_FILE_OPEN** is done with mode **MPI_MODE_UNIQUE_OPEN**, file locking is not done.

Because the actual file I/O is carried out by agent threads spread across all tasks of the job, hand-coded *optimizations* based on an assumption that I/O occurs at the task making the MPI-IO call are more likely to do harm than good. If this kind of optimization is done, set the *IBM_largeblock_io* hint to **true**. This will shut off the shipping of data to agents and cause file I/O to be done by the calling task.

In an environment that uses dynamic process management, **MP_FILE_OPEN** can take an input communicator that covers 2 or more worlds. The additional connection among these worlds, created by an **MP_FILE_OPEN**, is undone by the **MPI_FILE_CLOSE** (similar to using **MPI_COMM_DISCONNECT** on a communicator that spans worlds).

Errors

Fatal errors:

MPI not initialized

MPI already finalized

Invalid communicator
comm is not a valid communicator.

Can't use an inter-communicator
comm is an inter-communicator.

Conflicting collective operations on communicator

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

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.

Invalid file handle

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

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 tasks.

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

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

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

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 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_PREALLOCATE, MPI_File_preallocate

Ensures that storage space is allocated for the first *size* bytes of the file associated with *fh*.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::File::Preallocate(MPI::Offset size);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_PREALLOCATE(INTEGER FH, INTEGER(KIND=MPI_ADDRESS_KIND)
         SIZE, INTEGER IERROR)
```

Description

This subroutine ensures that storage space is allocated for the first *size* bytes of the file associated with *fh*. MPI_FILE_PREALLOCATE is collective; all tasks in the group must pass identical values for *size*. Regions of the file that have previously been written are unaffected. For newly-allocated regions of the file, MPI_FILE_PREALLOCATE has the same effect as writing undefined data. If *size* is larger than the current file size, the file size increases to size. If size is less than or equal to the current file size, the file size is unchanged. The treatment of file pointers, pending nonblocking accesses, and file consistency, is the same as with MPI_FILE_SET_SIZE. If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous to call this subroutine.

Parameters

fh The file handle (handle) (INOUT)

size
The size to preallocate the file (integer) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

GPFS handles this operation efficiently; this may not be true for other file systems.

Errors

Fatal errors:

MPI not initialized

MPI already finalized

Returning Errors:

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

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.

Invalid file size (MPI_ERR_ARG)
size is a negative value.

Locally detected error occurred on another task (MPI_ERR_OTHER)
A local parameter check failed on one or more other tasks.

Inconsistent file sizes (MPI_ERR_NOT_SAME)
The local *size* is not consistent with the file size on other tasks.

Consistency error occurred on another task (MPI_ERR_OTHER)
A consistency check failed on one or more other tasks.

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

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

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

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

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

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

Related information

- MPI_FILE_SET_SIZE

MPI_FILE_READ, MPI_File_read

Reads from a file.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::File::Read(void* buf, int count, const MPI::Datatype& datatype,
       MPI::Status& status);
```

```
#include mpi.h
void MPI::File::Read(void* buf, int count, const MPI::Datatype& datatype);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_READ(INTEGER FH,CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,
              INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
```

Description

This subroutine tries to read, from the file referred to by *fh*, *count* items of type *datatype* into the buffer *buf*, starting at the current file location as determined by the value of the individual file pointer. The call returns only when data is available in *buf*. *status* contains the number of bytes successfully read. You can use accessor functions MPI_GET_COUNT and MPI_GET_ELEMENTS 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.

Parameters

fh The file handle (handle) (INOUT).

buf
The initial address of the buffer (choice) (OUT).

count
The number of elements in the buffer (integer) (IN).

datatype
The data type of each buffer element (handle) (IN).

status
The status object (Status) (OUT).

IERROR
The Fortran return code. It is always the last argument.

Notes

Passing MPI_STATUS_IGNORE for the *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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):

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 data type.

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.

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

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

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

Read conversion error (MPI_ERR_CONVERSION)
The conversion attempted during the read operation failed.

Invalid status ignore value

Related information

- MPI_FILE_IREAD
- MPI_FILE_READ_ALL
- MPI_FILE_READ_ALL_BEGIN
- MPI_FILE_READ_ALL_END

MPI_FILE_READ_ALL, MPI_File_read_all

Reads from a file collectively.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::File::Read_all(void* buf, int count, const MPI::Datatype& datatype,
          MPI::Status& status);
```

```
#include mpi.h
void MPI::File::Read_all(void* buf, int count, const MPI::Datatype& datatype);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_READ_ALL(INTEGER FH,CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,
             INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
```

Description

This subroutine is the collective version of MPI_FILE_READ. It performs the same function as MPI_FILE_READ. The number of bytes actually read by the calling task is stored 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.

Parameters

fh The file handle (handle) (INOUT).

buf
The initial address of the buffer (choice) (OUT).

count
The number of elements in the buffer (integer) (IN).

datatype
The data type of each buffer element (handle) (IN).

status
The status object (Status) (OUT).

IERROR
The Fortran return code. It is always the last argument.

Notes

Passing MPI_STATUS_IGNORE for the *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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

For more information, see “MPI_FILE_READ, MPI_File_read” on page 235.

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 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 data type.

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

Uncommitted datatype (MPI_ERR_TYPE)
datatype must be committed.

Pending split collective data access operation (MPI_ERR_OTHER)
A collective data access operation is attempted while there is a pending split collective data access operation on the same file handle.

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

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

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

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

Read conversion error (MPI_ERR_CONVERSION)
The conversion attempted during the read operation failed.

Invalid status ignore value

Related information

- MPI_FILE_IREAD
- MPI_FILE_READ
- MPI_FILE_READ_ALL_BEGIN
- MPI_FILE_READ_ALL_END

MPI_FILE_READ_ALL_BEGIN, MPI_File_read_all_begin

Initiates a split collective read operation from a file.

C synopsis

```
#include <mpi.h>
int MPI_File_read_all_begin (MPI_File fh, void *buf, int count,
                             MPI_Datatype datatype);
```

C++ synopsis

```
#include mpi.h
void MPI::File::Read_all_begin(void* buf, int count, const MPI::Datatype& datatype);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_READ_ALL_BEGIN (INTEGER FH, CHOICE BUF, INTEGER COUNT,
                         INTEGER DATATYPE, INTEGER IERROR)
```

Description

This subroutine initiates a split collective operation that, when completed by the matching end subroutine (MPI_FILE_READ_ALL_END), produces an equivalent result to that of the collective routine MPI_FILE_READ_ALL.

This subroutine returns immediately.

Begin operations are collective over the group of tasks that participated in the collective open and follow the ordering rules for collective operations.

As with any nonblocking data access operation, the user must not use the buffer passed to a begin subroutine while the operation is outstanding. The operation must be completed with an end subroutine before it is safe to access, reuse, or free the buffer.

Parameters

fh The file handle (handle) (INOUT).

buf
The initial address of the buffer (choice) (OUT).

count
The number of elements in the buffer (integer) (IN).

datatype
The data type of each buffer element (handle) (IN).

IERROR
The Fortran return code. It is always the last argument.

Notes

Only one split collective operation can be active on any given file handle.

A file handle that is being used in a split collective operation cannot be used for a blocking collective operation.

Split collective operations do not inter-operate with the corresponding regular collective operation. For example, in a single collective read operation, an

MPI_FILE_READ_ALL on one task does not match an MPI_FILE_READ_ALL_BEGIN and MPI_FILE_READ_ALL_END pair on another task.

The begin and end subroutines must be called from the same thread.

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 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 data type.

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.

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

Pending split collective data access operation (MPI_ERR_OTHER)
A collective data access operation is attempted while there is a pending split collective data access operation on the same file handle.

Related information

- MPI_FILE_READ_ALL
- MPI_FILE_READ_ALL_END

MPI_FILE_READ_ALL_END, MPI_File_read_all_end

Completes a split collective read operation from a file.

C synopsis

```
#include <mpi.h>
int MPI_File_read_all_end(MPI_File fh,void *buf,MPI_Status *status);
```

C++ synopsis

```
#include mpi.h
void MPI::File::Read_all_end(void* buf);
```

```
#include mpi.h
void MPI::File::Read_all_end(void* buf, MPI::Status& status);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_READ_ALL_END(INTEGER FH,CHOICE BUF,INTEGER STATUS(MPI_STATUS_SIZE),
                      INTEGER IERROR)
```

Description

This subroutine ends a split collective operation that was initiated by the matching begin subroutine (MPI_FILE_READ_ALL_BEGIN). Combined with the begin routine, it produces an equivalent result to that of the collective routine MPI_FILE_READ_ALL.

End calls are collective over the group of tasks that participated in the collective open and follow the ordering rules for collective operations. Each end call matches the preceding begin call for the same collective operation. When an end call is made, exactly one unmatched begin call for the same operation must precede it.

This subroutine returns only when the data to be read is available in the user's buffer. The call does not wait for accesses from other tasks associated with the file handle to have data available in their user's buffers.

The number of bytes actually read by the calling task is stored in *status*.

Parameters

fh The file handle (handle) (INOUT).

buf
The initial address of the buffer (choice) (OUT).

status
The status object (Status) (OUT).

IERROR
The Fortran return code. It is always the last argument.

Notes

Only one split collective operation can be active on any given file handle.

A file handle that is being used in a split collective operation cannot be used for a blocking collective operation.

Split collective operations do not inter-operate with the corresponding regular collective operation. For example, in a single collective read operation, an MPI_FILE_READ_ALL on one task does not match an MPI_FILE_READ_ALL_BEGIN and MPI_FILE_READ_ALL_END pair on another task.

The begin and end subroutines must be called from the same thread.

Passing MPI_STATUS_IGNORE for the *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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 write-only mode.

No pending split collective data access operation (MPI_ERR_OTHER)
The end phase of a split collective data access operation is attempted while there is no pending split collective data access operation.

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

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

Read conversion error (MPI_ERR_CONVERSION)
The conversion attempted during the read operation failed.

Invalid status ignore value

Related information

- MPI_FILE_READ_ALL
- MPI_FILE_READ_ALL_BEGIN

MPI_FILE_READ_AT, MPI_File_read_at

Reads from a file using an explicit 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);
```

C++ synopsis

```
#include mpi.h
void MPI::File::Read_at(MPI::Offset offset, void* buf, int count,
   const MPI::Datatype& datatype);
```

```
#include mpi.h
void MPI::File::Read_at(MPI::Offset offset, void* buf, int count,
   const MPI::Datatype& datatype, MPI::Status& status);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
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)
```

Description

This subroutine tries to read, from the file referred to by *fh*, *count* items of type *datatype* into the buffer *buf*, starting at *offset*, relative to the current view. The call returns only when data is available in *buf*. *status* contains the number of bytes successfully read. You can use accessor functions MPI_GET_COUNT and MPI_GET_ELEMENTS 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.

Parameters

fh The file handle (handle) (IN).

offset
The file offset (long long) (IN).

buf
The initial address of the buffer (choice) (OUT).

count
The number of elements in the buffer (integer) (IN).

datatype
The data type of each buffer element (handle) (IN).

status
The status object (Status) (OUT).

IERROR
The Fortran return code. It is always the last argument.

Notes

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 causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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 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 data type.

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 not a valid offset.

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

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

Read conversion error (MPI_ERR_CONVERSION)
The conversion attempted during the read operation failed.

Invalid status ignore value

Related information

- MPI_FILE_IREAD_AT
- MPI_FILE_READ_AT_ALL
- MPI_FILE_READ_AT_ALL_BEGIN
- MPI_FILE_READ_AT_ALL_END

MPI_FILE_READ_AT_ALL, MPI_File_read_at_all

Reads from a file collectively using an explicit offset.

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);
```

C++ synopsis

```
#include mpi.h
void MPI::File::Read_at_all(MPI::Offset offset, void* buf, int count,
       const MPI::Datatype& datatype);
```

```
#include mpi.h
void MPI::File::Read_at_all(MPI::Offset offset, void* buf, int count,
       const MPI::Datatype& datatype, MPI::Status& status);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
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)
```

Description

This subroutine is the collective version of MPI_FILE_READ_AT. It performs the same function as MPI_FILE_READ_AT. 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.

Parameters

fh The file handle (handle) (IN).

offset
The file offset (long long) (IN).

buf
The initial address of the buffer (choice) (OUT).

count
The number of elements in the buffer (integer) (IN).

datatype
The data type of each buffer element (handle) (IN).

status
The status object (Status) (OUT).

IERROR
The Fortran return code. It is always the last argument.

Notes

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 causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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

For more information, see "MPI_FILE_READ_AT, MPI_File_read_at" on page 243.

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 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 data type.

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 not a valid offset.

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

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

Pending split collective data access operation (MPI_ERR_OTHER)
A collective data access operation is attempted while there is a pending split collective data access operation on the same file handle.

Read conversion error (MPI_ERR_CONVERSION)
The conversion attempted during the read operation failed.

Invalid status ignore value

Related information

- MPI_FILE_IREAD_AT
- MPI_FILE_READ_AT
- MPI_FILE_READ_AT_ALL_BEGIN
- MPI_FILE_READ_AT_ALL_END

MPI_FILE_READ_AT_ALL_BEGIN, MPI_File_read_at_all_begin

Initiates a split collective read operation from a file using an explicit offset.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::File::Read_at_all_begin(MPI::Offset offset, void* buf, int count,
            const MPI::Datatype& datatype);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_READ_AT_ALL_BEGIN(INTEGER FH,INTEGER(KIND=MPI_OFFSET_KIND) OFFSET,
                           CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,
                           INTEGER IERROR)
```

Description

This subroutine initiates a split collective operation that, when completed by the matching end subroutine (MPI_FILE_READ_AT_ALL_END), produces an equivalent result to that of the collective routine MPI_FILE_READ_AT_ALL.

This subroutine returns immediately.

Begin calls are collective over the group of tasks that participated in the collective open and follow the ordering rules for collective operations.

As with any nonblocking data access operation, the user must not use the buffer passed to a begin subroutine while the operation is outstanding. The operation must be completed with an end subroutine before it is safe to access, reuse, or free the buffer.

Parameters

fh The file handle (handle) (IN).

offset
The file offset (integer) (IN).

buf
The initial address of the buffer (choice) (OUT).

count
The number of elements in the buffer (integer) (IN).

datatype
The data type of each buffer element (handle) (IN).

IERROR
The Fortran return code. It is always the last argument.

Notes

Only one split collective operation can be active on any given file handle.

A file handle that is being used in a split collective operation cannot be used for a blocking collective operation.

Split collective operations do not inter-operate with the corresponding regular collective operation. For example, in a single collective read operation, an MPI_FILE_READ_AT_ALL on one task does not match an MPI_FILE_READ_AT_ALL_BEGIN and MPI_FILE_READ_AT_ALL_END pair on another task.

The begin and end subroutines must be called from the same thread.

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 offset (MPI_ERR_ARG)
offset is not a valid offset.

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 data type.

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.

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

Pending split collective data access operation (MPI_ERR_OTHER)
A collective data access operation is attempted while there is a pending split collective data access operation on the same file handle.

Related information

- MPI_FILE_READ_AT_ALL
- MPI_FILE_READ_AT_ALL_END

MPI_FILE_READ_AT_ALL_END, MPI_File_read_at_all_end

Completes a split collective read operation from a file.

C synopsis

```
#include <mpi.h>
int MPI_File_read_at_all_end(MPI_File fh,void *buf,MPI_Status *status);
```

C++ synopsis

```
#include mpi.h
void MPI::File::Read_at_all_end(void *buf, MPI::Status& status);
```

```
#include mpi.h
void MPI::File::Read_at_all_end(void *buf);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_READ_AT_ALL_END(INTEGER FH,CHOICE BUF,
                         INTEGER STATUS(MPI_STATUS_SIZE),
                         INTEGER IERROR)
```

Description

This subroutine ends a split collective operation that was initiated by the matching begin subroutine (MPI_FILE_READ_AT_ALL_BEGIN). Combined with the begin subroutine, it produces an equivalent result to that of the collective routine MPI_FILE_READ_AT_ALL.

End calls are collective over the group of tasks that participated in the collective open and follow the ordering rules for collective operations. Each end operation matches the preceding begin call for the same collective operation. When an end call is made, exactly one unmatched begin call for the same operation must precede it.

This subroutine returns only when the data to be read is available in the user's buffer. The operation does not wait for accesses from other tasks associated with the file handle to have data available in their user's buffers.

The number of bytes actually read by the calling task is stored in *status*.

Parameters

fh The file handle (handle) (IN).

buf
The initial address of the buffer (choice) (OUT).

status
The status object (Status) (OUT).

IERROR
The Fortran return code. It is always the last argument.

Notes

Only one split collective operation can be active on any given file handle.

A file handle that is being used in a split collective operation cannot be used for a blocking collective operation.

Split collective operations do not inter-operate with the corresponding regular collective operation. For example, in a single collective read operation, an MPI_FILE_READ_AT_ALL on one task does not match an MPI_FILE_READ_AT_ALL_BEGIN and MPI_FILE_READ_AT_ALL_END pair on another task.

The begin and end subroutines must be called from the same thread.

Passing MPI_STATUS_IGNORE for the *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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

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

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

Read conversion error (MPI_ERR_CONVERSION)
The conversion attempted during the read operation failed.

Invalid status ignore value

Related information

- MPI_FILE_READ_AT_ALL
- MPI_FILE_READ_AT_ALL_BEGIN

MPI_FILE_READ_ORDERED, MPI_File_read_ordered

Reads from a file collectively using the shared file pointer.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::File::Read_ordered(void* buf, int count, const MPI::Datatype& datatype,
        MPI::Status& status);
```

```
#include mpi.h
void MPI::File::Read_ordered(void* buf, int count, const MPI::Datatype& datatype);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_READ_ORDERED(INTEGER FH,CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,
              INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
```

Description

This subroutine is a collective version of MPI_FILE_READ_SHARED. It performs the same function as MPI_FILE_READ_SHARED, except that it behaves as if the operations were initiated by the participating tasks in rank order. The number of bytes actually read by the calling task is stored in *status*. The call returns only when data requested by the calling task is available in *buf*, disregarding data accesses from other tasks associated with file handle *fh*.

Parameters

fh The file handle (handle) (INOUT).

buf
The initial address of the buffer (choice) (OUT).

count
The number of elements in the buffer (integer) (IN).

datatype
The data type of each buffer element (handle) (IN).

status
The status object (Status) (OUT).

IERROR
The Fortran return code. It is always the last argument.

Notes

Passing MPI_STATUS_IGNORE for the *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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

For more information, see "MPI_FILE_READ_SHARED, MPI_File_read_shared" on page 259.

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 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 data type.

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

Uncommitted datatype (MPI_ERR_TYPE)
datatype must be committed.

Pending split collective data access operation (MPI_ERR_OTHER)
A collective data access operation is attempted while there is a pending split collective data access operation on the same file handle.

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

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

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

Read conversion error (MPI_ERR_CONVERSION)
The conversion attempted during the read operation failed.

Invalid status ignore value

Related information

- MPI_FILE_IREAD_SHARED
- MPI_FILE_READ_ORDERED_BEGIN
- MPI_FILE_READ_ORDERED_END
- MPI_FILE_READ_SHARED

MPI_FILE_READ_ORDERED_BEGIN, MPI_File_read_ordered_begin

Initiates a split collective read operation from a file using the shared file pointer.

C synopsis

```
#include <mpi.h>
int MPI_File_read_ordered_begin(MPI_File fh, void *buf, int count,
                                MPI_Datatype datatype);
```

C++ synopsis

```
#include mpi.h
void MPI::File::Read_ordered_begin(void* buf, int count,
       const MPI::Datatype& datatype);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_READ_ORDERED_BEGIN (INTEGER FH, CHOICE BUF, INTEGER COUNT,
                             INTEGER DATATYPE, INTEGER IERROR)
```

Description

This subroutine initiates a split collective operation that, when completed by the matching end subroutine (MPI_FILE_READ_ORDERED_END), produces an equivalent result to that of the collective routine MPI_FILE_READ_ORDERED.

This subroutine returns immediately.

Begin calls are collective over the group of tasks that participated in the collective open and follow the ordering rules for collective operations.

As with any nonblocking data access operation, the user must not use the buffer passed to a begin subroutine while the operation is outstanding. The operation must be completed with an end subroutine before it is safe to access, reuse, or free the buffer.

Parameters

fh The file handle (handle) (INOUT).

buf
The initial address of the buffer (choice) (OUT).

count
The number of elements in the buffer (integer) (IN).

datatype
The data type of each buffer element (handle) (IN).

IERROR
The Fortran return code. It is always the last argument.

Notes

Only one split collective operation can be active on any given file handle.

A file handle that is being used in a split collective operation cannot be used for a blocking collective operation.

Split collective operations do not inter-operate with the corresponding regular collective operation. For example, in a single collective read operation, an MPI_FILE_READ_ORDERED on one task does not match an MPI_FILE_READ_ORDERED_BEGIN and MPI_FILE_READ_ORDERED_END pair on another task.

The begin and end subroutines must be called from the same thread.

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 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 data type.

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

Uncommitted datatype (MPI_ERR_TYPE)
datatype must be committed.

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

Pending split collective data access operation (MPI_ERR_OTHER)
A collective data access operation is attempted while there is a pending split collective data access operation on the same file handle.

Related information

- MPI_FILE_READ_ORDERED
- MPI_FILE_READ_ORDERED_END

MPI_FILE_READ_ORDERED_END, MPI_File_read_ordered_end

Completes a split collective read operation from a file using the shared file pointer.

C synopsis

```
#include <mpi.h>
int MPI_File_read_ordered_end(MPI_File fh,void *buf,MPI_Status *status);
```

C++ synopsis

```
#include mpi.h
void MPI::File::Read_ordered_end(void* buf, MPI::Status& status);
```

```
#include mpi.h
void MPI::File::Read_ordered_end(void* buf);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_READ_ORDERED_END(INTEGER FH,CHOICE BUF,INTEGER STATUS(MPI_STATUS_SIZE),
                          INTEGER IERROR)
```

Description

This subroutine ends a split collective operation that was initiated by the matching begin subroutine (MPI_FILE_READ_ORDERED_BEGIN). Combined with the begin subroutine, it produces an equivalent result to that of the collective routine MPI_FILE_READ_ORDERED.

End calls are collective over the group of tasks that participated in the collective open and follow the ordering rules for collective operations. Each end call matches the preceding begin call for the same collective operation. When an end call is made, exactly one unmatched begin call for the same operation must precede it.

This subroutine returns only when the data to be read is available in the user's buffer. The call does not wait for accesses from other tasks associated with the file handle to have data available in their user's buffers.

The number of bytes actually read by the calling task is stored in *status*.

Parameters

fh The file handle (handle) (INOUT).

buf
The initial address of the buffer (choice) (OUT).

status
The status object (Status) (OUT).

IERROR
The Fortran return code. It is always the last argument.

Notes

Only one split collective operation can be active on any given file handle.

A file handle that is being used in a split collective operation cannot be used for a blocking collective operation.

Split collective operations do not inter-operate with the corresponding regular collective operation. For example, in a single collective read operation, an MPI_FILE_READ_ORDERED on one task does not match an MPI_FILE_READ_ORDERED_BEGIN and MPI_FILE_READ_ORDERED_END pair on another task.

The begin and end subroutines must be called from the same thread.

Passing MPI_STATUS_IGNORE for the *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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

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

Read conversion error (MPI_ERR_CONVERSION)
The conversion attempted during the read operation failed.

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

No pending split collective data access operation (MPI_ERR_OTHER)
The end phase of a split collective data access operation is attempted while there is no pending split collective data access operation.

Invalid status ignore value

Related information

- MPI_FILE_READ_ORDERED
- MPI_FILE_READ_ORDERED_BEGIN

MPI_FILE_READ_SHARED, MPI_File_read_shared

Reads from a file using the shared file pointer.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::File::Read_shared(void* buf, int count, const MPI::Datatype& datatype);
```

```
#include mpi.h
void MPI::File::Read_shared(void* buf, int count, const MPI::Datatype& datatype,
       MPI::Status& status);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_READ_SHARED(INTEGER FH,CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,
              INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
```

Description

This subroutine tries to read, from the file referred to by *fh*, *count* items of type *datatype* into the buffer *buf*, starting at the current file location as determined by the value of the shared file pointer. The call returns only when data is available in *buf*. *status* contains the number of bytes successfully read. You can use accessor functions MPI_GET_COUNT and MPI_GET_ELEMENTS 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.

Parameters

fh The file handle (handle) (INOUT).

buf
The initial address of the buffer (choice) (OUT).

count
The number of elements in the buffer (integer) (IN).

datatype
The data type of each buffer element (handle) (IN).

status
The status object (Status) (OUT).

IERROR
The Fortran return code. It is always the last argument.

Notes

Passing MPI_STATUS_IGNORE for the *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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):

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 data type.

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

Uncommitted datatype (MPI_ERR_TYPE)
datatype must be committed.

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

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

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

Read conversion error (MPI_ERR_CONVERSION)
The conversion attempted during the read operation failed.

Invalid status ignore value

Related information

- MPI_FILE_IREAD_SHARED
- MPI_FILE_READ_ORDERED
- MPI_FILE_READ_ORDERED_BEGIN
- MPI_FILE_READ_ORDERED_END

MPI_FILE_SEEK, MPI_File_seek

Sets a file pointer.

C synopsis

```
#include <mpi.h>
int MPI_File_seek (MPI_File fh,MPI_Offset offset, int whence);
```

C++ synopsis

```
#include mpi.h
void MPI::File::Seek(MPI::Offset offset, int whence);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_SEEK (INTEGER FH, INTEGER(KIND=MPI_OFFSET_KIND) OFFSET,
               INTEGER WHENCE, INTEGER IERROR)
```

Description

This subroutine updates the individual file pointer according to *whence*, which can have one of the following values:

MPI_SEEK_CUR
: the file pointer is set to its current position plus *offset*

MPI_SEEK_END
: the file pointer is set to the end of the file position plus *offset*

MPI_SEEK_SET
: the file pointer is set to *offset*

The offset can be negative, which allows to seek backwards. However, it is erroneous to seek to a negative position in the current file view. A seek past the end of the file is valid.

Parameters

fh The file handle (handle) (INOUT).

offset
: The file offset (integer) (IN).

whence
: The update mode (state) (IN).

IERROR
: The Fortran return code. It is always the last argument.

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 offset (MPI_ERR_ARG)
offset is not a valid offset.

Invalid whence (MPI_ERR_ARG)
whence must be MPI_SEEK_CUR, MPI_SEEK_END, or MPI_SEEK_SET

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

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

Related information

- MPI_FILE_READ
- MPI_FILE_SEEK_SHARED
- MPI_FILE_WRITE

MPI_FILE_SEEK_SHARED, MPI_File_seek_shared

Sets a shared file pointer.

C synopsis

```
#include <mpi.h>
int MPI_File_seek_shared(MPI_File fh,MPI_Offset offset,int whence);
```

C++ synopsis

```
#include mpi.h
void MPI::File::Seek_shared(MPI::Offset offset, int whence);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_SEEK_SHARED(INTEGER FH, INTEGER(KIND=MPI_OFFSET_KIND) OFFSET,
                     INTEGER WHENCE, INTEGER IERROR)
```

Description

This subroutine updates the shared file pointer according to *whence*, which can have one of the following values:

MPI_SEEK_CUR
: the file pointer is set to its current position plus *offset*

MPI_SEEK_END
: the file pointer is set to the end of the file position plus *offset*

MPI_SEEK_SET
: the file pointer is set to *offset*

This is a collective operation. All participating tasks must specify the same values for *offset* and *whence*. The offset can be negative, which allows to seek backwards. However, it is erroneous to seek to a negative position in the current file view. A seek past the end of the file is valid.

Parameters

fh The file handle (handle) (INOUT).

offset
: The file offset (integer) (IN).

whence
: The update mode (state) (IN).

IERROR
: The Fortran return code. It is always the last argument.

Notes

The position set may already be outdated at the time the subroutine returns if other tasks are concurrently making calls that alter the shared file pointer. It is the user's responsibility to ensure that there are no race conditions between calls to this subroutine and other calls that may alter the shared file pointer.

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 offset (MPI_ERR_ARG)
offset is not a valid offset.

Invalid whence (MPI_ERR_ARG)
whence must be MPI_SEEK_CUR, MPI_SEEK_END, or MPI_SEEK_SET

Inconsistent offsets (MPI_ERR_NOT_SAME)
Local *offset* is not consistent with neighbor's offset.

Inconsistent whences (MPI_ERR_NOT_SAME)
Local *whence* is not consistent with neighbor's whence.

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

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

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

Related information

- MPI_FILE_READ_SHARED
- MPI_FILE_SEEK
- MPI_FILE_WRITE_SHARED

MPI_FILE_SET_ATOMICITY, MPI_File_set_atomicity

Modifies the current atomicity mode for an opened file.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::File::Set_atomicity(bool flag);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_SET_ATOMICITY (INTEGER FH,LOGICAL FLAG,INTEGER IERROR)
```

Description

This subroutine modifies the current atomicity mode for an opened file. This is a collective operation. All participating tasks must specify the same value for *flag*.

Parameters

fh The file handle (handle) (INOUT)

flag
Set to **true** if atomic mode, **false** if nonatomic mode (logical) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

When you open a file, the atomicity is set to **false**.

Reading or writing a file in atomic mode can have a substantial negative impact on performance. Use atomic mode only when it is essential.

Parameter consistency checking is performed only if the environment variable **MP_EUIDEVELOP** is set to **yes**. If this variable is set and the flags specified are not identical, the error **Inconsistent flags** 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 file handle (MPI_ERR_FILE)
fh is not a valid file handle.

Inconsistent flags (MPI_ERR_NOT_SAME)
Local *flag* is not consistent with neighbor's flag.

Related information

- MPI_FILE_GET_ATOMICITY
- MPI_FILE_OPEN

MPI_FILE_SET_ERRHANDLER, MPI_File_set_errhandler

Associates a new error handler to a file.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::File::Set_errhandler(const MPI::Errhandler& errhandler);
```

Fortran synopsis

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

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, this subroutine associates the error handler *errhandler* with the file referred to by *fh*.

Parameters

fh The valid file handle (handle) (IN)

errhandler
The new error handler for the opened file (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

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.

For information about a predefined error handler for C++, see *IBM Parallel Environment Runtime Edition: MPI Programming Guide*.

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_ERRHANDLER_FREE
- MPI_FILE_CALL_ERRHANDLER
- MPI_FILE_CREATE_ERRHANDLER
- MPI_FILE_GET_ERRHANDLER

MPI_FILE_SET_INFO, MPI_File_set_info

Specifies new hints for an open file.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::File::Set_info(const MPI::Info& info);
```

Fortran synopsis

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

Description

This subroutine associates legitimate file-related hints contained in the Info argument with the file referred to by *fh*. This is a collective operation. If I/O operations are pending on *fh*, hint values are ignored.

MPI_FILE_SET_INFO ignores the hint value if it is not valid. Any Info *key, value* pair the user provides will either be accepted or ignored. There will never be an error returned or change in semantic as a result of a hint.

See subroutine "MPI_FILE_OPEN, MPI_File_open" on page 226 for a list of supported file hints.

Parameters

fh The file handle (handle) (INOUT)

info
The Info object (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

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

Expands or truncates an open file.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::File::Set_size(MPI::Offset size);
```

Fortran synopsis

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

Description

MPI_FILE_SET_SIZE is a collective operation that lets you 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*, 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.

Parameters

fh The file handle (handle) (INOUT)

size
The requested size of the file after truncation or expansion (long long) (IN).

IERROR
The Fortran return code. It is always the last argument.

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 performed only 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 tasks.

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 tasks.

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

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

Related information

- MPI_FILE_GET_SIZE
- MPI_FILE_PREALLOCATE

MPI_FILE_SET_VIEW, MPI_File_set_view

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);
```

C++ synopsis

```
#include mpi.h
void MPI::File::Set_view(MPI::Offset disp, const MPI::Datatype& etype,
                         const MPI::Datatype& filetype, const char* datarep,
           const MPI::Info& info);
```

Fortran synopsis

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

Description

This subroutine associates a new view defined by **disp**, **etype**, **filetype**, and **datarep** with the open file referred to by *fh*. This is a collective operation. 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 data types. 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 effective use of MPI_FILE_SET_VIEW by each task of a file group can be critical to obtaining the performance benefits of MPI-IO. When the tasks each set a file view that is complementary to the views set by other tasks and use collective MPI-IO operations in conjunction with these views, the MPI library has the information that will allow it to optimize the I/O. Without the information available in the file view settings, fewer opportunities for optimization by MPI-IO exist.

Valid values for *datarep* are:

external32
: States that read and write operations convert all data from and to the **external32** representation that is documented in the MPI-2 standard. The *external32* data representation is not currently supported on Linux running on IBM System x® servers.

internal
: Can be used for I/O operations in a homogeneous or heterogeneous environment. IBM has defined its internal format with the intent that any implementation of MPI provided by IBM can use this format. The *internal* data representation is not currently supported on Linux running on IBM System x servers.

Note: For IBM implementations of MPI, the *internal* data representation is interpreted as one which allows a file generated on one IBM platform to be read on another without discarding precision. The intent of the internal data representation on IBM platforms is essentially *external64*, but because the MPI standard does not currently define *external64*, you cannot be certain that IBM *internal* will exactly match *external64* when, or if, it is defined.

For applications that do not require file portability, use the *native* data representation because *internal* adds data conversion overhead for certain MPI data types. The data types that incur overhead depends on the particular platform's native data representations.

native Should be used in most situations. Data in this representation is stored in a file exactly as it is in memory. This representation is always suitable in a homogeneous MPI environment and does not incur conversion costs.

File hints can be associated with a file when a view is set on it. MPI_FILE_SET_VIEW ignores the hint value if it is not valid. Any Info *key, value* pair the user provides will either be accepted or ignored. There will never be an error returned or change in semantic as a result of a hint.

See "MPI_FILE_OPEN, MPI_File_open" on page 226 for a list of supported file hints.

Parameters

fh The file handle (handle) (IN).

disp
The displacement (long long) (IN).

etype
The elementary data type (handle) (IN).

filetype
The filetype (handle) (IN).

datarep
The data representation (string) (IN).

info
The Info object (handle) (IN).

IERROR
The Fortran return code. It is always the last argument.

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 performed only if the environment variable **MP_EUIDEVELOP** is set to **yes**. If this variable is set and the extents of the elementary data types 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 data type.

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 not a valid 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 tasks.

Inconsistent elementary datatypes (MPI_ERR_NOT_SAME)
Local *etype* extent is not consistent with the elementary data type extent of other tasks.

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

Related information

- MPI_FILE_GET_VIEW

MPI_FILE_SYNC, MPI_File_sync

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);
```

C++ synopsis

```
#include mpi.h
void MPI::File::Sync();
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_SYNC (INTEGER FH,INTEGER IERROR)
```

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 (or devices) 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.

Parameters

fh The file handle (handle) (INOUT)

IERROR
The Fortran return code. It is always the last argument.

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 tasks.

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

MPI_FILE_WRITE, MPI_File_write

Writes to a file.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::File::Write(const void* buf, int count, const MPI::Datatype& datatype);
```

```
#include mpi.h
void MPI::File::Write(const void* buf, int count, const MPI::Datatype& datatype,
        MPI::Status& status);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_WRITE(INTEGER FH,CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,
               INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
```

Description

This subroutine tries to write, into the file referred to by *fh*, *count* items of type *datatype* out of the buffer *buf*, starting at the current file location as determined by the value of the individual file pointer. MPI_FILE_WRITE returns when it is safe to reuse *buf*. *status* contains the number of bytes successfully written. You can use accessor functions MPI_GET_COUNT and MPI_GET_ELEMENTS to extract from *status* the number of items and the number of intrinsic MPI elements successfully written, respectively.

Parameters

fh The file handle (handle) (INOUT).

buf
The initial address of the buffer (choice) (IN).

count
The number of elements in the buffer (integer) (IN).

datatype
The data type of each buffer element (handle) (IN).

status
The status object (Status) (OUT).

IERROR
The Fortran return code. It is always the last argument.

Notes

Return from the call does not guarantee that the data has been written to the storage device (or devices). In particular, written data may still be present in system buffers. However, it guarantees that the memory buffer can be safely reused.

Passing MPI_STATUS_IGNORE for the *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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

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 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 data type.

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.

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.

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

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

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

Write conversion error (MPI_ERR_CONVERSION)
The conversion attempted during the write operation failed.

Invalid status ignore value

Related information

- MPI_FILE_IWRITE
- MPI_FILE_WRITE_ALL
- MPI_FILE_WRITE_ALL_BEGIN
- MPI_FILE_WRITE_ALL_END

MPI_FILE_WRITE_ALL, MPI_File_write_all

Writes to a file collectively.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::File::Write_all(const void* buf, int count,
    const MPI::Datatype& datatype);
```

```
#include mpi.h
void MPI::File::Write_all(const void* buf, int count,
  const MPI::Datatype& datatype, MPI::Status& status);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_WRITE_ALL(INTEGER FH,CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,
                   INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
```

Description

This subroutine is the collective version of MPI_FILE_WRITE. It performs the same function as MPI_FILE_WRITE. MPI_FILE_WRITE_ALL tries to write, into the file referred to by *fh*, *count* items of type *datatype* out of the buffer *buf*, starting at the current file location as determined by the value of the individual file pointer. MPI_FILE_WRITE returns when it is safe to reuse *buf*. *status* contains the number of bytes successfully written. You can use accessor functions MPI_GET_COUNT and MPI_GET_ELEMENTS to extract from *status* the number of items and the number of intrinsic MPI elements successfully written, respectively.

Parameters

fh The file handle (handle) (INOUT).

buf
The initial address of the buffer (choice) (IN).

count
The number of elements in the buffer (integer) (IN).

datatype
The data type of each buffer element (handle) (IN).

status
The status object (Status) (OUT).

IERROR
The Fortran return code. It is always the last argument.

Notes

Return from the call does not guarantee that the data has been written to the storage device (or devices). In particular, written data may still be present in system buffers. However, it guarantees that the memory buffer can be safely reused.

Passing MPI_STATUS_IGNORE for the *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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

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 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 data type.

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

Uncommitted datatype (MPI_ERR_TYPE)
datatype must be committed.

Pending split collective data access operation (MPI_ERR_OTHER)
A collective data access operation is attempted while there is a pending split collective data access operation on the same file handle.

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

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.

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

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

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

Write conversion error (MPI_ERR_CONVERSION)
The conversion attempted during the write operation failed.

Invalid status ignore value

Related information

- MPI_FILE_IWRITE
- MPI_FILE_WRITE
- MPI_FILE_WRITE_ALL_BEGIN
- MPI_FILE_WRITE_ALL_END

MPI_FILE_WRITE_ALL_BEGIN, MPI_File_write_all_begin

Initiates a split collective write operation to a file.

C synopsis

```
#include <mpi.h>
int MPI_File_write_all_begin (MPI_File fh, void *buf, int count,
                              MPI_Datatype datatype);
```

C++ synopsis

```
#include mpi.h
void MPI::File::Write_all_begin(const void* buf, int count,
          const MPI::Datatype& datatype);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_WRITE_ALL_BEGIN (INTEGER FH, CHOICE BUF, INTEGER COUNT,
                          INTEGER DATATYPE, INTEGER IERROR)
```

Description

This subroutine initiates a split collective operation that, when completed by the matching end subroutine (MPI_FILE_WRITE_ALL_END), produces an equivalent result to that of the collective routine MPI_FILE_WRITE_ALL.

This subroutine returns immediately.

Begin calls are collective over the group of tasks that participated in the collective open and follow the ordering rules for collective operations.

As with any nonblocking data access operation, the user must not use the buffer passed to a begin subroutine while the operation is outstanding. The operation must be completed with an end subroutine before it is safe to access, reuse, or free the buffer.

Parameters

fh The file handle (handle) (INOUT).

buf
The initial address of the buffer (choice) (IN).

count
The number of elements in the buffer (integer) (IN).

datatype
The data type of each buffer element (handle) (IN).

IERROR
The Fortran return code. It is always the last argument.

Notes

Only one split collective operation can be active on any given file handle.

A file handle that is being used in a split collective operation cannot be used for a blocking collective operation.

Split collective operations do not inter-operate with the corresponding regular collective operation. For example, in a single collective write operation, an MPI_FILE_WRITE_ALL on one task does not match an MPI_FILE_WRITE_ALL_BEGIN and MPI_FILE_WRITE_ALL_END pair on another task.

The begin and end subroutines must be called from the same thread.

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 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 data type.

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.

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

Pending split collective data access operation (MPI_ERR_OTHER)
A collective data access operation is attempted while there is a pending split collective data access operation on the same file handle.

Related information

- MPI_FILE_WRITE
- MPI_FILE_WRITE_ALL
- MPI_FILE_WRITE_ALL_END

MPI_FILE_WRITE_ALL_END, MPI_File_write_all_end

Completes a split collective write operation to a file.

C synopsis

```
#include <mpi.h>
int MPI_File_write_all_end(MPI_File fh,void *buf,MPI_Status *status);
```

C++ synopsis

```
#include mpi.h
void MPI::File::Write_all_end(void* buf);
```

```
#include mpi.h
void MPI::File::Write_all_end(void* buf, MPI::Status& status);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_WRITE_ALL_END(INTEGER FH,CHOICE BUF,INTEGER STATUS(MPI_STATUS_SIZE)
                       INTEGER IERROR)
```

Description

This subroutine ends a split collective operation that was initiated by the matching begin subroutine (MPI_FILE_WRITE_ALL_BEGIN). Combined with the begin routine, it produces an equivalent result to that of the collective routine MPI_FILE_WRITE_ALL.

End calls are collective over the group of tasks that participated in the collective open and follow the ordering rules for collective operations. Each end call matches the preceding begin call for the same collective operation. When an end call is made, exactly one unmatched begin call for the same operation must precede it.

This subroutine returns only when the user's buffer that contains the data to be written can be modified safely.

The number of bytes actually written by the calling task is stored in *status*.

Parameters

fh The file handle (handle) (INOUT).

buf
The initial address of the buffer (choice) (IN).

status
The status object (Status) (OUT).

IERROR
The Fortran return code. It is always the last argument.

Notes

Return from the call does not guarantee that the data has been written to the storage device (or devices). In particular, written data may still be present in system buffers. However, it guarantees that the memory buffer can be safely reused.

Only one split collective operation can be active on any given file handle.

A file handle that is being used in a split collective operation cannot be used for a blocking collective operation.

Split collective operations do not inter-operate with the corresponding regular collective operation. For example, in a single collective write operation, an MPI_FILE_WRITE_ALL on one task does not match an MPI_FILE_WRITE_ALL_BEGIN and MPI_FILE_WRITE_ALL_END pair on another task.

The begin and end subroutines must be called from the same thread.

Passing MPI_STATUS_IGNORE for the *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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

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

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

No pending split collective data access operation (MPI_ERR_OTHER)
The end phase of a split collective data access operation is attempted while there is no pending split collective data access operation.

Write conversion error (MPI_ERR_CONVERSION)
The conversion attempted during the write operation failed.

Invalid status ignore value

Related information

- MPI_FILE_WRITE
- MPI_FILE_WRITE_ALL
- MPI_FILE_WRITE_ALL_BEGIN

MPI_FILE_WRITE_AT, MPI_File_write_at

Performs a blocking write operation using an explicit 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);
```

C++ synopsis

```
#include mpi.h
void MPI::File::Write_at(MPI::Offset offset, const void* buf,
    int count, const MPI::Datatype& datatype);
```

```
#include mpi.h
void MPI::File::Write_at(MPI::Offset offset, const void* buf,
    int count, const MPI::Datatype& datatype,
    MPI::Status& status);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
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)
```

Description

MPI_FILE_WRITE_AT tries to write into the file referred to by *fh count* items of type *datatype* out of the buffer *buf,* starting at *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 allow you to extract from *status* the number of items and the number of intrinsic MPI elements successfully written, respectively.

Parameters

fh The file handle (handle) (INOUT).

offset
The file offset (long long) (IN).

buf
The initial address of buffer (choice) (IN).

count
The number of elements in buffer (integer) (IN).

datatype
The data type of each buffer element (handle) (IN).

status
The status object (Status) (OUT).

IERROR
The Fortran return code. It is always the last argument.

Notes

Return from the call does not guarantee that the data has been written to the storage device (or devices). In particular, written data may still be present in system buffers. However, it guarantees that the memory buffer can be safely reused.

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 causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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

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 not a valid 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.

Write conversion error (MPI_ERR_CONVERSION)
The conversion attempted during the write operation failed.

Invalid status ignore value

Related information

- MPI_FILE_IWRITE
- MPI_FILE_WRITE_AT_ALL
- MPI_FILE_WRITE_AT_ALL_BEGIN
- MPI_FILE_WRITE_AT_ALL_END

MPI_FILE_WRITE_AT_ALL, MPI_File_write_at_all

Performs a blocking write operation collectively using an explicit offset.

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);
```

C++ synopsis

```
#include mpi.h
void MPI::File::Write_at_all(MPI::Offset offset, const void* buf,
        int count, const MPI::Datatype& datatype);
```

```
#include mpi.h
void MPI::File::Write_at_all(MPI::Offset offset, const void* buf,
        int count, const MPI::Datatype& datatype,
        MPI::Status& status);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
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)
```

Description

This subroutine is the collective version of MPI_FILE_WRITE_AT. The number of bytes actually written by the calling task is stored in *status*. 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 reused.

Parameters

fh The file handle (handle) (INOUT).

offset
The file offset (long long) (IN).

buf
The initial address of buffer (choice) (IN).

count
The number of elements in buffer (integer) (IN).

datatype
The data type of each buffer element (handle) (IN).

status
The status object (Status) (OUT).

IERROR
The Fortran return code. It is always the last argument.

Notes

Return from the call does not guarantee that the data has been written to the storage device (or devices). In particular, written data may still be present in system buffers. However, it guarantees that the memory buffer can be safely reused.

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 causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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

For more information, see "MPI_FILE_WRITE_AT, MPI_File_write_at" on page 285.

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 data type.

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

Uncommitted datatype (MPI_ERR_TYPE)
datatype must be committed.

Pending split collective data access operation (MPI_ERR_OTHER)
A collective data access operation is attempted while there is a pending split collective data access operation on the same file handle.

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 not a valid 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.

Write conversion error (MPI_ERR_CONVERSION)
The conversion attempted during the write operation failed.

Invalid status ignore value

Related information

- MPI_FILE_IWRITE_AT
- MPI_FILE_WRITE_AT
- MPI_FILE_WRITE_AT_ALL_BEGIN
- MPI_FILE_WRITE_AT_ALL_END

MPI_FILE_WRITE_AT_ALL_BEGIN, MPI_File_write_at_all_begin

Initiates a split collective write operation to a file using an explicit offset.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::File::Write_at_all_begin(MPI::Offset offset, const void* buf,
              int count, const MPI::Datatype& datatype);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_WRITE_AT_ALL_BEGIN(INTEGER FH,INTEGER(KIND=MPI_OFFSET_KIND) OFFSET,
                            CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,
                            INTEGER IERROR)
```

Description

This subroutine initiates a split collective operation that, when completed by the matching end subroutine (MPI_FILE_WRITE_AT_ALL_END), produces an equivalent result to that of the collective routine MPI_FILE_WRITE_AT_ALL.

This subroutine returns immediately.

Begin calls are collective over the group of tasks that participated in the collective open and follow the ordering rules for collective operations.

As with any nonblocking data access operation, the user must not use the buffer passed to a begin subroutine while the operation is outstanding. The operation must be completed with an end subroutine before it is safe to access, reuse, or free the buffer.

Parameters

fh The file handle (handle) (INOUT).

offset
The file offset (integer) (IN).

buf
The initial address of the buffer (choice) (IN).

count
The number of elements in the buffer (integer) (IN).

datatype
The data type of each buffer element (handle) (IN).

IERROR
The Fortran return code. It is always the last argument.

Notes

Only one split collective operation can be active on any given file handle.

A file handle that is being used in a split collective operation cannot be used for a blocking collective operation.

Split collective operations do not inter-operate with the corresponding regular collective operation. For example, in a single collective write operation, an MPI_FILE_WRITE_AT_ALL on one task does not match an MPI_FILE_WRITE_AT_ALL_BEGIN and MPI_FILE_WRITE_AT_ALL_END pair on another task.

The begin and end subroutines must be called from the same thread.

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 offset (MPI_ERR_ARG)
offset is not a valid offset.

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 data type.

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.

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

Pending split collective data access operation (MPI_ERR_OTHER)
A collective data access operation is attempted while there is a pending split collective data access operation on the same file handle.

Related information

- MPI_FILE_WRITE
- MPI_FILE_WRITE_AT
- MPI_FILE_WRITE_AT_ALL
- MPI_FILE_WRITE_AT_ALL_END

MPI_FILE_WRITE_AT_ALL_END, MPI_File_write_at_all_end

Completes a split collective write operation to a file using an explicit offset.

C synopsis

```
#include <mpi.h>
int MPI_File_write_at_all_end(MPI_File fh,void *buf,MPI_Status *status);
```

C++ synopsis

```
#include mpi.h
void MPI::File::Write_at_all_end(const void* buf);
```

```
#include mpi.h
void MPI::File::Write_at_all_end(const void* buf, MPI::Status& status);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_WRITE_AT_ALL_END(INTEGER FH,CHOICE BUF,
                          INTEGER STATUS(MPI_STATUS_SIZE),
                          INTEGER IERROR)
```

Description

This subroutine ends a split collective operation that was initiated by the matching begin subroutine (MPI_FILE_WRITE_AT_ALL_BEGIN). Combined with the begin subroutine, it produces an equivalent result to that of the collective routine MPI_FILE_WRITE_AT_ALL.

End calls are collective over the group of tasks that participated in the collective open and follow the ordering rules for collective operations. Each end call matches the preceding begin call for the same collective operation. When an end call is made, exactly one unmatched begin call for the same operation must precede it.

This subroutine returns only when the user's buffer that contains the data to be written can be modified safely.

The number of bytes actually written by the calling task is stored in *status*.

Parameters

fh The file handle (handle) (INOUT).

buf
The initial address of the buffer (choice) (IN).

status
The status object (Status) (OUT).

IERROR
The Fortran return code. It is always the last argument.

Notes

Return from the call does not guarantee that the data has been written to the storage device (or devices). In particular, written data may still be present in system buffers. However, it guarantees that the memory buffer can be safely reused.

Only one split collective operation can be active on any given file handle.

A file handle that is being used in a split collective operation cannot be used for a blocking collective operation.

Split collective operations do not inter-operate with the corresponding regular collective operation. For example, in a single collective write operation, an MPI_FILE_WRITE_AT_ALL on one task does not match an MPI_FILE_WRITE_AT_ALL_BEGIN and MPI_FILE_WRITE_AT_ALL_END pair on another task.

The begin and end subroutines must be called from the same thread.

Passing MPI_STATUS_IGNORE for the *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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

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

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

No pending split collective data access operation (MPI_ERR_OTHER)
The end phase of a split collective data access operation is attempted while there is no pending split collective data access operation.

Write conversion error (MPI_ERR_CONVERSION)
The conversion attempted during the write operation failed.

Invalid status ignore value

Related information

- MPI_FILE_WRITE
- MPI_FILE_WRITE_AT
- MPI_FILE_WRITE_AT_ALL
- MPI_FILE_WRITE_AT_ALL_BEGIN

MPI_FILE_WRITE_ORDERED, MPI_File_write_ordered

Writes to a file collectively using the shared file pointer.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::File::Write_ordered(const void* buf, int count,
         const MPI::Datatype& datatype);
```

```
#include mpi.h
void MPI::File::Write_ordered(const void* buf, int count,
         const MPI::Datatype& datatype,
         MPI::Status& status);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_WRITE_ORDERED(INTEGER FH,CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,
              INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
```

Description

This subroutine is a collective version of MPI_FILE_WRITE_SHARED. It performs the same function as MPI_FILE_WRITE_SHARED, except that it behaves as if the operations were initiated by the participating tasks in rank order. The number of bytes actually written by the calling task is stored in *status*. The call returns only when the calling task can safely reuse *buf*, disregarding data accesses from other tasks associated with file handle *fh*.

Parameters

fh The file handle (handle) (INOUT).

buf
The initial address of the buffer (choice) (IN).

count
The number of elements in the buffer (integer) (IN).

datatype
The data type of each buffer element (handle) (IN).

status
The status object (Status) (OUT).

IERROR
The Fortran return code. It is always the last argument.

Notes

Return from the call does not guarantee that the data has been written to the storage device (or devices). In particular, written data may still be present in system buffers. However, it guarantees that the memory buffer can be safely reused.

Passing MPI_STATUS_IGNORE for the *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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):

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 data type.

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

Uncommitted datatype (MPI_ERR_TYPE)
datatype must be committed.

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.

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

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

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

Write conversion error (MPI_ERR_CONVERSION)
The conversion attempted during the write operation failed.

Invalid status ignore value

Related information

- MPI_FILE_IWRITE_SHARED
- MPI_FILE_WRITE_ORDERED_BEGIN
- MPI_FILE_WRITE_ORDERED_END
- MPI_FILE_WRITE_SHARED

MPI_FILE_WRITE_ORDERED_BEGIN, MPI_File_write_ordered_begin

Initiates a split collective write operation to a file using the shared file pointer.

C synopsis

```
#include <mpi.h>
int MPI_File_write_ordered_begin(MPI_File fh, void *buf, int count,
                                 MPI_Datatype datatype);
```

C++ synopsis

```
#include mpi.h
void MPI::File::Write_ordered_begin(const void* buf, int count,
              const MPI::Datatype& datatype);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_WRITE_ORDERED_BEGIN (INTEGER FH, CHOICE BUF, INTEGER COUNT,
                              INTEGER DATATYPE, INTEGER IERROR)
```

Description

This subroutine initiates a split collective operation that, when completed by the matching end subroutine (MPI_FILE_WRITE_ORDERED_END), produces an equivalent result to that of the collective routine MPI_FILE_WRITE_ORDERED.

This subroutine returns immediately.

Begin calls are collective over the group of tasks that participated in the collective open and follow the ordering rules for collective operations.

As with any nonblocking data access operation, the user must not use the buffer passed to a begin subroutine while the operation is outstanding. The operation must be completed with an end subroutine before it is safe to access, reuse, or free the buffer.

Parameters

fh The file handle (handle) (INOUT).

buf
The initial address of the buffer (choice) (IN).

count
The number of elements in the buffer (integer) (IN).

datatype
The data type of each buffer element (handle) (IN).

IERROR
The Fortran return code. It is always the last argument.

Notes

Only one split collective operation can be active on any given file handle.

A file handle that is being used in a split collective operation cannot be used for a blocking collective operation.

Split collective operations do not inter-operate with the corresponding regular collective operation. For example, in a single collective write operation, an MPI_FILE_WRITE_ORDERED on one task does not match an MPI_FILE_WRITE_ORDERED_BEGIN and MPI_FILE_WRITE_ORDERED_END pair on another task.

The begin and end subroutines must be called from the same thread.

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 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 data type.

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

Uncommitted datatype (MPI_ERR_TYPE)
datatype must be committed.

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

Pending split collective data access operation (MPI_ERR_OTHER)
A collective data access operation is attempted while there is a pending split collective data access operation on the same file handle.

Related information

- MPI_FILE_WRITE_ORDERED
- MPI_FILE_WRITE_ORDERED_END
- MPI_FILE_WRITE_SHARED

MPI_FILE_WRITE_ORDERED_END, MPI_File_write_ordered_end

Completes a split collective write operation to a file using the shared file pointer.

C synopsis

```
#include <mpi.h>
int MPI_File_write_ordered_end(MPI_File fh,void *buf,MPI_Status *status);
```

C++ synopsis

```
#include mpi.h
void MPI::File::Write_ordered_end(const void* buf);
```

```
#include mpi.h
void MPI::File::Write_ordered_end(const void* buf, MPI::Status& status);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_WRITE_ORDERED_END(INTEGER FH,CHOICE BUF,INTEGER STATUS(MPI_STATUS_SIZE),
                           INTEGER IERROR)
```

Description

This subroutine ends a split collective operation that was initiated by the matching begin subroutine (MPI_FILE_WRITE_ORDERED_BEGIN). Combined with the begin subroutine, it produces an equivalent result to that of the collective routine MPI_FILE_WRITE_ORDERED.

End calls are collective over the group of tasks that participated in the collective open and follow the ordering rules for collective operations. Each end call matches the preceding begin call for the same collective operation. When an end call is made, exactly one unmatched begin call for the same operation must precede it.

This subroutine returns only when the user's buffer that contains the data to be written can be modified safely.

The number of bytes actually written by the calling task is stored in *status*.

Parameters

fh The file handle (handle) (INOUT).

buf
The initial address of the buffer (choice) (IN).

status
The status object (Status) (OUT).

IERROR
The Fortran return code. It is always the last argument.

Notes

Return from the call does not guarantee that the data has been written to the storage device (or devices). In particular, written data may still be present in system buffers. However, it guarantees that the memory buffer can be safely reused.

Only one split collective operation can be active on any given file handle.

A file handle that is being used in a split collective operation cannot be used for a blocking collective operation.

Split collective operations do not inter-operate with the corresponding regular collective operation. For example, in a single collective write operation, an MPI_FILE_WRITE_ORDERED on one task does not match an MPI_FILE_WRITE_ORDERED_BEGIN and MPI_FILE_WRITE_ORDERED_END pair on another task.

The begin and end subroutines must be called from the same thread.

Passing MPI_STATUS_IGNORE for the *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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

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

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

No pending split collective data access operation (MPI_ERR_OTHER)
The end phase of a split collective data access operation is attempted while there is no pending split collective data access operation.

Write conversion error (MPI_ERR_CONVERSION)
The conversion attempted during the write operation failed.

Invalid status ignore value

Related information

- MPI_FILE_WRITE_ORDERED
- MPI_FILE_WRITE_ORDERED_BEGIN
- MPI_FILE_WRITE_SHARED

MPI_FILE_WRITE_SHARED, MPI_File_write_shared

Writes to a file using the shared file pointer.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::File::Write_shared(const void* buf, int count,
        const MPI::Datatype& datatype);
```

```
#include mpi.h
void MPI::File::Write_shared(const void* buf, int count,
        const MPI::Datatype& datatype,
            MPI::Status& status);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FILE_WRITE_SHARED(INTEGER FH,CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,
              INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
```

Description

This subroutine tries to write, into the file referred to by *fh*, *count* items of type *datatype* out of the buffer *buf*, starting at the current file location as determined by the value of the shared file pointer. The call returns only when it is safe to reuse *buf*. *status* contains the number of bytes successfully written. You can use accessor functions MPI_GET_COUNT and MPI_GET_ELEMENTS to extract from *status* the number of items and the number of intrinsic MPI elements successfully written, respectively.

Parameters

fh The file handle (handle) (INOUT).

buf
The initial address of the buffer (choice) (IN).

count
The number of elements in the buffer (integer) (IN).

datatype
The data type of each buffer element (handle) (IN).

status
The status object (Status) (OUT).

IERROR
The Fortran return code. It is always the last argument.

Notes

Return from the call does not guarantee that the data has been written to the storage device (or devices). In particular, written data may still be present in system buffers. However, it guarantees that the memory buffer can be safely reused.

Passing MPI_STATUS_IGNORE for the *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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):

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 data type.

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

Uncommitted datatype (MPI_ERR_TYPE)
datatype must be committed.

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.

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

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

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

Write conversion error (MPI_ERR_CONVERSION)
The conversion attempted during the write operation failed.

Invalid status ignore value

Related information

- MPI_FILE_IWRITE_SHARED
- MPI_FILE_WRITE_ORDERED
- MPI_FILE_WRITE_ORDERED_BEGIN
- MPI_FILE_WRITE_ORDERED_END

MPI_FINALIZE, MPI_Finalize

Terminates all MPI processing.

C synopsis

```
#include <mpi.h>
int MPI_Finalize(void);
```

C++ synopsis

```
#include mpi.h
void MPI::Finalize();
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FINALIZE(INTEGER IERROR)
```

Description

Make sure this subroutine 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 MPI_FILE_OPEN 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 nonMPI processing after calling MPI_FINALIZE, but no other MPI calls (including MPI_INIT) can be made.

In a threads 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.

At MPI_FINALIZE there is now an implicit MPI_COMM_FREE of MPI_COMM_SELF. Because MPI_COMM_SELF cannot have been freed by user code and cannot be used after MPI_FINALIZE, there is no direct effect of this change. The value of this implicit free is that any attribute that a user may attach to MPI_COMM_SELF will be deleted in MPI_FINALIZE and its attribute delete function called. A library layered on MPI can take advantage of this to force its own cleanup code to run whenever MPI_FINALIZE gets called. This is done by packaging the cleanup logic as an attribute delete function and attaching an attribute to MPI_COMM_SELF. It is legitimate to make MPI calls in the attribute callbacks and a call to MPI_FINALIZED inside a delete function will report that MPI is still active.

If an attribute delete function returns a nonzero return code, the code it does return is passed to the error handler associated with MPI_COMM_WORLD. The default handler, MPI_ERROR_ARE_FATAL, will embed the error code in the message it prints. If there is a returning error handler on MPI_COMM_WORLD, MPI_FINALIZE will return a code indicating that a delete callback failed. MPI_FINALIZE does not return the error return code issued by the delete function.

In an environment that uses dynamic process management, MPI_FINALIZE is collective over the processes of the connected worlds. Also, if a process terminates without calling MPI_FINALIZE, independent processes are not affected, but the effect on the processes of the connected worlds is not defined.

Parameters

IERROR
The Fortran return code. It is always the last argument.

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, MPI_Buffer_detach" on page 90.

Errors

MPI_COMM_SELF attribute delete function returned error

MPI already finalized

MPI not initialized

Related information

- MPI_ABORT
- MPI_BUFFER_DETACH
- MPI_INIT

MPI_FINALIZED, MPI_Finalized

Returns **true** if MPI_FINALIZE has completed.

C synopsis

```
#include <mpi.h>
int MPI_Finalized(int *flag);
```

C++ synopsis

```
#include mpi.h
bool MPI::Is_finalized();
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FINALIZED(LOGICAL FLAG, INTEGER IERROR)
```

Description

This subroutine returns **true** if MPI_FINALIZE has completed. It is legal to call MPI_FINALIZED before MPI_INIT and after MPI_FINALIZE.

Parameters

flag
Set to **true** if MPI is finalized (logical) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

Once MPI has been finalized, it is no longer active and cannot be restarted. A library layered on top of MPI needs to be able to determine this to act accordingly.

MPI is *active* and it is thus safe to call MPI functions if MPI_INIT has completed and MPI_FINALIZE has not completed. If a library has no other way of knowing whether MPI is active or not, it can use MPI_INITIALIZED and MPI_FINALIZED to determine this. For example, MPI is still *active* in callback functions that are invoked during the MPI_FINALIZE actions to free MPI_COMM_SELF.

Errors

MPI already finalized

MPI not initialized

Related information

- MPI_FINALIZE
- MPI_INIT
- MPI_INITIALIZED

MPI_FREE_MEM, MPI_Free_mem

Frees a block of storage.

C synopsis

```
#include <mpi.h>
int MPI_Free_mem (void *base);
```

C++ synopsis

```
#include mpi.h
void MPI::Free_mem(void *base):
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_FREE_MEM(CHOICE BASE, INTEGER IERROR)
```

Description

This subroutine frees a block of storage previously allocated by the **MPI_ALLOC_MEM** routine and pointed to by the *base* argument. Undefined results occur if the *base* argument is not a pointer to a block of storage that is currently allocated.

Parameters

base
The initial address of the memory segment allocated by **MPI_ALLOC_MEM** (choice) (IN)

IERROR
The Fortran return code. It is always the last argument.

Errors

Fatal errors:

MPI not initialized (MPI_ERR_OTHER)

MPI already finalized (MPI_ERR_OTHER)

Related information

- MPI_ALLOC_MEM

MPI_GATHER, MPI_Gather

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Comm::Gather(const void* sendbuf, int sendcount,
         const MPI::Datatype& sendtype, void* recvbuf,
         int recvcount, const MPI::Datatype& recvtype,
         int root) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GATHER(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,
    CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER ROOT,
    INTEGER COMM,INTEGER IERROR)
```

Description

This subroutine 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, pair-wise between each task and the root. Distinct type maps between sender and receiver are allowed.

The following information applies to 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.

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

If *comm* is an inter-communicator, the call involves all tasks in the inter-communicator, but with one group (group A) defining the root task. All tasks in the other group (group B) pass the same value in *root*, which is the rank of the root in group A. The root passes the value MPI_ROOT in *root*. All other tasks in group A pass the value MPI_PROC_NULL in *root*. Data is gathered from all tasks

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

MPI_IN_PLACE is not supported for inter-communicators.

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator occur in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

Parameters

sendbuf
The starting address of the send buffer (choice) (IN)

sendcount
The number of elements in the send buffer (integer) (IN)

sendtype
The data type of the send buffer elements (handle) (IN)

recvbuf
The address of the receive buffer (choice, significant only at *root*) (OUT)

recvcount
The number of elements for any single receive (integer, significant only at *root*) (IN)

recvtype
The data type of the receive buffer elements (handle, significant only at *root*) (IN)

root
The rank of the receiving task (integer) (IN)

comm
The communicator (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

In the 64-bit library, this function uses a shared memory optimization among the tasks on a node. This optimization is discussed in the chapter *Using shared memory* of *IBM Parallel Environment Runtime Edition: MPI Programming Guide*, and is enabled by default. This optimization is not available to 32-bit programs.

Errors

Fatal errors:

Invalid communicator

Invalid counts
count < **0**

Invalid datatypes

Type not committed

Invalid root

For an intra-communicator: *root* < **0** or *root* >= *groupsize*

For an inter-communicator: *root* < **0** and is neither MPI_ROOT nor MPI_PROC_NULL, or *root* >= *groupsize* of the remote group

Unequal message lengths

Invalid use of MPI_IN_PLACE

MPI not initialized

MPI already finalized

Develop mode error if:

Inconsistent root

Inconsistent message length

Related information

- MPE_IGATHER
- MPI_ALLGATHER
- MPI_GATHER
- MPI_SCATTER

MPI_GATHERV, MPI_Gatherv

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Comm::Gatherv(const void* sendbuf, int sendcount,
         const MPI::Datatype& sendtype, void* recvbuf,
         const int recvcounts[], const int displs[],
         const MPI::Datatype& recvtype, int root) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GATHERV(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,
    CHOICE RECVBUF,INTEGER RECVCOUNTS(*),INTEGER DISPLS(*),
    INTEGER RECVTYPE,INTEGER ROOT,INTEGER COMM,INTEGER IERROR)
```

Description

This subroutine 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, pair-wise 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.

Parameters

sendbuf
The starting address of the send buffer (choice) (IN)

sendcount
The number of elements in the send buffer (integer) (IN)

sendtype
The data type of the send buffer elements (handle) (IN)

recvbuf
The address of the receive buffer (choice, significant only at *root*) (OUT)

recvcounts
An integer array (of length *groupsize*) that contains the number of elements received from each task (significant only at *root*) (IN)

displs
An integer array (of length *groupsize*). Entry *i* specifies the displacement relative to *recvbuf* at which to place the incoming data from task *i* (significant only at *root*) (IN)

recvtype
The data type of the receive buffer elements (handle, significant only at *root*) (IN)

root
The rank of the receiving task (integer) (IN)

comm
The communicator (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

Displacements are expressed as elements of type *recvtype*, not as bytes.

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

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

MPI_IN_PLACE is not supported for inter-communicators.

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator occur in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

In the 64-bit library, this function uses a shared memory optimization among the tasks on a node. This optimization is discussed in the chapter *Using shared memory* of *IBM Parallel Environment Runtime Edition: MPI Programming Guide*, and is enabled by default. This optimization is not available to 32-bit programs.

Errors

Fatal errors:

Invalid communicator

Invalid counts
count < **0**

Invalid datatypes

Type not committed

Invalid root

For an intra-communicator: *root* < **0** or *root* >= *groupsize*

For an inter-communicator: *root* < **0** and is neither MPI_ROOT nor MPI_PROC_NULL, or *root* >= *groupsize* of the remote group

Unequal message lengths

Invalid use of MPI_IN_PLACE

MPI not initialized

MPI already finalized

Develop mode error if:

Inconsistent root

Related information

- MPE_IGATHER
- MPI_GATHER

MPI_GET, MPI_Get

Transfers data from a window at the target task to the origin task.

C synopsis

```
#include <mpi.h>
int MPI_Get (void *origin_addr, int origin_count,
             MPI_Datatype origin_datatype, int target_rank,
             MPI_Aint target_disp, int target_count,
             MPI_Datatype target_datatype, MPI_Win win);
```

C++ synopsis

```
#include mpi.h
void MPI::Win::Get(void* origin_addr, int origin_count,
     const MPI::Datatype& origin_datatype, int target_rank,
                    MPI::Aint target_disp, int target_count,
     const MPI::Datatype& target_datatype) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GET(CHOICE ORIGIN_ADDR, INTEGER ORIGIN_COUNT, INTEGER ORIGIN_DATATYPE,
        INTEGER TARGET_RANK, INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP,
        INTEGER TARGET_COUNT, INTEGER TARGET_DATATYPE, INTEGER WIN,
        INTEGER IERROR)
```

Description

MPI_GET transfers *origin_count* successive entries of the type specified by *origin_datatype*, starting at address *origin_addr* on the origin task from the target task specified by *win* and *target_rank*.

The data are read from the target buffer at address (*target_addr* = *window_base* + *target_disp* * *disp_unit*), where *window_base* and *disp_unit* are the base address and window displacement unit specified at window initialization, by the target task. The target buffer is specified by the arguments *target_count* and *target_datatype*.

The data transfer is the same as that which would occur if the origin task issued a receive operation with arguments *origin_addr*, *origin_count*, *origin_datatype*, *target_rank*, *tag*, *comm*, and the target task issued a send operation with arguments *target_addr*, *target_count*, *target_datatype*, *source*, *tag*, *comm*, where *target_addr* is the target buffer address computed as shown in the previous paragraph, and *comm* is a communicator for the group of *win*.

The communication must satisfy the same constraints as for a similar message-passing communication. The *target_datatype* may not specify overlapping entries in the target buffer. The message sent must fit, without truncation, in the target buffer. Furthermore, the target buffer must fit in the target window.

The *target_datatype* argument is a handle to a data type object that is defined at the origin task, even though it defines a data layout in the target task memory. This does not cause any problems in a homogeneous environment. In a heterogeneous environment, only portable data types are valid.

The data type object is interpreted at the target task. The outcome is as if the target data type object were defined at the target task, by the same sequence of calls used to define it at the origin task. The target data type must contain relative displacements, not absolute addresses.

Parameters

origin_addr
The initial address of the origin buffer (choice) (IN)

origin_count
The number of entries in origin buffer (nonnegative integer) (IN)

origin_datatype
The data type of each entry in the origin buffer (handle) (IN)

target_rank
The rank of the target (nonnegative integer) (IN)

target_disp
The displacement from the start of the window to the target buffer (nonnegative integer) (IN)

target_count
The number of entries in the target buffer (nonnegative integer) (IN)

target_datatype
The data type of each entry in the target buffer (handle) (IN)

win
The window object used for communication (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_GET does not require that data move from target to origin until some synchronization occurs. IBM PE MPI may try to combine multiple gets from a target within an epoch into a single data transfer. The user must not modify the source buffer or make any assumption about the contents of the destination buffer until after a synchronization operation has closed the epoch.

On some systems, there may be reasons to use special memory for one-sided communication buffers. MPI_ALLOC_MEM may be the preferred way to allocate buffers on these systems. With IBM PE MPI, there is no advantage to using MPI_ALLOC_MEM, but you can use it to improve the portability of your MPI code.

Errors

Invalid origin count (*count*)

Invalid origin datatype (*handle*)

Invalid target rank (*rank*)

Invalid target displacement (*value*)

Invalid target count (*count*)

Invalid target datatype (*handle*)

Invalid window handle (*handle*)

Target outside access group

Origin buffer too small (*size*)

Target buffer ends outside target window

Target buffer starts outside target window

RMA communication call outside access epoch

RMA communication call in progress

RMA synchronization call in progress

Related information

- MPI_ACCUMULATE
- MPI_PUT

MPI_GET_ADDRESS, MPI_Get_address

Returns the address of a location in memory.

C synopsis

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

C++ synopsis

```
#include mpi.h
MPI::Aint MPI::Get_address(void* location);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GET_ADDRESS(CHOICE LOCATION(*),
  INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS,
  INTEGER IERROR)
```

Description

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

Parameters

location
The location in caller memory (choice) (IN)

address
The address of the location (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_GET_ADDRESS is equivalent to **address= (MPI_Aint) location** in C, but this subroutine is portable to processors with less straightforward addressing.

MPI_GET_ADDRESS is synonymous with MPI_ADDRESS. MPI_ADDRESS is not available in C++. In Fortran, MPI_GET_ADDRESS returns an argument of type INTEGER(KIND=MPI_ADDRESS_KIND) to support 32-bit and 64-bit addresses. Such variables may be declared as INTEGER*4 in purely 32-bit codes and as INTEGER*8 in 64-bit codes; KIND=MPI_ADDRESS_KIND works correctly in either mode. MPI_ADDRESS is provided for backward compatibility. However, users are encouraged to switch to MPI_GET_ADDRESS, in both Fortran and C.

Current Fortran MPI codes will run unmodified, and will port to any system. However, these codes may fail if addresses larger than (2 (to the power of 32) -1) are used in the program. New codes should be written so that they use MPI_GET_ADDRESS. This provides compatibility with C and C++ and avoids errors on 64-bit architectures. However, such newly-written codes may need to be rewritten slightly to port to old Fortran 77 environments that do not support KIND declarations.

Errors

Fatal errors:

MPI not initialized

MPI already finalized

Related information

- MPI_TYPE_CREATE_HINDEXED
- MPI_TYPE_CREATE_HVECTOR
- MPI_TYPE_CREATE_STRUCT

MPI_GET_COUNT, MPI_Get_count

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);
```

C++ synopsis

```
#include mpi.h
int MPI::Status::Get_count(const MPI::Datatype& datatype) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GET_COUNT(INTEGER STATUS(MPI_STATUS_SIZE),INTEGER DATATYPE,
    INTEGER COUNT,INTEGER IERROR)
```

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 nonblocking operation request and the corresponding blocking operation does not provide a *status* argument, the *status* from this wait or test call does not contain meaningful source, tag, or message size information.

Parameters

status
A status object (Status) (IN). Note that in Fortran a single status object is an array of integers.

datatype
The data type of each message element (handle) (IN)

count
The number of elements (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid datatype

Type not committed

MPI not initialized

MPI already finalized

Related information

- MPI_IRECV
- MPI_PROBE
- MPI_RECV
- MPI_WAIT

MPI_GET_ELEMENTS, MPI_Get_elements

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);
```

C++ synopsis

```
#include mpi.h
int MPI::Status::Get_elements(const MPI::Datatype& datatype) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GET_ELEMENTS(INTEGER STATUS(MPI_STATUS_SIZE),INTEGER DATATYPE,
    INTEGER COUNT,INTEGER IERROR)
```

Description

This subroutine 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 or test call does not contain meaningful source, tag, or message size information.

Parameters

status
A status of object (status) (IN). Note that in Fortran a single status object is an array of integers.

datatype
The data type used by the operation (handle) (IN)

count
An integer specifying the number of basic elements (OUT)

IERROR
The Fortran return code. It is always the last argument.

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

Returns the name of the local processor.

C synopsis

```
#include <mpi.h>
int MPI_Get_processor_name(char *name,int *resultlen);
```

C++ synopsis

```
#include mpi.h
void MPI::Get_processor_name(char*& name, int& resultlen);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GET_PROCESSOR_NAME(CHARACTER NAME(*),INTEGER RESULTLEN,
      INTEGER IERROR)
```

Description

This subroutine 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*. For C, the returned *name* is a null-terminated string with the terminating byte not counted in *resultlen*. For Fortran, the returned *name* is a blank-padded string.

Parameters

name
A unique specifier for the actual node (OUT)

resultlen
Specifies the printable character length of the result returned in *name* (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

MPI not initialized

MPI already finalized

MPI_GET_VERSION, MPI_Get_version

Returns the version of the MPI standard supported in this release.

C synopsis

```
#include <mpi.h>
int MPI_Get_version(int *version,int *subversion);
```

C++ synopsis

```
#include mpi.h
void MPI::Get_version(int& version, int& subversion);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GET_VERSION(INTEGER VERSION, INTEGER SUBVERSION, INTEGER IERROR)
```

Description

This subroutine is used to determine the version of the MPI standard supported by the MPI implementation.

The symbolic constants MPI_VERSION and MPI_SUBVERSION, which are included in **mpi.h** and **mpif.h**, provide similar compile-time information.

MPI_GET_VERSION can be called before MPI_INIT.

Parameters

version
MPI standard version number (integer) (OUT)

subversion
MPI standard subversion number (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

MPI_GRAPH_CREATE, MPI_Graph_create

Creates a new communicator containing graph topology information.

C synopsis

```
#include <mpi.h>
int MPI_Graph_create(MPI_Comm comm_old,int nnodes, int *index,
    int *edges,int reorder,MPI_Comm *comm_graph);
```

C++ synopsis

```
#include mpi.h
MPI::Graphcomm MPI::Intracomm::Create_graph(int nnodes, const int index[],
         const int edges[], bool reorder) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GRAPH_CREATE(INTEGER COMM_OLD,INTEGER NNODES,INTEGER INDEX(*),
    INTEGER EDGES(*),LOGICAL REORDER,INTEGER COMM_GRAPH,
    INTEGER IERROR)
```

Description

This subroutine 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* than there are in *nnodes*, some tasks are returned with a value of MPI_COMM_NULL for *comm_graph*.

Parameters

comm_old
The input communicator (handle) (IN)

nnodes
An integer specifying the number of nodes in the graph (IN)

index
An array of integers describing node degrees (IN)

edges
An array of integers describing graph edges (IN)

reorder
Set to **true** means that ranking may be reordered (logical) (IN)

comm_graph
The communicator with the graph topology added (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

Table 2 on page 323 and Table 3 on page 323 provide an example of how to define the arguments *nnodes*, *index*, and *edges*. Suppose there are four tasks (0, 1, 2, 3) with the following adjacency matrix:

Table 2. Example in MPI_GRAPH_CREATE of adjacency matrix

Task	Neighbors
0	1, 3
1	0
2	3
3	0, 2

Then the input arguments are:

Table 3. Input arguments for example in MPI_GRAPH_CREATE

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 0, and *index[i]–index[i–1]* is the degree of node *i*, *i=1*, **...**, *nnodes–1*. The list of neighbors of node 0 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 0, and *index(i+1)– index(i)* is the degree of node *i*, *i=1*, **...**, *nnodes–1*. The list of neighbors of node 0 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)*.

Errors

MPI not initialized

MPI already finalized

Invalid communicator

Invalid communicator type
must be intra-communicator

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*

Conflicting collective operations on communicator

Related information

- MPI_CART_CREATE

MPI_GRAPH_GET, MPI_Graph_get

Retrieves graph topology information from a communicator.

C synopsis

```
#include <mpi.h>
int MPI_Graph_get(MPI_Comm comm,int maxindex,int maxedges,
      int *index,int *edges);
```

C++ synopsis

```
#include mpi.h
void MPI::Graphcomm::Get_topo(int maxindex, int maxedges, int index[],
         int edges[]) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GRAPH_GET(INTEGER COMM,INTEGER MAXINDEX,INTEGER MAXEDGES,
      INTEGER INDEX(*),INTEGER EDGES(*),INTEGER IERROR)
```

Description

This subroutine retrieves the *index* and *edges* graph topology information associated with a communicator.

Parameters

comm
A communicator with graph topology (handle) (IN)

maxindex
An integer specifying the length of *index* in the calling program (IN)

maxedges
An integer specifying the length of *edges* in the calling program (IN)

index
An array of integers containing node degrees (OUT)

edges
An array of integers containing node neighbors (OUT)

IERROR
The Fortran return code. It is always the last argument.

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_GRAPH_CREATE
- MPI_GRAPHDIMS_GET

MPI_GRAPH_MAP, MPI_Graph_map

Computes placement of tasks on the physical processor.

C synopsis

```
#include <mpi.h>
int MPI_Graph_map(MPI_Comm comm,int nnodes,int *index,int *edges,int *newrank);
```

C++ synopsis

```
#include mpi.h
int MPI::Graphcomm::Map(int nnodes, const int index[],
   const int edges[]) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GRAPH_MAP(INTEGER COMM,INTEGER NNODES,INTEGER INDEX(*),
              INTEGER EDGES(*),INTEGER NEWRANK,INTEGER IERROR)
```

Description

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

Parameters

comm
The input communicator (handle) (IN)

nnodes
The number of graph nodes (integer) (IN)

index
An integer array specifying node degrees (IN)

edges
An integer array specifying node adjacency (IN)

newrank
The reordered rank, or MPI_Undefined if the calling task does not belong to the graph (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_GRAPH_MAP returns **newrank** as the original rank of the calling task if it belongs to the graph or MPI_UNDEFINED if it does not.

Errors

Invalid communicator

Invalid communicator type
must be intra-communicator

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_CART_MAP
- MPI_GRAPH_CREATE

MPI_GRAPH_NEIGHBORS, MPI_Graph_neighbors

Returns the neighbors of the given task.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::Graphcomm::Get_neighbors(int rank, int maxneighbors,
       int neighbors[])
       const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GRAPH_NEIGHBORS(MPI_COMM COMM,INTEGER RANK,INTEGER MAXNEIGHBORS,
      INTEGER NNEIGHBORS(*),INTEGER IERROR)
```

Description

This subroutine retrieves the adjacency information for a particular task.

Parameters

comm
A communicator with graph topology (handle) (IN)

rank
The rank of a task within group of *comm* (integer) (IN)

maxneighbors
The size of array *neighbors* (integer) (IN)

neighbors
The ranks of tasks that are neighbors of the specified task (array of integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

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_CREATE
- MPI_GRAPH_NEIGHBORS_COUNT

MPI_GRAPH_NEIGHBORS_COUNT, MPI_Graph_neighbors_count

Returns the number of neighbors of the given task.

C synopsis

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

C++ synopsis

```
#include mpi.h
int MPI::Graphcomm::Get_neighbors_count(int rank) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GRAPH_NEIGHBORS_COUNT(INTEGER COMM,INTEGER RANK,
INTEGER NEIGHBORS(*),INTEGER IERROR)
```

Description

This subroutine returns the number of neighbors of the given task.

Parameters

comm
A communicator with graph topology (handle) (IN)

rank
The rank of a task within *comm* (integer) (IN)

neighbors
The number of neighbors of the specified task (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

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_CREATE
- MPI_GRAPH_NEIGHBORS

MPI_GRAPHDIMS_GET, MPI_Graphdims_get

Retrieves graph topology information from a communicator.

C synopsis

```
#include <mpi.h>
int MPI_Graphdims_get(MPI_Comm comm,int *nnodes,int *nedges);
```

C++ synopsis

```
#include mpi.h
void MPI::Graphcomm::Get_dims(int nnodes[],
         int nedges[])
         const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GRAPHDIMS_GET(INTEGER COMM,INTEGER NNDODES,INTEGER NEDGES,
      INTEGER IERROR)
```

Description

This subroutine retrieves the number of nodes and the number of edges in the graph topology associated with a communicator.

Parameters

comm
A communicator with graph topology (handle) (IN)

nnodes
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
An integer specifying the number of edges in the graph. (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

MPI not initialized

MPI already finalized

Invalid communicator

No topology

Invalid topology type
topology type must be graph

Related information

- MPI_GRAPH_CREATE
- MPI_GRAPH_GET

MPI_GREQUEST_COMPLETE, MPI_Grequest_complete

Marks the generalized request complete.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::Grequest::Complete();
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GREQUEST_COMPLETE(INTEGER REQUEST, INTEGER IERROR)
```

Description

This subroutine informs MPI that the operations represented by the generalized request are complete. A call to MPI_WAIT(*request, status*) will return and a call to MPI_TEST(*request, flag, status*) will return *flag* = **true** only after a call to MPI_GREQUEST_COMPLETE has declared that these operations are complete.

Parameters

request
The generalized request (handle) (INOUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Fatal errors:

A GRequest free function returned an error

MPI_GRequest_free function fails

Invalid request handle

Not a GRequest handle

MPI already finalized

MPI not initialized

Related information

- MPI_GREQUEST_START
- MPI_TEST
- MPI_WAIT

MPI_GREQUEST_START, MPI_Grequest_start

Initializes a generalized request.

C synopsis

```
#include <mpi.h>
int MPI_Grequest_start(MPI_Grequest_query_function *query_fn,
         MPI_Grequest_free_function *free_fn,
         MPI_Grequest_cancel_function *cancel_fn,
         void *extra_state, MPI_Request *request);
```

C++ synopsis

```
#include mpi.h
MPI::Grequest MPI::Grequest::Start(MPI::Grequest::Query_function query_fn,
       MPI::Grequest::Free_function free_fn,
       MPI::Grequest::Cancel_function cancel_fn,
       void *extra_state);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GREQUEST_START(EXTERNAL QUERY_FN, EXTERNAL FREE_FN, EXTERNAL CANCEL_FN,
     INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, INTEGER REQUEST, INTEGER IERROR)
```

Description

This subroutine starts a generalized request and returns a handle to it in *request*. This is a nonblocking operation.

Parameters

query_fn
The callback function that is invoked when the request status is queried (function) (IN)

free_fn
The callback function that is invoked when the request is freed (function) (IN)

cancel_fn
The callback function that is invoked when the request is cancelled (function) (IN)

extra_state
The extra state (integer) (IN)

request
The generalized request (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

For a generalized request, the operation associated with the request is designed by the application programmer and performed by the application; therefore, the application must notify MPI when the operation has finished. It does this by making a call to MPI_GREQUEST_COMPLETE. MPI maintains the *completion* status of generalized requests. Any other request state has to be maintained by the user.

In C++, a generalized request belongs to the class MPI::Grequest, which is a derived class of MPI::Request. It is of the same type as regular requests, in C and Fortran.

The syntax and meaning of the callback functions follow. All callback functions are passed the *extra_state* argument that was associated with the request by the starting call MPI_GREQUEST_START. This can be used to provide extra information to the callback functions or to maintain the user-defined state for the request.

In C, the query function is:

```
typedef int MPI_Grequest_query_function(void *extra_state, MPI_Status *status);
```

In Fortran, the query function is:

```
SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)
INTEGER STATUS(MPI_STATUS_SIZE), IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

In C++, the query function is:

```
typedef int MPI::Grequest::Query_function(void* extra_state, MPI::Status& status);
```

The *query_fn* function computes the status that should be returned for the generalized request. The status should include information about the successful or unsuccessful cancellation of the request (the result to be returned by MPI_TEST_CANCELLED).

The *query_fn* callback is invoked by the MPI_WAIT or MPI_TEST {ANY|SOME|ALL} call that completed the generalized request associated with this callback. The callback function is also invoked by calls to MPI_REQUEST_GET_STATUS, if the request is complete when the call occurs. In both cases, the callback is passed a reference to the corresponding status variable passed by the user to the MPI call; the status set by the callback function is returned by the MPI call.

If the user provided MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE to the MPI function that causes *query_fn* to be called, MPI passes a valid temporary status object to *query_fn*, and this status is discarded upon return of the callback function. This protects the *query_fn* from any need to deal with MPI_STATUS_IGNORE. *query_fn* is invoked only after MPI_GREQUEST_COMPLETE is called on the request; it may be invoked several times for the same generalized request, that is, if the user calls MPI_REQUEST_GET_STATUS several times for this request. A call to MPI_WAIT or MPI_TEST {SOME|ALL} may cause multiple invocations of query_fn callback functions, one for each generalized request that is completed by the MPI call. The order of these invocations is not specified by MPI.

In C, the free function is:

```
typedef int MPI_Grequest_free_function(void *extra_state);
```

In Fortran, the free function is:

```
SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)
INTEGER IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

In C++, the free function is:

```
typedef int MPI::Grequest::Free_function(void* extra_state);
```

The *free_fn* function is used to clean up user-allocated resources when the generalized request is freed or completed. Freeing *extra_state* is an example.

The *free_fn* callback is invoked by the MPI_WAIT or MPI_TEST {ANY|SOME|ALL} call that completed the generalized request associated with this callback. *free_fn* is invoked after the call to *query_fn* for the same request. However, if the MPI call completed multiple generalized requests, the order in which *free_fn* callback functions are invoked is not specified by MPI.

The *free_fn* is also invoked for generalized requests that are freed by a call to MPI_REQUEST_FREE (no call to MPI_WAIT or MPI_TEST {ANY|SOME|ALL} occurs for such a request). In this case, the callback function is called either in the MPI call MPI_REQUEST_FREE(*request*), or in the MPI call MPI_GREQUEST_COMPLETE(*request*), whichever happens last. That is, in this case the actual freeing code is run as soon as both MPI_REQUEST_FREE and MPI_GREQUEST_COMPLETE have occurred. The request is not deallocated until after *free_fn* completes. *free_fn* is invoked only once per request by a correct program.

Calling MPI_REQUEST_FREE(*request*) causes the request handle to be set to MPI_REQUEST_NULL. This handle to the generalized request is no longer valid. However, user copies of this handle are valid until after free_fn completes because MPI does not deallocate the object until then. Because *free_fn* is not called until after MPI_GREQUEST_COMPLETE, the user copy of the handle can be used to make this call. Normally, the routine that is to carry out the user's operation is passed its own copy of the request handle at the time it is started. It will us this copy of the request handle in a call to MPI_GREQUEST_COMPLETE once it has finished. MPI deallocates the object after *free_fn* completes. At this point, user copies of the request handle no longer point to a valid request. MPI does not set user copies to MPI_REQUEST_NULL in this case, so it is up to the user to avoid accessing this stale handle.

In C, the cancel function is:

```
typedef int MPI_Grequest_cancel_function(void *extra_state, int complete);
```

In Fortran, the cancel function is:

```
SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)
INTEGER IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
LOGICAL COMPLETE
```

In C++, the cancel function is:

```
typedef int MPI::Grequest::Cancel_function(void* extra_state, bool complete);
```

The *cancel_fn* function is invoked to attempt the cancellation of a generalized request. It is called by MPI_CANCEL(request). MPI passes complete = true to the callback function if MPI_GREQUEST_COMPLETE was already called on the request, and complete = false otherwise. The user's *cancel_fn* must not try to cancel the operation if it is already complete.

All callback functions must return an error code. The code is passed back and dealt with as appropriate for that error code by the MPI function that invoked the callback function. For example, the callback function return code may be returned as the return code of the function triggering the callback. In the case of an MPI_WAIT or MPI_TEST call that invokes both *query_fn* and *free_fn* and both returning errors, the MPI completion function will return the error code returned

by the last callback, namely *free_fn*. If one or more of the requests in a call to MPI_WAIT or MPI_TEST {SOME | ALL} failed, the MPI call returns MPI_ERR_IN_STATUS. In such a case, if the MPI call was passed an array of statuses, MPI returns in each of the statuses that correspond to a completed generalized request the error code returned by the corresponding invocation of its *query_fn* or *free_fn* callback function. However, if the MPI function was passed MPI_STATUSES_IGNORE, then the individual error codes returned by each callback functions will be lost.

query_fn must not set the error field of status because it (*query_fn* may be called by MPI_WAIT or MPI_TEST, in which case the error field of status should not change. The MPI library knows the *context* in which *query_fn* is invoked and can decide correctly when to put in the error field of status the returned error code.

When the MPI_ERRORS_ARE_FATAL error handler is in effect, the MPI library issues the same message for all *query_fn* or *free_fn* return codes. The return code value is embedded in the message.

Errors

Fatal errors:

MPI already finalized

MPI not initialized

Related information

- MPI_CANCEL
- MPI_GREQUEST_COMPLETE
- MPI_REQUEST_FREE
- MPI_REQUEST_GET_STATUS
- MPI_TEST
- MPI_TEST_CANCELLED
- MPI_WAIT

MPI_Group_c2f

Translates a C group handle into a Fortran handle to the same group.

C synopsis

```
#include <mpi.h>
MPI_Fint MPI_Group_c2f(MPI_Group group);
```

Description

This function does not have C++ or Fortran bindings. MPI_Group_c2f translates a C group handle into a Fortran handle to the same group; it maps a null handle into a null handle and a non-valid handle into a non-valid handle. The converted handle is returned as the function's value. There is no error detection or return code.

Parameters

group
The group (handle) (IN)

Related information

- MPI_Group_f2c

MPI_GROUP_COMPARE, MPI_Group_compare

Compares the contents of two task groups.

C synopsis

```
#include <mpi.h>
int MPI_Group_compare(MPI_Group group1,MPI_Group group2,
    int *result);
```

C++ synopsis

```
#include mpi.h
static int MPI::Group::Compare(const MPI::Group& group1, const MPI::Group& group2);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GROUP_COMPARE(INTEGER GROUP1,INTEGER GROUP2,INTEGER RESULT,
    INTEGER IERROR)
```

Description

This subroutine 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 is different or group members are different, or both

Parameters

group1
: The first group (handle) (IN)

group2
: The second group (handle) (IN)

result
: The result (integer) (OUT)

IERROR
: The Fortran return code. It is always the last argument.

Errors

Invalid groups

MPI not initialized

MPI already finalized

Related information

- MPI_COMM_COMPARE

MPI_GROUP_DIFFERENCE, MPI_Group_difference

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);
```

C++ synopsis

```
#include mpi.h
static MPI::Group MPI::Group::Difference(const MPI::Group& group1,
      const MPI::Group& group2);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GROUP_DIFFERENCE(INTEGER GROUP1,INTEGER GROUP2,
      INTEGER NEWGROUP,INTEGER IERROR)
```

Description

This subroutine 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.

Parameters

group1
The first group (handle) (IN)

group2
The second group (handle) (IN)

newgroup
The difference group (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid groups

MPI not initialized

MPI already finalized

Related information

- MPI_GROUP_INTERSECTION
- MPI_GROUP_UNION

MPI_GROUP_EXCL, MPI_Group_excl

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);
```

C++ synopsis

```
#include mpi.h
MPI::Group MPI::Group::Excl(int n, const int ranks[]) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GROUP_EXCL(INTEGER GROUP,INTEGER N,INTEGER RANKS(*),
      INTEGER NEWGROUP,INTEGER IERROR)
```

Description

This subroutine 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, *newgroup* is identical to *group*.

Parameters

group
The group (handle) (IN)

n The number of elements in array **ranks** (integer) (IN)

ranks
The array of integer ranks in *group* that is not to appear in *newgroup* (IN)

newgroup
The new group derived from the preceding parameters, preserving the order defined by *group* (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid group

Invalid size
n < 0 or *n* > *groupsize*

Invalid ranks
ranks[*i*] < **0** or *ranks*[*i*] > = *groupsize*

Duplicate ranks

MPI not initialized

MPI already finalized

Related information

- MPI_GROUP_INCL
- MPI_GROUP_RANGE_EXCL
- MPI_GROUP_RANGE_INCL

MPI_Group_f2c

Returns a C handle to a group.

C synopsis

```
#include <mpi.h>
MPI_Group MPI_Group_f2c(MPI_Fint group);
```

Description

This function does not have C++ or Fortran bindings. MPI_Group_f2c returns a C handle to a group. If *group* is a valid Fortran handle to a group, MPI_Group_f2c returns a valid C handle to that same group. If *group* is set to the Fortran value MPI_GROUP_NULL, MPI_Group_f2c returns the equivalent null C handle. If *group* is not a valid Fortran handle, MPI_Group_f2c returns a non-valid C handle. The converted handle is returned as the function's value. There is no error detection or return code.

Parameters

group
The group (handle) (IN)

Errors

None.

Related information

- MPI_Group_c2f

MPI_GROUP_FREE, MPI_Group_free

Marks a group for deallocation.

C synopsis

```
#include <mpi.h>
int MPI_Group_free(MPI_Group *group);
```

C++ synopsis

```
#include mpi.h
void MPI::Group::Free();
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GROUP_FREE(INTEGER GROUP,INTEGER IERROR)
```

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.

Parameters

group
The group (handle) (INOUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid group

MPI not initialized

MPI already finalized

MPI_GROUP_INCL, MPI_Group_incl

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);
```

C++ synopsis

```
#include mpi.h
MPI::Group MPI::Group::Incl(int n, const int ranks[]) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GROUP_INCL(INTEGER GROUP,INTEGER N,INTEGER RANKS(*),
      INTEGER NEWGROUP,INTEGER IERROR)
```

Description

This subroutine 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**, *newgroup* is MPI_GROUP_EMPTY. This function can be used to reorder the elements of a group.

Parameters

group
The group (handle) (IN)

n The number of elements in array *ranks* and the size of *newgroup* (integer) (IN)

ranks
The ranks of tasks in *group* to appear in *newgroup* (array of integers) (IN)

newgroup
The new group derived in the preceding example, in the order defined by *ranks* (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid group

Invalid size
n < **0** or *n* > *groupsize*

Invalid ranks
ranks[i] < 0 or *ranks*[i] >= *groupsize*

Duplicate ranks

MPI not initialized

MPI already finalized

Related information

- MPI_GROUP_EXCL
- MPI_GROUP_RANGE_EXCL
- MPI_GROUP_RANGE_INCL

MPI_GROUP_INTERSECTION, MPI_Group_intersection

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);
```

C++ synopsis

```
#include mpi.h
static MPI::Group MPI::Group::Intersect(const MPI::Group& group1,
     const MPI::Group& group2);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GROUP_INTERSECTION(INTEGER GROUP1,INTEGER GROUP2,
      INTEGER NEWGROUP,INTEGER IERROR)
```

Description

This subroutine 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.

Parameters

group1
The first group (handle) (IN)

group2
The second group (handle) (IN)

newgroup
The intersection group (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid groups

MPI not initialized

MPI already finalized

Related information

- MPI_GROUP_DIFFERENCE
- MPI_GROUP_UNION

MPI_GROUP_RANGE_EXCL, MPI_Group_range_excl

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);
```

C++ synopsis

```
#include mpi.h
MPI::Group MPI::Group::Range_excl(int n, const int ranges[][3])
      const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GROUP_RANGE_EXCL(INTEGER GROUP,INTEGER N,INTEGER RANGES(3,*),
    INTEGER NEWGROUP,INTEGER IERROR)
```

Description

This subroutine 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 subroutine 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*.

Parameters

group
The group (handle) (IN)

n The number of triplets in array ranges (integer) (IN)

ranges
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
The new group derived from the preceding parameters that preserves the order in *group* (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid group

Invalid size
n < 0 or *n* > *groupsize*

Invalid rank
a computed rank < 0 or >= *groupsize*

Duplicate ranks

Invalid strides
stride[i] = 0

Too many ranks
Number of ranks > *groupsize*

MPI not initialized

MPI already finalized

Related information

- MPI_GROUP_EXCL
- MPI_GROUP_INCL
- MPI_GROUP_RANGE_INCL

MPI_GROUP_RANGE_INCL, MPI_Group_range_incl

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);
```

C++ synopsis

```
#include mpi.h
MPI::Group MPI::Group::Range_incl(int n, const int ranges[][3])
      const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GROUP_RANGE_INCL(INTEGER GROUP,INTEGER N,INTEGER RANGES(3,*),
    INTEGER NEWGROUP,INTEGER IERROR)
```

Description

This subroutine creates a new group consisting of selected ranges of tasks from an existing group. The function of this subroutine 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*.

Parameters

group
The group (handle) (IN)

n The number of triplets in array *ranges* (integer) (IN)

ranges
A one-dimensional array of integer triplets of the form (*first_rank, last_rank, stride*) indicating ranks in *group* of tasks to be included in *newgroup* (IN)

newgroup
The new group derived from the preceding parameters in the order defined by *ranges* (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid group

Invalid size
n < 0 or *n* > *groupsize*

Invalid ranks
a computed rank < 0 or >= *groupsize*

Duplicate ranks

Invalid strides
stride[i] = 0

Too many ranks
nranks > groupsize

MPI not initialized

MPI already finalized

Related information

- MPI_GROUP_EXCL
- MPI_GROUP_INCL
- MPI_GROUP_RANGE_EXCL

MPI_GROUP_RANK, MPI_Group_rank

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);
```

C++ synopsis

```
#include mpi.h
int MPI::Group::Get_rank() const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GROUP_RANK(INTEGER GROUP,INTEGER RANK,INTEGER IERROR)
```

Description

This subroutine returns the rank of the local task with respect to *group*. This local operation does not require any intertask communication.

Parameters

group
The group (handle) (IN)

rank
An integer that specifies the rank of the calling task in group or MPI_UNDEFINED if the task is not a member. (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid group

MPI not initialized

MPI already finalized

Related information

- MPI_COMM_RANK

MPI_GROUP_SIZE, MPI_Group_size

Returns the number of tasks in a group.

C synopsis

```
#include <mpi.h>
int MPI_Group_size(MPI_Group group,int *size);
```

C++ synopsis

```
#include mpi.h
int MPI::Group::Get_size() const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GROUP_SIZE(INTEGER GROUP,INTEGER SIZE,INTEGER IERROR)
```

Description

This subroutine returns the number of tasks in a group. This is a local operation and does not require any intertask communication.

Parameters

group
The group (handle) (IN)

size
The number of tasks in the group (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid group

MPI not initialized

MPI already finalized

Related information

- MPI_COMM_SIZE

MPI_GROUP_TRANSLATE_RANKS, MPI_Group_translate_ranks

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Group::Translate_ranks(const MPI::Group& group1, int n,
     const int ranks1[],
     const MPI::Group& group2, int ranks2[]);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GROUP_TRANSLATE_RANKS(INTEGER GROUP1, INTEGER N,
    INTEGER RANKS1(*),INTEGER GROUP2,INTEGER RANKS2(*),INTEGER IERROR)
```

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.

Parameters

group1
The first group (handle) (IN)

n An integer that specifies the number of ranks in *ranks1* and *ranks2* arrays (IN)

ranks1
An array of zero or more valid ranks in *group1* (IN)

group2
The second group (handle) (IN)

ranks2
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
The Fortran return code. It is always the last argument.

Errors

Invalid groups

Invalid rank count
n < **0**

Invalid rank
ranks1[i] < 0 or *ranks1[i]* > = size of *group1*

MPI not initialized

MPI already finalized

Related information

MPI_COMM_COMPARE

MPI_GROUP_UNION, MPI_Group_union

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);
```

C++ synopsis

```
#include mpi.h
static MPI::Group MPI::Group::Union(const MPI::Group& group1,
        const MPI::Group& group2);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_GROUP_UNION(INTEGER GROUP1,INTEGER GROUP2,INTEGER NEWGROUP,
  INTEGER IERROR)
```

Description

This subroutine 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.

Parameters

group1
The first group (handle) (IN)

group2
The second group (handle) (IN)

newgroup
The union group (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid groups

MPI not initialized

MPI already finalized

Related information

- MPI_GROUP_DIFFERENCE
- MPI_GROUP_INTERSECTION

MPI_IBSEND, MPI_Ibsend

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);
```

C++ synopsis

```
#include mpi.h
MPI::Request MPI::Comm::Ibsend(const void* buf, int count,
          const MPI::Datatype& datatype,
          int dest, int tag) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_IBSEND(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,
    INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

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.

Parameters

buf
The initial address of the send buffer (choice) (IN)

count
The number of elements in the send buffer (integer) (IN)

datatype
The data type of each send buffer element (handle) (IN)

dest
The rank of the destination task in *comm* (integer) (IN)

tag
The message tag (positive integer) (IN)

comm
The communicator (handle) (IN)

request
The communication request (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

Make sure you have enough buffer space available. An error occurs if the message must be buffered and there is there is not enough buffer space. The amount of buffer space needed to be safe depends on the expected peak of pending messages. The sum of the sizes of all of the pending messages at that point plus (**MPI_IBSEND_OVERHEAD***number_of_messages*) should be sufficient.

Avoid using **MPI_IBSEND** if possible. It adds overhead because it requires an extra memory-to-memory copy of the outgoing data. If **MPI_IBSEND** is used, the associated receive operations may perform better with **MPI_CSS_INTERRUPT** enabled.

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_BUFFER_ATTACH
- MPI_WAIT

MPI_Info_c2f

Translates a C Info object handle into a Fortran handle to the same Info object.

C synopsis

```
#include <mpi.h>
MPI_Fint MPI_Info_c2f(MPI_Info info);
```

Description

This function does not have C++ or Fortran bindings. MPI_Info_c2f translates a C Info object handle into a Fortran handle to the same Info object; it maps a null handle into a null handle and a non-valid handle into a non-valid handle. The converted handle is returned as the function's value. There is no error detection or return code.

Parameters

info
The Info object (handle) (IN)

Related information

- MPI_Info_f2c

MPI_INFO_CREATE, MPI_Info_create

Creates a new Info object.

C synopsis

```
#include <mpi.h>
int MPI_Info_create(MPI_Info *info);
```

C++ synopsis

```
#include mpi.h
static MPI::Info MPI::Info::Create();
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_INFO_CREATE(INTEGER INFO,INTEGER IERROR)
```

Description

This subroutine creates a new Info object and returns a handle to it in the *info* argument. The new Info object does not contain any (*key,value*) pairs, or hints. Any hints are added to an Info object using MPI_INFO_SET. See "MPI_INFO_SET, MPI_Info_set" on page 372 for information about the **MP_HINTS_FILTERED** environment variable.

Parameters

info
The Info object created (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Fatal errors:

MPI not initialized

MPI already finalized

Related information

- MPI_INFO_DELETE
- MPI_INFO_DUP
- MPI_INFO_FREE
- MPI_INFO_GET
- MPI_INFO_GET_NKEYS
- MPI_INFO_GET_NTHKEY
- MPI_INFO_GET_VALUELEN
- MPI_INFO_SET

MPI_INFO_DELETE, MPI_Info_delete

Deletes a (*key, value*) pair from an Info object.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::Info::Delete(const char* key);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_INFO_DELETE(INTEGER INFO,CHARACTER KEY(*),INTEGER IERROR)
```

Description

This subroutine deletes a (key,value) pair from the Info object referred to by *info*. If the key is unrecognized, the attempt to delete it will be ignored and no error occurs. In other words, an attempt to delete with a key that exists in the object will succeed. An attempt to delete with a recognized key that is not present in the object will raise an error. An attempt to delete with an unrecognized key has no effect. See "MPI_INFO_SET, MPI_Info_set" on page 372 for information about how the **MP_HINTS_FILTERED** environment variable can affect which keys are recognized.

Parameters

info
The Info object (handle) (OUT)

key
The key of the pair to be deleted (string) (IN)

IERROR
The Fortran return code. It is always the last argument.

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

Key not found in info

Related information

- MPI_INFO_CREATE
- MPI_INFO_DUP
- MPI_INFO_FREE
- MPI_INFO_GET

- MPI_INFO_GET_NKEYS
- MPI_INFO_GET_NTHKEY
- MPI_INFO_GET_VALUELEN
- MPI_INFO_SET

MPI_INFO_DUP, MPI_Info_dup

Duplicates an Info object.

C synopsis

```
#include <mpi.h>
int MPI_Info_dup(MPI_Info info,MPI_Info *newinfo);
```

C++ synopsis

```
#include mpi.h
MPI::Info MPI::Info::Dup() const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_INFO_DUP(INTEGER INFO,INTEGER NEWINFO,INTEGER IERROR)
```

Description

This subroutine duplicates the Info object referred to by *info* and returns in *newinfo* a handle to the newly-created object. The new object has the same (*key,value*) pairs and ordering of keys as the old object.

Parameters

info
The Info object to be duplicated(handle) (IN)

newinfo
The new Info object (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

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_FREE
- MPI_INFO_GET
- MPI_INFO_GET_NKEYS
- MPI_INFO_GET_NTHKEY
- MPI_INFO_GET_VALUELEN
- MPI_INFO_SET

MPI_Info_f2c

Returns a C handle to an Info object.

C synopsis

```
#include <mpi.h>
MPI_Info MPI_Info_f2c(MPI_Fint info);
```

Description

This function does not have C++ or Fortran bindings. MPI_Info_f2c returns a C handle to an Info object. If *info* is a valid Fortran handle to an Info object, MPI_Info_f2c returns a valid C handle to that same file. If *info* is set to the Fortran value MPI_INFO_NULL, MPI_Info_f2c returns the equivalent null C handle. If *info* is not a valid Fortran handle, MPI_Info_f2c returns a non-valid C handle. The converted handle is returned as the function's value. There is no error detection or return code.

Parameters

info
The Info object (handle) (IN)

Related information

- MPI_Info_c2f

MPI_INFO_FREE, MPI_Info_free

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Info::Free();
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_INFO_FREE(INTEGER INFO,INTEGER IERROR)
```

Description

MPI_INFO_FREE frees the Info object referred to by the *info* argument and sets *info* to MPI_INFO_NULL.

Parameters

info
The Info object to be freed (handle) (INOUT)

IERROR
The Fortran return code. It is always the last argument.

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_DUP
- MPI_INFO_GET
- MPI_INFO_GET_NKEYS
- MPI_INFO_GET_NTHKEY
- MPI_INFO_GET_VALUELEN
- MPI_INFO_SET

MPI_INFO_GET, MPI_Info_get

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);
```

C++ synopsis

```
#include mpi.h
bool MPI::Info::Get(const char* key, int valuelen, char* value) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_INFO_GET(INTEGER INFO,CHARACTER KEY(*),INTEGER VALUELEN,CHARACTER VALUE(*),
      LOGICAL FLAG,INTEGER IERROR)
```

Description

This subroutine retrieves the value associated with the key in the Info object referred to by *info*. If the (*key,value*) pair is present in the Info object, MPI_INFO_GET sets *flag* to **true** and returns the value in *value*. Otherwise, *flag* is set to **false** and *value* remains unchanged.

Parameters

info
The Info object (handle) (IN)

key
The key (string) (IN)

valuelen
The length of the value argument (integer) (IN)

value
The value (string) (OUT)

flag
Set to **true** if *key* is defined and set to **false** if not (boolean) (OUT)

IERROR
The Fortran return code. It is always the last argument.

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_DELETE
- MPI_INFO_DUP
- MPI_INFO_FREE
- MPI_INFO_GET_NKEYS
- MPI_INFO_GET_NTHKEY
- MPI_INFO_GET_VALUELEN
- MPI_INFO_SET

MPI_INFO_GET_NKEYS, MPI_Info_get_nkeys

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);
```

C++ synopsis

```
#include mpi.h
int MPI::Info::Get_nkeys() const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_INFO_GET_NKEYS(INTEGER INFO,INTEGER NKEYS,INTEGER IERROR)
```

Description

MPI_INFO_GET_NKEYS returns in *nkeys* the number of keys currently defined in the Info object referred to by *info*.

Parameters

info
The Info object (handle) (IN)

nkeys
The number of defined keys (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

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_DUP
- MPI_INFO_FREE
- MPI_INFO_GET
- MPI_INFO_GET_NTHKEY
- MPI_INFO_GET_VALUELEN
- MPI_INFO_SET

MPI_INFO_GET_NTHKEY, MPI_Info_get_nthkey

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Info::Get_nthkey(int n, char* key) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_INFO_GET_NTHKEY(INTEGER INFO,INTEGER N,CHARACTER KEY(*),
      INTEGER IERROR)
```

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** and less than the number of keys returned by MPI_INFO_GET_NKEYS.

Parameters

info
The Info object (handle) (IN)

n The key number (integer) (IN)

key
The key (string) (OUT)

IERROR
The Fortran return code. It is always the last argument.

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_DELETE
- MPI_INFO_DUP
- MPI_INFO_FREE
- MPI_INFO_GET
- MPI_INFO_GET_NKEYS

- MPI_INFO_GET_VALUELEN
- MPI_INFO_SET

MPI_INFO_GET_VALUELEN, MPI_Info_get_valuelen

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);
```

C++ synopsis

```
#include mpi.h
bool MPI::Info::Get_valuelen(const char* key, int& valuelen) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_INFO_GET_VALUELEN(INTEGER INFO,CHARACTER KEY(*),INTEGER VALUELEN,
        LOGICAL FLAG,INTEGER IERROR)
```

Description

This subroutine retrieves the length of the value associated with the key in the Info object referred to by *info*. If *key* is defined, *valuelen* is set to the length of the associated value after it has been converted to a string and *flag* is set to **true**. Otherwise, *flag* is set to **false** and *valuelen* remains unchanged.

Parameters

info
The Info object (handle) (IN)

key
The key (string) (IN)

valuelen
The length of the value associated with **key** (integer) (OUT)

flag
Set to **true** if *key* is defined or **false** if *key* is not defined (boolean) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

Use this subroutine before 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_DELETE
- MPI_INFO_DUP
- MPI_INFO_FREE
- MPI_INFO_GET
- MPI_INFO_GET_NKEYS
- MPI_INFO_GET_NTHKEY
- MPI_INFO_SET

MPI_INFO_SET, MPI_Info_set

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Info::Set(const char* key, const char* value);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_INFO_SET(INTEGER INFO,CHARACTER KEY(*),CHARACTER VALUE(*),INTEGER IERROR)
```

Description

This subroutine adds the (*key,value*) pair to the Info object referred to by *info*, and overrides the value if a value for the same key was previously set. The **MP_HINTS_FILTERED** environment variable determines the behavior of Info object subroutines.

If the variable is set to **no**, or allowed to default, the (*key,value*) pairs are unfiltered, meaning the key and the value may be any strings the user provides. Unfiltered mode is the default behavior required by an MPI 2.1 clarification of MPI_Info semantic. In unfiltered mode, all hints will be recorded in the Info object. There is no way to determine which hints are understood. Unfiltered mode must be used if there is a need for hints other than those supported by IBM PE MPI. This might occur if any additional MPI-like functions layered on IBM PE MPI need to store and retrieve hints.

If the variable is set to **yes**, the (*key,value*) pairs are filtered, meaning only those keys that pertain to supported hints are recognized by MPI_INFO subroutines, may be recorded in, and will be accepted. In filtered mode, an attempt to set an unsupported hint will leave the Info object unchanged. A subsequent MPI_INFO_GET with the key will indicate that the hint is not present. A recognized hint may also be ignored if it has a value that is not valid. This allows the user to detect whether any provided hint is actually supported by IBM PE MPI.

MP_HINTS_FILTERED=yes was the default IBM PE MPI mode before IBM PE 5.1. This has been changed in response to the MPI 2.1 clarification. The new default is not expected to affect existing applications, except that an application that depends on unfiltered (key,value) pairs no longer needs to have an explicit setting for **MP_HINTS_FILTERED** in its environment.

Parameters

info
The Info object (handle) (INOUT)

key
The key (string) (IN)

value
The value (string) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

Only Info object (*key,value*) pairs associated with supported hints and containing valid values will affect MPI subroutines that take an Info object as a parameter. The MP_HINTS_FILTERED variable affects only the behavior of the MPI_INFO subroutines. Unsupported (*key,value*) pairs in an Info object are ignored by the subroutines that accept hints.

For a list of hints that apply to MPI_FILE subroutines, see "MPI_FILE_OPEN, MPI_File_open" on page 226.

For a list of hints that apply to MPI_WIN subroutines, see "MPI_WIN_CREATE, MPI_Win_create" on page 573.

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_DELETE
- MPI_INFO_DUP
- MPI_INFO_FREE
- MPI_INFO_GET
- MPI_INFO_GET_NKEYS
- MPI_INFO_GET_NTHKEY
- MPI_INFO_GET_VALUELEN

MPI_INIT, MPI_Init

Initializes MPI.

C synopsis

```
#include <mpi.h>
int MPI_Init(int *argc,char ***argv);
```

C++ synopsis

```
#include mpi.h
void MPI::Init(int& argc, char**& argv);
```

```
#include mpi.h
void MPI::Init();
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_INIT(INTEGER IERROR)
```

Description

This subroutine initializes MPI. All MPI programs must call MPI_INIT before any other MPI routine (with the exception of MPI_INITIALIZED). More than one call to MPI_INIT by any task is erroneous.

Parameters

IERROR
The Fortran return code. It is always the last argument.

Notes

For either MPI_INIT or MPI_INIT_THREAD, IBM PE MPI normally returns support that is equivalent to MPI_THREAD_MULTIPLE. For more information, see "MPI_INIT_THREAD, MPI_Init_thread" on page 376.

argc and **argv** are the arguments passed to **main**. IBM PE MPI does not examine or modify these arguments when they are passed to MPI_INIT. In accordance with MPI-2, it is valid to pass NULL in place of **argc** and **argv**.

In a threads environment, MPI_INIT needs to be called once per task and not once per thread. You do not 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 threads 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 nonthreads program is relinked using the threads library, the code prior to calling MPI_INIT should be examined with these thoughts in mind.

Also for the threads library, if you had registered a function as a 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 will need to set the environment variable **MP_CSS_INTERRUPT** to **YES** in order to get arriving packets to invoke the interrupt service thread.

Errors

MPI already finalized

MPI already initialized

Related information

- MPI_FINALIZE
- MPI_INITIALIZED
- MPI_INIT_THREAD

MPI_INIT_THREAD, MPI_Init_thread

Initializes MPI and the MPI threads environment.

C synopsis

```
#include <mpi.h>
int MPI_Init_thread(int *argc, char *((*argv)[]), int required,
      int *provided);
```

C++ synopsis

```
#include mpi.h
int MPI::Init_thread(int& argc, char**& argv, int required);
```

```
#include mpi.h
int MPI::Init_thread(int required);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_INIT_THREAD(INTEGER REQUIRED, INTEGER PROVIDED, INTEGER IERROR)
```

Description

This subroutine initializes MPI in the same way that a call to MPI_INIT would. In some implementations, it may do special threads environment initialization. In IBM PE MPI, MPI_INIT_THREAD is equivalent to MPI_INIT. The argument *required* is used to specify the desired level of thread support. The possible values for *required* are listed in increasing order of thread support:

MPI_THREAD_SINGLE
Only one thread will run.

MPI_THREAD_FUNNELED
The task can be multi-threaded, but only the main thread will make MPI calls. All MPI calls are funneled to the main thread.

MPI_THREAD_SERIALIZED
The task can be multi-threaded and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads. All MPI calls are *serialized* by explicit application thread synchronizations.

MPI_THREAD_MULTIPLE
Multiple threads can call MPI with no restrictions.

These values are monotonic: MPI_THREAD_SINGLE, MPI_THREAD_FUNNELED, MPI_THREAD_SERIALIZED, MPI_THREAD_MULTIPLE.

MPI_INIT_THREAD returns information about the actual level of thread support that MPI will provide in the *provided* argument. It can be MPI_THREAD_SINGLE, MPI_THREAD_FUNNELED, MPI_THREAD_SERIALIZED, or MPI_THREAD_MULTIPLE.

Parameters

required
The desired level of thread support (integer) (IN)

provided
The level of thread support that is provided (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

For IBM PE MPI, the required argument is ignored. In normal use, IBM PE MPI always provides a level of thread support equivalent to MPI_THREAD_MULTIPLE. If the **MPI_SINGLE_THREAD** environment variable is set to **yes**, MPI_INIT_THREAD returns MPI_THREAD_FUNNELED.

In C and C++, the passing of *argc* and *argv* is optional. In C, this is accomplished by passing the appropriate null pointer. In C++, this is accomplished with two separate bindings to cover these two cases.

Errors

Fatal errors:

MPI already finalized

MPI already initialized

Unrecognized thread support level
required must be MPI_THREAD_SINGLE, MPI_THREAD_FUNNELED, MPI_THREAD_SERIALIZED, or MPI_THREAD_MULTIPLE.

Related information

- MPI_INIT

MPI_INITIALIZED, MPI_Initialized

Determines whether MPI is initialized.

C synopsis

```
#include <mpi.h>
int MPI_Initialized(int *flag);
```

C++ synopsis

```
#include mpi.h
bool MPI::Is_initialized();
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_INITIALIZED(LOGICAL FLAG,INTEGER IERROR)
```

Description

This subroutine determines if MPI is initialized. MPI_INITIALIZED and MPI_GET_VERSION are the only MPI calls that can be made before MPI_INIT is called.

Parameters

flag
Set to **true** if MPI_INIT was called; otherwise set to **false** (OUT)

IERROR
The Fortran return code. It is always the last argument.

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

Creates an inter-communicator from two intra-communicators.

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);
```

C++ synopsis

```
#include mpi.h
MPI::Intercomm MPI::Intracomm::Create_intercomm(int local_leader,
      const MPI::Comm& peer_comm,
      int remote_leader, int tag) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_INTERCOMM_CREATE(INTEGER LOCAL_COMM,INTEGER LOCAL_LEADER,
    INTEGER PEER_COMM,INTEGER REMOTE_LEADER,INTEGER TAG,
    INTEGER NEWINTERCOM,INTEGER IERROR)
```

Description

This subroutine creates an inter-communicator from two intra-communicators 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.

Parameters

local_comm
The local intra-communicator (handle) (IN)

local_leader
An integer specifying the rank of local group leader in *local_comm* (IN)

peer_comm
The *peer* intra-communicator (significant only at the *local_leader*) (handle) (IN)

remote_leader
The rank of the remote group leader in *peer_comm* (significant only at the *local_leader*) (integer) (IN)

tag
A safe tag (integer) (IN)

newintercom
The new inter-communicator (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Conflicting collective operations on communicator

Invalid communicators

Invalid communicator types
must be intra-communicators

Invalid ranks
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

Creates an intra-communicator by merging the local and remote groups of an inter-communicator.

C synopsis

```
#include <mpi.h>
int MPI_Intercomm_merge(MPI_Comm intercomm,int high,
   MPI_Comm *newintracomm);
```

C++ synopsis

```
#include mpi.h
MPI::Intracomm MPI::Intercomm::Merge(bool high);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_INTERCOMM_MERGE(INTEGER INTERCOMM,LOGICAL HIGH,
      INTEGER NEWINTRACOMM,INTEGER IERROR)
```

Description

This subroutine creates an intra-communicator 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**, the union orders the low group before the high group. If all tasks provided the same *high* argument, the order of the union is arbitrary. MPI_INTERCOMM_MERGE is blocking and collective within the union of the two groups.

Parameters

intercomm
The inter-communicator (handle) (IN)

high
(logical) (IN)

newintracomm
The new intra-communicator (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid communicator

Invalid communicator type
must be inter-communicator

Inconsistent *high* within group

MPI not initialized

MPI already finalized

Related information

MPI_INTERCOMM_CREATE

MPI_IPROBE, MPI_Iprobe

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);
```

C++ synopsis

```
#include mpi.h
bool MPI::Comm::Iprobe(int source, int tag) const;
```

```
#include mpi.h
bool MPI::Comm::Iprobe(int source, int tag, MPI::Status& status) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_IPROBE(INTEGER SOURCE,INTEGER TAG,INTEGER COMM,LOGICAL FLAG,
    INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
```

Description

This subroutine lets you 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*) issued 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 operation processed 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 with any tag or both, 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.

Passing MPI_STATUS_IGNORE for the *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

Some older MPI applications that were written for certain open source MPI implementations include regular calls to MPI_IPROBE, not to detect messages, but to allow the MPI library to make progress. This is neither required nor recommended for IBM PE MPI applications. These artificial MPI_IPROBE calls are not required for program correctness and may hurt performance.

Parameters

source
A source rank or MPI_ANY_SOURCE (integer) (IN)

tag
A tag value or MPI_ANY_TAG (positive integer) (IN)

comm
A communicator (handle) (IN)

flag
(logical) (OUT)

status
A status object (Status) (INOUT). Note that in Fortran a single status object is an array of integers.

IERROR
The Fortran return code. It is always the last argument.

Notes

In a threads environment, MPI_PROBE or MPI_IPROBE followed by MPI_RECV, based on the information from the probe, may not be a threadsafe 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 to ensure the message is not cancelled between the time it is detected by a call to MPI_IPROBE or MPI_PROBE and the time the receive is posted.

Errors

Invalid communicator

Invalid source
source < 0 or *source* > = *groupsize*

Invalid status ignore value

Invalid tag
tag < 0

MPI already finalized

MPI not initialized

Related information

- MPI_PROBE
- MPI_RECV

MPI_IRECV, MPI_Irecv

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);
```

C++ synopsis

```
#include mpi.h
MPI::Request MPI::Comm::Irecv(void *buf, int count, const MPI::Datatype& datatype,
         int source, int tag) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_IRECV(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER SOURCE,
   INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Description

This subroutine 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.

Parameters

buf
The initial address of the receive buffer (choice) (OUT)

count
The number of elements in the receive buffer (integer) (IN)

datatype
The data type of each receive buffer element (handle) (IN)

source
The rank of source or MPI_ANY_SOURCE (integer) (IN)

tag
The message tag or MPI_ANY_TAG (positive integer) (IN)

comm
The communicator (handle) (IN)

request
The communication request (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

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, MPI_Recv" on page 421 for more 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

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);
```

C++ synopsis

```
#include mpi.h
MPI::Request MPI::Comm::Irsend(const void *buf, int count,
          const MPI::Datatype& datatype,
          int dest, int tag) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_IRSEND(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,
      INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Description

MPI_IRSEND starts a ready mode, nonblocking send operation. 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.

Parameters

buf
The initial address of the send buffer (choice) (IN)

count
The number of elements in the send buffer (integer) (IN)

datatype
The data type of each send buffer element (handle) (IN)

dest
The rank of the destination task in *comm* (integer) (IN)

tag
The message tag (positive integer) (IN)

comm
The communicator (handle) (IN)

request
The communication request (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

See "MPI_RSEND, MPI_Rsend" on page 443 for more 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_IS_THREAD_MAIN, MPI_Is_thread_main

Determines whether the calling thread is the thread that called MPI_INIT or MPI_INIT_THREAD.

C synopsis

```
#include <mpi.h>
int MPI_Is_thread_main(int *flag);
```

C++ synopsis

```
#include mpi.h
bool MPI::Is_thread_main();
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_IS_THREAD_MAIN(LOGICAL FLAG, INTEGER IERROR)
```

Description

This subroutine can be called by a thread to find out whether it is the main thread (the thread that called MPI_INIT or MPI_INIT_THREAD). Because MPI_FINALIZE must be called on the same thread that called MPI_INIT or MPI_INIT_THREAD, this subroutine can be used when the identity of the main thread is no longer known.

Parameters

flag
Set to **true** if the calling thread is the main thread; otherwise it is **flase** (logical) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

Errors

Fatal errors:

MPI already finalized

MPI not initialized

Related information

- MPI_INIT
- MPI_INIT_THREAD

MPI_ISEND, MPI_Isend

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);
```

C++ synopsis

```
#include mpi.h
MPI::Request MPI::Comm::Isend(const void *buf, int count,
         const MPI::Datatype& datatype,
         int dest, int tag) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_ISEND(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,
   INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Description

This subroutine 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.

Parameters

buf
The initial address of the send buffer (choice) (IN)

count
The number of elements in the send buffer (integer) (IN)

datatype
The data type of each send buffer element (handle) (IN)

dest
The rank of the destination task in *comm* (integer) (IN)

tag
The message tag (positive integer) (IN)

comm
The communicator (handle) (IN)

request
The communication request (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

See "MPI_SEND, MPI_Send" on page 456 for more 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

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);
```

C++ synopsis

```
#include mpi.h
MPI::Request MPI::Comm::Issend(const void *buf, int count,
          const MPI::Datatype& datatype,
          int dest, int tag) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_ISSEND(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,
    INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

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.

Parameters

buf
The initial address of the send buffer (choice) (IN)

count
The number of elements in the send buffer (integer) (IN)

datatype
The data type of each send buffer element (handle) (IN)

dest
The rank of the destination task in *comm* (integer) (IN)

tag
The message tag (positive integer) (IN)

comm
The communicator (handle) (IN)

request
The communication request (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

See "MPI_SSEND, MPI_Ssend" on page 465 for more 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

Generates a new communicator 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' or USE MPI
MPI_KEYVAL_CREATE(EXTERNAL COPY_FN,EXTERNAL DELETE_FN,
      INTEGER KEYVAL,INTEGER EXTRA_STATE,INTEGER IERROR)
```

Description

This subroutine 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)
```

The *attribute_val_in* parameter is the value of the attribute. The *attribute_val_out* parameter is the address of the value, so the function can set a new value. The *attribute_val_out* parameter is logically a **void****, but it is prototyped as **void***, to avoid the need for complex casting.

You can use these predefined functions:

MPI_DUP_FN
Function to always copy

MPI_NULL_COPY_FN
Function to never copy

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, a specific error code is provided for a *copy_fn* failure and another is provided for a *delete_fn* failure. These error codes can be found in error class **MPI_ERR_OTHER**. The *copy_fn* return code or the *delete_fn* return code is not preserved.

Parameters

copy_fn
The copy callback function for *keyval* (IN)

delete_fn
The delete callback function for *keyval* (IN)

keyval
An integer specifying the key value for future access (OUT)

extra_state
The extra state for callback functions (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_COMM_CREATE_KEYVAL supersedes MPI_KEYVAL_CREATE.

MPI_KEYVAL_CREATE does not inter-operate with MPI_COMM_CREATE_KEYVAL. The Fortran bindings for MPI-1 caching functions presume that an attribute is an INTEGER. The MPI-2 caching bindings use INTEGER (KIND=MPI_ADDRESS_KIND). In an MPI implementation that uses 64-bit addresses and 32-bit INTEGERS, the two formats would be incompatible.

Errors

MPI not initialized

MPI already finalized

Related information

- MPI_ATTR_DELETE
- MPI_ATTR_PUT
- MPI_COMM_DUP
- MPI_COMM_FREE

MPI_KEYVAL_FREE, MPI_Keyval_free

Marks a communicator attribute key for deallocation.

C synopsis

```
#include <mpi.h>
int MPI_Keyval_free(int *keyval);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_KEYVAL_FREE(INTEGER KEYVAL,INTEGER IERROR)
```

Description

This subroutine 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.

Parameters

keyval
The attribute key (integer) (INOUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_COMM_FREE_KEYVAL supersedes MPI_KEYVAL_FREE.

MPI_KEYVAL_FREE does not inter-operate with MPI_COMM_FREE_KEYVAL. The Fortran bindings for MPI-1 caching functions presume that an attribute is an INTEGER. The MPI-2 caching bindings use INTEGER (KIND=MPI_ADDRESS_KIND). In an MPI implementation that uses 64-bit addresses and 32-bit INTEGERS, the two formats would be incompatible.

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_LOOKUP_NAME, MPI_Lookup_name

This function retrieves a *port_name* published by **MPI_PUBLISH_NAME** with *service_name*.

C synopsis

```
#include <mpi.h>
int MPI_Lookup_name(char *service_name, MPI_Info info, char *port_name);
```

C++ synopsis

```
#include <mpi.h>
void MPI::Lookup_name(const char* service_name, const MPI::Info& info,
char* port_name);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
CHARACTER*(*) SERVICE_NAME, PORT_NAME
INTEGER INFO, IERROR
```

Description

This function retrieves a *port_name* published by **MPI_PUBLISH_NAME** with *service_name*. If *service_name* has not been published, it raises an error in the error class **MPI_ERR_NAME**.

The application must supply a *port_name* buffer large enough to hold the largest possible port name. The constant **MPI_MAX_PORT_NAME** can be used to allocate enough space.

Parameters

service_name
A service name (string) (IN)

info
An info is an object containing {key,value} pairs. IBM PE MPI **MPI_LOOKUP_NAME** does not recognize any info keys. **MPI_INFO_NULL** is always valid (IN)

port_name
A port name (string) (OUT)

Errors

Invalid service name passed to MPI_LOOKUP_NAME

Invalid port name (NULL)

Related information

- **MPI_PUBLISH_NAME**

MPI_Op_c2f

Translates a C reduction operation handle into a Fortran handle to the same operation.

C synopsis

```
#include <mpi.h>
MPI_Fint MPI_Op_c2f(MPI_Op op);
```

Description

This function does not have C++ or Fortran bindings. MPI_Op_c2f translates a C reduction operation handle into a Fortran handle to the same operation; it maps a null handle into a null handle and a non-valid handle into a non-valid handle. The converted handle is returned as the function's value. There is no error detection or return code.

Parameters

op The reduction operation (handle) (IN)

Related information

- MPI_Op_f2c

MPI_OP_COMMUTATIVE, MPI_Op_commutative

Queries reduction operations for commutativity.

C synopsis

```
#include <mpi.h>
int MPI_Op_commutative(MPI_Op op, int *commute);
```

C++ synopsis

```
#include mpi.h
bool MPI::Op::Is_commutative() const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR)
   LOGICAL COMMUTE
   INTEGER OP, IERROR
```

Description

This function queries reduction operations for commutativity.

Parameters

op Reduction operation (handle) (IN)

commute
True if *op* is commutative, false otherwise (logical) (OUT)

Errors

Invalid MPI_Op

MPI_OP_CREATE, MPI_Op_create

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Op::Init(MPI::User_function *func, bool commute);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_OP_CREATE(EXTERNAL FUNCTION,LOGICAL COMMUTE,INTEGER OP,INTEGER IERROR)
```

Description

This subroutine binds a user-defined reduction operation to an **op** handle, which you can then use in MPI_REDUCE, MPI_ALLREDUCE, MPI_REDUCE_SCATTER, MPI_SCAN, and MPI_EXSCAN.

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 a 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
```

Parameters

function
The user-defined reduction function (function) (IN)

commute
Set to **true** if commutative; otherwise it is **false** (IN)

op The reduction operation (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for information about reduction functions.

Errors

Null function

MPI not initialized

MPI already finalized

Related information

- MPI_ALLREDUCE
- MPI_OP_FREE
- MPI_REDUCE
- MPI_REDUCE_SCATTER
- MPI_SCAN

MPI_Op_f2c

Returns a C reduction operation handle to an operation.

C synopsis

```
#include <mpi.h>
MPI_Op MPI_Op_f2c(MPI_Fint op);
```

Description

This function does not have C++ or Fortran bindings. MPI_Op_f2c returns a C handle to an operation. If *op* is a valid Fortran handle to n operation, MPI_Op_f2c returns a valid C handle to that same group. If *op* is set to the Fortran value MPI_OP_NULL, MPI_Op_f2c returns the equivalent null C handle. If *op* is not a valid Fortran handle, MPI_Op_f2c returns a non-valid C handle. The converted handle is returned as the function's value. There is no error detection or return code.

Parameters

op The reduction operation (handle) (IN)

Related information

- MPI_Op_c2f

MPI_OP_FREE, MPI_Op_free

Marks a user-defined reduction operation for deallocation.

C synopsis

```
#include <mpi.h>
int MPI_Op_free(MPI_Op *op);
```

C++ synopsis

```
#include mpi.h
void MPI::Op::Free();
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_OP_FREE(INTEGER OP,INTEGER IERROR)
```

Description

This subroutine marks a reduction operation for deallocation, and set **op** to MPI_OP_NULL. Actual deallocation occurs when the operation's reference count is zero.

Parameters

op The reduction operation (handle) (INOUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid operation

Predefined operation

MPI not initialized

MPI already finalized

Related information

- MPI_OP_CREATE

MPI_OPEN_PORT, MPI_Open_port

Establishes a network address at which the server is able to accept connections from clients.

C synopsis

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

C++ synopsis

```
#include <mpi.h>
void MPI::Open_port(const MPI::Info& info, char* port_name);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_OPEN_PORT(INFO, PORT_NAME, IERROR)
CHARACTER*(*) PORT_NAME
INTEGER INFO, IERROR
```

Description

This subroutine establishes a network address, encoded in the *port_name* string, at which the server is able to accept connections from clients. MPI copies a system-supplied port name into *port_name*. *port_name* identifies the newly opened port and can be used by a client to contact the server.

The application must supply a *port_name* buffer large enough to hold the largest possible port name. The constant **MPI_MAX_PORT_NAME** can be used to allocate enough space.

Parameters

info
: An info is an object containing {key,value} pairs. IBM PE MPI **MPI_OPEN_PORT** does not recognize any info keys. **MPI_INFO_NULL** is always valid (IN)

port_name
: A newly established port (string) (OUT)

Notes

The system copies the port name into *port_name*. The application must pass a buffer of sufficient size to hold this value.

Related information

- **MPI_CLOSE_PORT**
- **MPI_COMM_ACCEPT**
- **MPI_COMM_CONNECT**

MPI_PACK, MPI_Pack

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Datatype::Pack(const void* inbuf, int incount, void* outbuf,
    int outsize, int& position, const MPI::Comm& comm)
    const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_PACK(CHOICE INBUF,INTEGER INCOUNT,INTEGER DATATYPE,CHOICE OUTBUF,
  INTEGER OUTSIZE,INTEGER POSITION,INTEGER COMM,INTEGER IERROR)
```

Description

This subroutine 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.

Parameters

inbuf
The input buffer start (choice) (IN)

incount
An integer specifying the number of input data items (IN)

datatype
The data type of each input data item (handle) (IN)

outbuf
The output buffer start (choice) (OUT)

outsize
An integer specifying the output buffer size in bytes (OUT)

position
The current position in the output buffer counted in bytes (integer) (INOUT)

comm
The communicator for sending the packed message (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_PACK must be used with some care in 64-bit applications because *outsize* and *position* are integers and can be subject to overflow.

Errors

Invalid incount
incount < 0

Invalid datatype

Type not committed

Invalid communicator

Outbuf too small

Negative length or position for buffer
outsize < 0 or *position* < 0

MPI not initialized

MPI already finalized

Related information

- MPI_PACK_SIZE
- MPI_UNPACK

MPI_PACK_EXTERNAL, MPI_Pack_external

Packs the message in the specified send buffer into the specified buffer space, using the external32 data format.

C synopsis

```
#include <mpi.h>
int MPI_Pack_external(char *datarep, void *inbuf, int incount,
        MPI_Datatype datatype, void *outbuf,
        MPI_Aint outsize, MPI_Aint *position);
```

C++ synopsis

```
#include mpi.h
void MPI::Datatype::Pack_external(const char* datarep, const void* inbuf,
      int incount, void* outbuf, MPI::Aint outsize,
      MPI_Aint& position) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_PACK_EXTERNAL(CHARACTER*(*) DATAREP, CHOICE INBUF(*), INTEGER INCOUNT,
    INTEGER DATATYPE, CHOICE OUTBUF(*), INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE,
    INTEGER(KIND=MPI_ADDRESS_KIND) POSITION, INTEGER IERROR)
```

Description

This subroutine 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.

If you are using IBM PE for Linux, note that MPI_PACK_EXTERNAL is currently not supported on IBM System x servers.

Parameters

datarep
The data representation (string) (IN)

inbuf
The input buffer start (choice) (IN)

incount
An integer specifying the number of input data items (IN)

datatype
The data type of each input data item (handle) (IN)

outbuf
The output buffer start (choice) (OUT)

outsize
An integer specifying the output buffer size, in bytes (IN)

position
The current position in the output buffer, in bytes (integer) (INOUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

In Fortran, MPI_PACK_EXTERNAL returns an argument of type INTEGER(KIND=MPI_ADDRESS_KIND), where type MPI_Aint is used in C. Such variables may be declared as INTEGER*4 in purely 32-bit codes and as INTEGER*8 in 64-bit codes; KIND=MPI_ADDRESS_KIND works correctly in either mode.

Errors

Invalid datarep

Invalid datatype

Invalid incount
incount < 0

Negative length or position for buffer
outsize < 0 or *position* < 0

Outbuf too small

Type not committed

MPI already finalized

MPI not initialized

Related information

- MPI_PACK_EXTERNAL_SIZE
- MPI_UNPACK_EXTERNAL

MPI_PACK_EXTERNAL_SIZE, MPI_Pack_external_size

Returns the number of bytes required to hold the data, using the external32 data format.

C synopsis

```
#include <mpi.h>
int MPI_Pack_external_size(char *datarep, int incount,
    MPI_Datatype datatype,MPI_Aint *size);
```

C++ synopsis

```
#include mpi.h
MPI::Aint MPI::Datatype::Pack_external_size(const char* datarep, int incount) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_PACK_EXTERNAL_SIZE(CHARACTER*(*) DATAREP, INTEGER INCOUNT, INTEGER DATATYPE,
         INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, INTEGER IERROR
```

Description

This subroutine returns the number of bytes required to pack *incount* replications of the data type. You can use MPI_PACK_EXTERNAL_SIZE to determine the size required for a packing buffer.

If you are using IBM PE for Linux, note that MPI_PACK_EXTERNAL_SIZE is currently not supported on IBM System x servers.

Parameters

datarep
The data representation (string) (IN)

incount
An integer specifying the number of input data items (IN)

datatype
The data type of each input data item (handle) (IN)

size
The size of the output buffer, in bytes (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

In Fortran, MPI_PACK_EXTERNAL_SIZE returns a size argument of type INTEGER(KIND=MPI_ADDRESS_KIND), where type MPI_Aint is used in C. Such variables may be declared as INTEGER*4 in purely 32-bit codes and as INTEGER*8 in 64-bit codes; KIND=MPI_ADDRESS_KIND works correctly in either mode.

Errors

Invalid datarep

Invalid datatype

Type is not committed

MPI not initialized

MPI already finalized

Invalid incount
incount < 0

Related information

- MPI_PACK_EXTERNAL
- MPI_UNPACK_EXTERNAL

MPI_PACK_SIZE, MPI_Pack_size

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);
```

C++ synopsis

```
#include mpi.h
int MPI::Datatype::Pack_size(int incount, const MPI::Comm& comm) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_PACK_SIZE(INTEGER INCOUNT,INTEGER DATATYPE,INTEGER COMM,
       INTEGER SIZE,INTEGER IERROR)
```

Description

This subroutine returns the number of bytes required to pack *incount* replications of the data type. You can use MPI_PACK_SIZE to determine the size required for a packing buffer or to track space needed for buffered sends.

Parameters

incount
An integer specifying the count argument to a packing call (IN)

datatype
The data type argument to a packing call (handle) (IN)

comm
The communicator to a packing call (handle) (IN)

size
The size of packed message in bytes (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_PACK_SIZE must be used with some care in 64-bit applications because the *size* argument is an integer and can be subject to overflow.

Errors

Invalid datatype

Type is not committed

MPI not initialized

MPI already finalized

Invalid communicator

Invalid incount
incount < 0

Size overflow
64-bit applications only

Related information

- MPI_PACK

MPI_PCONTROL, MPI_Pcontrol

Provides profiler control.

C synopsis

```
#include <mpi.h>
int MPI_Pcontrol(const int level, ...);
```

C++ synopsis

```
#include mpi.h
void MPI::Pcontrol(const int level, ...);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_PCONTROL(INTEGER LEVEL, ...)
```

Description

MPI_PCONTROL is a placeholder to let applications run with or without an independent profiling package without modification. MPI implementations do not use this subroutine and do not have any control of the implementation of the profiling code.

Calls to this subroutine 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.

Parameters

level
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
The Fortran return code. It is always the last argument.

Errors

MPI does not report any errors for MPI_PCONTROL.

MPI_PROBE, MPI_Probe

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Comm::Probe(int source, int tag) const;
```

```
#include mpi.h
void MPI::Comm::Probe(int source, int tag, MPI::Status& status) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_PROBE(INTEGER SOURCE,INTEGER TAG,INTEGER COMM,
   INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
```

Description

MPI_PROBE behaves like MPI_IPROBE. It lets you 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.

Passing MPI_STATUS_IGNORE for the *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

Parameters

source
A source rank or MPI_ANY_SOURCE (integer) (IN)

tag
A source tag or MPI_ANY_TAG (positive integer) (IN)

comm
A communicator (handle) (IN)

status
A status object (Status) (INOUT). Note that in Fortran a single status object is an array of integers.

IERROR
The Fortran return code. It is always the last argument.

Notes

In a threads environment, MPI_PROBE or MPI_IPROBE followed by MPI_RECV, based on the information from the probe, may not be a threadsafe operation. You must make sure 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 to ensure the message is not cancelled between the time it is detected by a call to MPI_IPROBE or MPI_PROBE and the time the receive is posted.

Errors

Invalid source
source < 0 or *source* > = *groupsize*

Invalid status ignore value

Invalid tag
tag < 0

Invalid communicator

MPI not initialized

MPI already finalized

Related information

- MPI_IPROBE
- MPI_RECV

MPI_PUBLISH_NAME, MPI_Publish_name

Publishes the *port_name/service_name* pair so that an application can retrieve a system-supplied port name using a well-known service name.

C synopsis

```
#include <mpi.h>
int MPI_Publish_name(char *service_name, MPI_Info info, char *port_name);
```

C++ synopsis

```
#include <mpi.h>
void MPI::Publish_name(const char* service_name, const MPI::Info& info,
const char* port_name);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
INTEGER INFO, IERROR
CHARACTER*(*) SERVICE_NAME, PORT_NAME
```

Description

This subroutine publishes the *port_name/service_name* pair so that an application can retrieve a system-supplied port name using a well-known service name.

The *scope* of a published service name is defined as the domain over which the service name is unique and, conversely, the domain over which the (port name, service name) pair may be retrieved. For IBM PE MPI, the scope of both *service_name* and *port_name* is global within the communication *universe*. Each **poe** invocation defines a distinct universe.

Parameters

service_name
A service name to associate with the port (string) (IN)

info
An info is an object containing {key,value} pairs. IBM PE MPI **MPI_PUBLISH_NAME** does not recognize any info keys. **MPI_INFO_NULL** is always valid (IN)

port_name
A port name (string) (IN)

Errors

Invalid service name (NULL)

Invalid port name (NULL)

Related information

- **MPI_UNPUBLISH_NAME**
- **MPI_LOOKUP_NAME**

MPI_PUT, MPI_Put

Transfers data from the origin task to a window at the target task.

C synopsis

```
#include <mpi.h>
int MPI_Put (void *origin_addr, int origin_count,
             MPI_Datatype origin_datatype, int target_rank,
             MPI_Aint target_disp, int target_count,
             MPI_Datatype target_datatype, MPI_Win win);
```

C++ synopsis

```
#include mpi.h
void MPI::Win::Put(const void* origin_addr, int origin_count,
     const MPI::Datatype& origin_datatype, int target_rank,
                   MPI::Aint target_disp, int target_count,
     const MPI::Datatype& target_datatype) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_PUT(CHOICE ORIGIN_ADDR, INTEGER ORIGIN_COUNT, INTEGER ORIGIN_DATATYPE,
        INTEGER TARGET_RANK, INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP,
        INTEGER TARGET_COUNT, INTEGER TARGET_DATATYPE, INTEGER WIN, INTEGER IERROR)
```

Description

MPI_PUT transfers *origin_count* successive entries of the type specified by *origin_datatype*, starting at address *origin_addr* on the origin task to the target task specified by *win* and *target_rank*.

The data are written in the target buffer at address (*target_addr* = *window_base* + *target_disp* * *disp_unit*), where *window_base* and *disp_unit* are the base address and window displacement unit specified at window initialization, by the target task. The target buffer is specified by the arguments *target_count* and *target_datatype*.

The data transfer is the same as that which would occur if the origin task issued a send operation with arguments *origin_addr*, *origin_count*, *origin_datatype*, *target_rank*, *tag*, *comm*, and the target task issued a receive operation with arguments *target_addr*, *target_count*, *target_datatype*, *source*, *tag*, *comm*, where *target_addr* is the target buffer address computed as shown in the previous paragraph, and *comm* is a communicator for the group of *win*.

The communication must satisfy the same constraints as for a similar message-passing communication. The *target_datatype* may not specify overlapping entries in the target buffer. The message sent must fit, without truncation, in the target buffer. Furthermore, the target buffer must fit in the target window.

The *target_datatype* argument is a handle to a data type object that is defined at the origin task, even though it defines a data layout in the target task memory. This does not cause any problems in a homogeneous environment. In a heterogeneous environment, only portable data types are valid.

The data type object is interpreted at the target task. The outcome is as if the target data type object were defined at the target task, by the same sequence of calls used to define it at the origin task. The target data type must contain relative displacements, not absolute addresses.

Parameters

origin_addr
The initial address of the origin buffer (choice) (IN)

origin_count
The number of entries in origin buffer (nonnegative integer) (IN)

origin_datatype
The data type of each entry in the origin buffer (handle) (IN)

target_rank
The rank of the target (nonnegative integer) (IN)

target_disp
The displacement from the start of the window to the target buffer (nonnegative integer) (IN)

target_count
The number of entries in the target buffer (nonnegative integer) (IN)

target_datatype
The data type of each entry in the target buffer (handle) (IN)

win
The window object used for communication (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_PUT is a special case of MPI_ACCUMULATE, with the operation MPI_REPLACE. Note, however, that MPI_PUT and MPI_ACCUMULATE have different constraints on concurrent updates.

MPI_PUT does not require that data move from origin to target until some synchronization occurs. IBM PE MPI may try to combine multiple puts to a target within an epoch into a single data transfer. The user must not modify the source buffer or make any assumption about the contents of the destination buffer until after a synchronization operation has closed the epoch.

On some systems, there may be reasons to use special memory for one-sided communication buffers. MPI_ALLOC_MEM may be the preferred way to allocate buffers on these systems. With IBM PE MPI, there is no advantage to using MPI_ALLOC_MEM, but you can use it to improve the portability of your MPI code.

MPI_PUT is more efficient than MPI_ACCUMULATE with MPI_REPLACE because MPI_PUT does not provide any guarantee for conflicting updates of a target object. For example, if more than one origin does an MPI_ACCUMULATE of MPI_LONGs with MPI_REPLACE to the same target and they touch the same memory range, MPI_ACCUMULATE will ensure each individual MPI_LONG replacement is atomic. With conflicting MPI_PUTs there is a risk that some bytes of the MPI_LONG will be from one MPI_PUT and some bytes will be from another MPI_PUT.

Use MPI_PUT if you can be confident that the RMAs in a single epoch will never overlap in the target memory, and use MPI_ACCUMULATE with MPI_REPLACE if conflicting updates are possible.

Errors

Invalid origin count (*count***)**

Invalid origin datatype (*handle***)**

Invalid target rank (*rank***)**

Invalid target displacement (*value***)**

Invalid target count (*count***)**

Invalid target datatype (*handle***)**

Invalid window handle (*handle***)**

Target outside access group

Origin buffer too small (*size***)**

Target buffer ends outside target window

Target buffer starts outside target window

RMA communication call outside access epoch

RMA communication call in progress

RMA synchronization call in progress

MPI not initialized

MPI already finalized

Related information

- MPI_ACCUMULATE
- MPI_GET
- MPI_WIN_COMPLETE
- MPI_WIN_FENCE
- MPI_WIN_LOCK
- MPI_WIN_POST
- MPI_WIN_START
- MPI_WIN_TEST
- MPI_WIN_UNLOCK
- MPI_WIN_WAIT

MPI_QUERY_THREAD, MPI_Query_thread

Returns the current level of threads support.

C synopsis

```
#include <mpi.h>
int MPI_Query_thread(int *provided);
```

C++ synopsis

```
#include mpi.h
int MPI::Query_thread();
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_QUERY_THREAD(INTEGER PROVIDED, INTEGER IERROR)
```

Description

This subroutine returns the current level of thread support in the *provided* argument. This will be the value returned in *provided* by MPI_INIT_THREAD, if MPI was initialized by a call to MPI_INIT_THREAD. The possible values for *provided* are listed in increasing order of thread support:

MPI_THREAD_SINGLE
Only one thread will run.

MPI_THREAD_FUNNELED
The task can be multi-threaded, but only the main thread will make MPI calls. All MPI calls are funneled to the main thread.

MPI_THREAD_SERIALIZED
The task can be multi-threaded and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads. All MPI calls are *serialized* by explicit application thread synchronizations.

MPI_THREAD_MULTIPLE
Multiple threads can call MPI with no restrictions.

These values are monotonic: MPI_THREAD_SINGLE, MPI_THREAD_FUNNELED, MPI_THREAD_SERIALIZED, MPI_THREAD_MULTIPLE.

Parameters

provided
The level of thread support that is provided (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

In normal use, IBM PE MPI always provides a level of thread support equivalent to MPI_THREAD_MULTIPLE. If the **MPI_SINGLE_THREAD** environment variable is set to **yes**, MPI_QUERY_THREAD returns MPI_THREAD_FUNNELED.

Errors

Fatal errors:

MPI already finalized

MPI not initialized

Related information

- MPI_INIT_THREAD

MPI_RECV, MPI_Recv

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Comm::Recv(void* buf, int count, const MPI::Datatype& datatype,
       int source, int tag) const;
```

```
#include mpi.h
void MPI::Comm::Recv(void* buf, int count, const MPI::Datatype& datatype,
       int source, int tag, MPI::Status& status) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_RECV(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER SOURCE,
      INTEGER TAG,INTEGER COMM,INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
```

Description

MPI_RECV is a blocking receive operation. 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.

Passing MPI_STATUS_IGNORE for the *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

Parameters

buf
The initial address of the receive buffer (choice) (OUT)

count
The number of elements to be received (integer) (IN)

datatype
The data type of each receive buffer element (handle) (IN)

source
The rank of the source task in *comm* or MPI_ANY_SOURCE (integer) (IN)

tag
The message tag or MPI_ANY_TAG (positive integer) (IN)

comm
The communicator (handle) (IN)

status
The status object (Status) (INOUT). Note that in Fortran a single status object is an array of integers.

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid count
count < **0**

Invalid datatype

Invalid status ignore value

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_SEND
- MPI_SENDRECV

MPI_RECV_INIT, MPI_Recv_init

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);
```

C++ synopsis

```
#include mpi.h
MPI::Prequest MPI::Comm::Recv_init(void* buf, int count,
       const MPI::Datatype& datatype,
       int source, int tag) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_RECV_INIT(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,
      INTEGER SOURCE,INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Description

This subroutine creates a persistent communication request for a receive operation. A communication started with MPI_RECV_INIT is completed by a call to one of the MPI wait or test operations. 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.

Parameters

buf
The initial address of the receive buffer (choice) (OUT)

count
The number of elements to be received (integer) (IN)

datatype
The type of each element (handle) (IN)

source
The rank of source or MPI_ANY_SOURCE (integer) (IN)

tag
The tag or MPI_ANY_TAG (integer) (IN)

comm
The communicator (handle) (IN)

request
The communication request (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

See “MPI_RECV, MPI_Recv” on page 421 for more 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_IRECV
- MPI_START

MPI_REDUCE, MPI_Reduce

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Comm::Reduce(const void* sendbuf, void* recvbuf, int count,
         const MPI::Datatype& datatype, const MPI::Op& op,
         int root) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_REDUCE(CHOICE SENDBUF,CHOICE RECVBUF,INTEGER COUNT,
    INTEGER DATATYPE,INTEGER OP,INTEGER ROOT,INTEGER COMM,
    INTEGER IERROR)
```

Description

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

The input buffer and the output buffer have the same number of elements with the same type. The arguments *sendbuf*, *count*, and *datatype* define the send or input buffer. The arguments *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, the reduction operation is processed element-wise on each entry of the sequence. This is 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), *recvbuf(1)* = global max(*sendbuf(1)*) and *recvbuf(2)* = global max(*sendbuf(2)*).

Users can define their own operations or use the predefined operations provided by MPI. User-defined operations can be overloaded to operate on several data types, either basic or derived. The argument *datatype* of MPI_REDUCE must be compatible with *op*.

The parameter *op* may be a predefined reduction operation or a user-defined function, created using MPI_OP_CREATE. This is a list of predefined reduction operations:

Operation
Definition

MPI_BAND
Bitwise AND

MPI_BOR
Bitwise OR

MPI_BXOR
Bitwise XOR

MPI_LAND
Logical AND

MPI_LOR
Logical OR

MPI_LXOR
Logical XOR

MPI_MAX
Maximum value

MPI_MAXLOC
Maximum value and location

MPI_MIN
Minimum value

MPI_MINLOC
Minimum value and location

MPI_PROD
Product

MPI_SUM
Sum

The *in place* option for intra-communicators is specified by passing the value MPI_IN_PLACE to the argument *sendbuf* at the root. In this case, the input data is taken at the root from the receive buffer, where it will be replaced by the output data.

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

MPI_IN_PLACE is not supported for inter-communicators.

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator occur in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

Parameters

sendbuf
The address of the send buffer (choice) (IN)

recvbuf
The address of the receive buffer (choice, significant only at *root*) (OUT)

count
The number of elements in the send buffer (integer) (IN)

datatype
The data type of elements of the send buffer (handle) (IN)

op The reduction operation (handle) (IN)

root
The rank of the root task (integer) (IN)

comm
The communicator (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

See *IBM Parallel Environment Runtime Edition: MPI Programming Guide*.

The MPI standard urges MPI implementations to use the same evaluation order for reductions every time, even if this negatively affects performance. IBM PE MPI adjusts its reduce algorithms for the optimal performance on a given task distribution. The MPI standard suggests, but does not mandate, this sacrifice of performance. IBM PE MPI maintains a balance between performance and the MPI standard's recommendation. IBM PE MPI does not promise that any two runs with the same task count will give the same answer, in the least significant bits, for floating point reductions. Changes to evaluation order may produce different rounding effects. However, IBM PE MPI does promise that two calls to MPI_REDUCE (or MPI_ALLREDUCE) on the same communicator with the same inputs, or two runs that use the same task count and the same distribution across nodes, will always give identical results.

In the 64-bit library, this function uses a shared memory optimization among the tasks on a node. This optimization is discussed in the chapter *Using shared memory* of *IBM Parallel Environment Runtime Edition: MPI Programming Guide*, and is enabled by default. This optimization is not available to 32-bit programs.

Errors

Fatal errors:

Invalid count
count < **0**

Invalid datatype

Type not committed

Invalid *op*

Invalid root

For an intra-communicator: *root* < **0** or *root* >= *groupsize*

For an inter-communicator: *root* < **0** and is neither MPI_ROOT nor MPI_PROC_NULL, or *root* >= *groupsize* of the remote group

Invalid communicator

Unequal message lengths

Invalid use of MPI_IN_PLACE

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_OP_CREATE
- MPI_REDUCE_SCATTER
- MPI_SCAN

MPI_REDUCE_LOCAL, MPI_Reduce_local

Applies the operation given by *op* element-wise to the elements of *inbuf* and *inoutbuf*.

C synopsis

```
#include <mpi.h>
int MPI_Reduce_local(void* inbuf, void* inoutbuf, int count,
    MPI_Datatype datatype, MPI_Op op);
```

C++ synopsis

```
#include mpi.h
void MPI::Op::Reduce_local(const void* inbuf, void* inoutbuf, int count,
         const MPI::Datatype& datatype) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_REDUCE_LOCAL(INBUF, INOUTBUF, COUNT, DATATYPE,
   OP, IERROR)
   <type> INBUF(*), INOUTBUG(*), INTEGER COUNT, DATATYPE, OP
          IERROR
```

Description

This function, applies the operation given by *op* element-wise to the elements of *inbuf* and *inoutbuf*. Both *inbuf* and *inoutbuf* (input as well as result) have the same number of elements given by count and the same datatype given by datatype. The **MPI_IN_PLACE** option is not allowed.

Parameters

inbuf
Input buffer (choice) (IN)

inoutbuf
Combined input and output buffer (choice) (INOUT)

count
Number of elements in the *inbuf* and *inoutbuf* buffers (non-negative integer) (IN)

datatype
Data type of elements of the *inbuf* and *inoutbuf* buffers (handle) (IN)

op Reduction operation (handle) (IN)

Errors

Invalid MPI_Op

Invalid MPI_Datatype

MPI_REDUCE_SCATTER, MPI_Reduce_scatter

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Comm::Reduce_scatter(const void* sendbuf, void* recvbuf,
          int recvcounts[],
          const MPI::Datatype& datatype,
          const MPI::Op& op) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_REDUCE_SCATTER(CHOICE SENDBUF,CHOICE RECVBUF,INTEGER RECVCOUNTS(*),
     INTEGER DATATYPE,INTEGER OP,INTEGER COMM,INTEGER IERROR)
```

Description

This subroutine first performs an element-wise reduction on a vector of *count* = the sum of *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*.

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*.

The *in place* option for intra-communicators is specified by passing MPI_IN_PLACE in the sendbuf argument. In this case, the input data is taken from the top of the receive buffer. The area occupied by the input data may be either longer or shorter than the data filled by the output data.

If *comm* is an inter-communicator, the result of the reduction of the data provided by tasks in group A is scattered among tasks in group B, and vice versa. Within each group, all tasks provide the same *recvcounts* argument, and the sum of the *recvcounts* entries should be the same for the two groups.

MPI_IN_PLACE is not supported for inter-communicators.

The parameter *op* may be a predefined reduction operation or a user-defined function, created using MPI_OP_CREATE. This is a list of predefined reduction operations:

Operation
Definition

MPI_BAND
Bitwise AND

MPI_BOR
Bitwise OR

MPI_BXOR
Bitwise XOR

MPI_LAND
Logical AND

MPI_LOR
Logical OR

MPI_LXOR
Logical XOR

MPI_MAX
Maximum value

MPI_MAXLOC
Maximum value and location

MPI_MIN
Minimum value

MPI_MINLOC
Minimum value and location

MPI_PROD
Product

MPI_SUM
Sum

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator occur in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

Parameters

sendbuf
The starting address of the send buffer (choice) (IN)

recvbuf
The starting address of the receive buffer (choice) (OUT)

recvcounts
An integer array specifying the number of elements in result distributed to each task. Must be identical on all calling tasks. (IN)

datatype
The data type of elements in the input buffer (handle) (IN)

op The reduction operation (handle) (IN)

comm
The communicator (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

See *IBM Parallel Environment Runtime Edition: MPI Programming Guide*.

The MPI standard urges MPI implementations to use the same evaluation order for reductions every time, even if this negatively affects performance. IBM PE MPI adjusts its reduce algorithms for the optimal performance on a given task distribution. The MPI standard suggests, but does not mandate, this sacrifice of performance. IBM PE MPI maintains a balance between performance and the MPI standard's recommendation. IBM PE MPI does not promise that any two runs with the same tack count will give the same answer, in the least significant bits, for floating point reductions. Changes to evaluation order may produce different rounding effects. However, IBM PE MPI does promise that two calls to MPI_REDUCE (or MPI_ALLREDUCE) on the same communicator with the same inputs, or two runs that use the same task count and the same distribution across nodes, will always give identical results.

In the 64-bit library, this function uses a shared memory optimization among the tasks on a node. This optimization is discussed in the chapter *Using shared memory* of *IBM Parallel Environment Runtime Edition: MPI Programming Guide*, and is enabled by default. This optimization is not available to 32-bit programs.

Errors

Fatal errors:

Invalid recvcounts
: **recvcounts[i]** < 0

Invalid datatype

Type not committed

Invalid op

Invalid communicator

Unequal message lengths

Invalid use of MPI_IN_PLACE

MPI not initialized

MPI already finalized

Develop mode error if:

Inconsistent op

Inconsistent datatype

Related information

- MPE_IREDUCE_SCATTER
- MPI_OP_CREATE
- MPI_REDUCE

MPI_REDUCE_SCATTER_BLOCK, MPI_Reduce_scatter_block

Performs a global, element-wise reduction on vectors of count = *n*recvcount* elements in the send buffers defined by *sendbuf*, *count* and *datatype*, using the operation *op*, where *n* is the number of processes in the group of *comm*.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::Comm::Reduce_scatter_block(const void* sendbuf,
     void* recvbuf, int recvcount,
          const MPI::Datatype& datatype,
          const MPI::Op& op);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_REDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT,
     DATATYPE, OP, COMM, IERROR)
   <type> SENDBUF(*), RECVBUF(*)
   INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR
```

Description

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

The *in place* option for intra-communicators is specified by passing **MPI_IN_PLACE** in the *sendbuf* argument. In this case, the input data is taken from the top of the receive buffer.

Parameters

sendbuf
The starting address of the send buffer (choice) (IN)

recvbuf
The starting address of the receive buffer (choice) (OUT)

recvcount
The element count per block (non-negative integer) (IN)

datatype
The data type of elements of the send and receive buffers (handle) (IN)

op The reduction operation (handle) (IN)

comm
The communicator (handle) (IN)

Errors

Fatal errors:

Invalid communicators

negative recvcount

Incorrect use of MPI_IN_PLACE

MPI_Op is invalid

Related information

- MPI_REDUCE_SCATTER

MPI_REGISTER_DATAREP, MPI_Register_datarep

Registers a data representation.

C synopsis

```
#include <mpi.h>
int MPI_Register_datarep(char *datarep,
                    MPI_Datarep_conversion_function *read_conversion_fn,
                    MPI_Datarep_conversion_function *write_conversion_fn,
                    MPI_Datarep_extent_function *dtype_file_extent_fn,
                    void *extra_state);
```

C++ synopsis

```
#include mpi.h
void MPI::Register_datarep(const char* datarep,
                           MPI::Datarep_conversion_function* read_conversion_fn,
                           MPI::Datarep_conversion_function* write_conversion_fn,
                           MPI::Datarep_extent_function* dtype_file_extent_fn,
                           void* extra_state);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_REGISTER_DATAREP(CHARACTER*(*) DATAREP, EXTERNAL READ_CONVERSION_FN,
                     EXTERNAL WRITE_CONVERSION_FN, EXTERNAL DTYPE_FILE_EXTENT_FN,
                     INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE,
                     INTEGER IERROR)
```

Description

This subroutine associates *read_conversion_fn*, *write_conversion_fn*, and *dtype_file_extent_fn* with the data representation identifier *datarep*. *datarep* can then be used as an argument to MPI_FILE_SET_VIEW, causing subsequent data access operations to call the conversion functions to convert all data items accessed between file data representation and native representation. MPI_REGISTER_DATAREP is a local operation and registers only the data representation for the calling MPI task. If *datarep* is already defined, an error in the MPI_ERR_DUP_DATAREP error class is raised using the default file error handler. The length of a data representation string is limited to the value of MPI_MAX_DATAREP_STRING. MPI_MAX_DATAREP_STRING must have a value of at least 64. No routines are provided to delete data representations and free the associated resources; it is not expected that an application will generate them in significant numbers.

The function *dtype_file_extent_fn* must return, in *file_extent*, the number of bytes required to store *datatype* in the file representation. The function is passed, in *extra_state*, the argument that was passed to MPI_REGISTER_DATAREP. MPI will call this subroutine only with predefined data types employed by the user.

Parameters

datarep
The data representation identifier (string) (IN)

read_conversion_fn
The function invoked to convert from file representation to native representation (function) (IN)

write_conversion_fn
The function invoked to convert from native representation to file representation (function) (IN)

dtype_file_extent_fn
The function invoked to get the extent of a data type in the file representation (function) (IN)

extra_state
The extra state (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

Before specifying your own data representation when setting a view for an opened file, you must first register your data representation using MPI_REGISTER_DATAREP.

IBM PE MPI supports the three predefined data representations: external32, internal, and native.

The C, C++, and Fortran versions of the function prototypes follow:

```
typedef int MPI_Datarep_extent_function(MPI_Datatype datatype,
                                        MPI_Aint *file_extent,
                                        void *extra_state);

typedef MPI::Datarep_extent_function(const MPI::Datatype& datatype,
                                     MPI::Aint& file_extent,
                                     void* extra_state);

SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)
INTEGER DATATYPE, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE

typedef int MPI_Datarep_conversion_function(void *userbuf,
                                            MPI_Datatype datatype,
                                            int count, void *filebuf,
                                            MPI_Offset position,
                                            void *extra_state);

typedef MPI::Datarep_conversion_function(void* userbuf,
                                         MPI::Datatype& datatype,
                                         int count, void* filebuf,
                                         MPI::Offset position,
                                         void* extra_state);

SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF,
                                       POSITION, EXTRA_STATE, IERROR)
<TYPE> USERBUF(*), FILEBUF(*)
INTEGER COUNT, DATATYPE, IERROR
INTEGER(KIND=MPI_OFFSET_KIND) POSITION
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

Errors

Fatal errors:

MPI not initialized

MPI already finalized

Data representation already defined

Data representation identifier too long

Related information

- MPI_FILE_GET_TYPE_EXTENT
- MPI_FILE_GET_VIEW
- MPI_FILE_SET_VIEW

MPI_Request_c2f

Translates a C request handle into a Fortran handle to the same request.

C synopsis

```
#include <mpi.h>
MPI_Fint MPI_Request_c2f(MPI_Request request);
```

Description

This function does not have C++ or Fortran bindings. MPI_Request_c2f translates a C request handle into a Fortran handle to the same request; it maps a null handle into a null handle and a non-valid handle into a non-valid handle. The converted handle is returned as the function's value. There is no error detection or return code.

Parameters

request
The request (handle) (IN)

Related information

- MPI_Request_f2c

MPI_Request_f2c

Returns a C handle to a request.

C synopsis

```
#include <mpi.h>
MPI_Request MPI_Request_f2c(MPI_Fint request);
```

Description

This function does not have C++ or Fortran bindings. MPI_Request_f2c returns a C handle to a request. If *request* is a valid Fortran handle to a request, MPI_Request_f2c returns a valid C handle to that same request. If *request* is set to the Fortran value MPI_REQUEST_NULL, MPI_Request_f2c returns the equivalent null C handle. If *request* is not a valid Fortran handle, MPI_Request_f2c returns a non-valid C handle. The converted handle is returned as the function's value. There is no error detection or return code.

Parameters

request
The request (handle) (IN)

Related information

- MPI_Request_c2f

MPI_REQUEST_FREE, MPI_Request_free

Marks a request for deallocation.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::Request::Free();
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_REQUEST_FREE(INTEGER REQUEST,INTEGER IERROR)
```

Description

This subroutine 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.

Parameters

request
A communication request (handle) (INOUT)

IERROR
The Fortran return code. It is always the last argument.

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

A Grequest free function returned an error

MPI not initialized

MPI already finalized

Related information

- MPI_WAIT

MPI_REQUEST_GET_STATUS, MPI_Request_get_status

Accesses the information associated with a request, without freeing the request.

C synopsis

```
#include <mpi.h>
int MPI_Request_get_status(MPI_Request request, int *flag, MPI_Status *status);
```

C++ synopsis

```
#include mpi.h
bool MPI::Request::Get_status() const;
```

```
#include mpi.h
bool MPI::Request::Get_status(MPI::Status&status) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_REQUEST_GET_STATUS(INTEGER REQUEST, LOGICAL FLAG,
    INTEGER STATUS, INTEGER IERROR)
```

Description

This subroutine accesses the information associated with a request, without freeing the request (in case the user is expected to access it later). It lets you layer libraries more conveniently, because multiple layers of software can access the same completed request and extract from it the status information.

MPI_REQUEST_GET_STATUS sets *flag* = **true** if the operation would complete by MPI_TEST, and, if so, returns in *status* the request status. However, unlike test or wait, it does not deallocate or inactivate the request; a subsequent call to MPI_FREE, MPI_TEST, or MPI_WAIT must still be called on the request to complete it properly. It sets *flag* = **false** if the operation is not complete.

If MPI_REQUEST_GET_STATUS is called with an MPI_REQUEST_NULL or with an inactive request, it will return *flag* = **true** and an empty status.

Parameters

request
The request (handle) (IN)

flag
A boolean flag, same as from MPI_TEST (logical) (OUT)

status
An MPI_STATUS object, if *flag* is **true** (Status) (INOUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

It is valid to call this subroutine with MPI_STATUS_IGNORE if only the *flag* value is needed.

Passing MPI_STATUS_IGNORE for the *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

Errors

Invalid status ignore value

GRequest query function returned an error

Fatal errors:

Invalid request

MPI already finalized

Related information

- MPI_TEST

MPI_RSEND, MPI_Rsend

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Comm::Rsend(const void* buf, int count, const MPI::Datatype& datatype,
        int dest, int tag) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_RSEND(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,
   INTEGER TAG,INTEGER COMM,INTEGER IERROR)
```

Description

This subroutine is a blocking ready mode send operation. 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.

Parameters

buf
The initial address of the send buffer (choice) (IN)

count
The number of elements in the send buffer (integer) (IN)

datatype
The data type of each send buffer element (handle) (IN)

dest
The rank of destination (integer) (IN)

tag
The message tag (positive integer) (IN)

comm
The communicator (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

A ready send for which no receive exists produces a fatal asynchronous error. 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

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);
```

C++ synopsis

```
#include mpi.h
MPI::Prequest MPI::Comm::Rsend_init(const void* buf, int count,
        const MPI::Datatype& datatype,
        int dest, int tag) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_RSEND_INIT(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,
        INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

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.

Parameters

buf
The initial address of the send buffer (choice) (IN)

count
The number of elements to be sent (integer) (IN)

datatype
The type of each element (handle) (IN)

dest
The rank of the destination task (integer) (IN)

tag
The message tag (positive integer) (IN)

comm
The communicator (handle) (IN)

request
The communication request (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

See "MPI_RSEND, MPI_Rsend" on page 443 for more 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_IRSEND
- MPI_START

MPI_SCAN, MPI_Scan

Performs a parallel prefix reduction operation 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);
```

C++ synopsis

```
#include mpi.h
void MPI::Intracomm::Scan(const void *sendbuf, void *recvbuf, int count,
                          const MPI::Datatype& datatype, const MPI::Op& op) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_SCAN(CHOICE SENDBUF,CHOICE RECVBUF,INTEGER COUNT,INTEGER DATATYPE,
  INTEGER OP,INTEGER COMM,INTEGER IERROR)
```

Description

Use this subroutine to perform a prefix reduction operation on data distributed across a 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 to *i* inclusive. The type of operations supported, their semantics, and the restrictions on send and receive buffers are the same as for MPI_REDUCE.

The *in place* option for intra-communicators is specified by passing MPI_IN_PLACE in the *sendbuf* argument. In this case, the input data is taken from the receive buffer, and replaced by the output data.

MPI_SCAN is not supported for inter-communicators.

The parameter *op* may be a predefined reduction operation or a user-defined function, created using MPI_OP_CREATE. This is a list of predefined reduction operations:

Operation
Definition

MPI_BAND
Bitwise AND

MPI_BOR
Bitwise OR

MPI_BXOR
Bitwise XOR

MPI_LAND
Logical AND

MPI_LOR
Logical OR

MPI_LXOR
Logical XOR

MPI_MAX
Maximum value

MPI_MAXLOC
Maximum value and location

MPI_MIN
Minimum value

MPI_MINLOC
Minimum value and location

MPI_PROD
Product

MPI_SUM
Sum

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator occur in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

Parameters

sendbuf
The starting address of the send buffer (choice) (IN)

recvbuf
The starting address of the receive buffer (choice) (OUT)

count
The number of elements in **sendbuf** (integer) (IN)

datatype
The data type of elements in **sendbuf** (handle) (IN)

op The reduction operation (handle) (IN)

comm
The communicator (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Errors

Fatal errors:

Invalid count
count < **0**

Invalid datatype

Type not committed

Invalid op

Invalid communicator

Unequal message lengths

Invalid use of MPI_IN_PLACE

MPI not initialized

MPI already finalized

Develop mode error if:

Inconsistent op

Inconsistent datatype

Inconsistent message length

Related information

- MPI_EXSCAN
- MPE_ISCAN
- MPI_OP_CREATE
- MPI_REDUCE

MPI_SCATTER, MPI_Scatter

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Comm::Scatter(const void* sendbuf, int sendcount,
   const MPI::Datatype& sendtype,
   void* recvbuf, int recvcount,
   const MPI::Datatype& recvtype, int root) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_SCATTER(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,CHOICE RECVBUF,
  INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER ROOT,INTEGER COMM,
  INTEGER IERROR)
```

Description

MPI_SCATTER distributes individual messages from *root* to each task in *comm*. This subroutine 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, pair-wise 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.

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

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

MPI_IN_PLACE is not supported for inter-communicators.

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator occur in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

Parameters

sendbuf
The address of the send buffer (choice, significant only at *root*) (IN)

sendcount
The number of elements to be sent to each task (integer, significant only at *root*) (IN)

sendtype
The data type of the send buffer elements (handle, significant only at *root*) (IN)

recvbuf
The address of the receive buffer (choice) (OUT)

recvcount
The number of elements in the receive buffer (integer) (IN)

recvtype
The data type of the receive buffer elements (handle) (IN)

root
The rank of the sending task (integer) (IN)

comm
The communicator (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

In the 64-bit library, this function uses a shared memory optimization among the tasks on a node. This optimization is discussed in the chapter *Using shared memory* of *IBM Parallel Environment Runtime Edition: MPI Programming Guide*, and is enabled by default. This optimization is not available to 32-bit programs.

Errors

Fatal errors:

Invalid communicator

Invalid counts
count < **0**

Invalid datatypes

Type not committed

Invalid root

For an intra-communicator: *root* < **0** or *root* >= *groupsize*

For an inter-communicator: *root* < **0** and is neither MPI_ROOT nor MPI_PROC_NULL, or *root* >= *groupsize* of the remote group

Unequal message lengths

Invalid use of MPI_IN_PLACE

MPI not initialized

MPI already finalized

Develop mode error if:

Inconsistent root

Inconsistent message length

Related information

- MPE_ISCATTER
- MPI_GATHER
- MPI_SCATTER

MPI_SCATTERV, MPI_Scatterv

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Comm::Scatterv(const void* sendbuf, const int sendcounts[],
    const int displs[], const MPI::Datatype& sendtype,
    void* recvbuf, int recvcount, const MPI::Datatype& recvtype,
    int root) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_SCATTERV(CHOICE SENDBUF,INTEGER SENDCOUNTS(*),INTEGER DISPLS(*),
  INTEGER SENDTYPE,CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,
  INTEGER ROOT,INTEGER COMM,INTEGER IERROR)
```

Description

This subroutine 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, pair-wise 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.

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

If *comm* is an inter-communicator, the call involves all tasks in the inter-communicator, but with one group (group A) defining the root task. All tasks

in the other group (group B) pass the same value in *root*, which is the rank of the root in group A. The root passes the value MPI_ROOT in *root*. All other tasks in group A pass the value MPI_PROC_NULL in *root*. Data is scattered from the root to all tasks in group B. The receive buffer arguments of the tasks in group B must be consistent with the send buffer argument of the root.

MPI_IN_PLACE is not supported for inter-communicators.

When you use this subroutine in a threads application, make sure all collective operations on a particular communicator occur in the same order at each task. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

Parameters

sendbuf
The address of the send buffer (choice, significant only at *root*) (IN)

sendcounts
An integer array (of length *groupsize*) that contains the number of elements to send to each task (significant only at *root*) (IN)

displs
An integer array (of length *groupsize*). Entry *i* specifies the displacement relative to *sendbuf* from which to send the outgoing data to task *i* (significant only at *root*) (IN)

sendtype
The data type of the send buffer elements (handle, significant only at *root*) (IN)

recvbuf
The address of the receive buffer (choice) (OUT)

recvcount
The number of elements in the receive buffer (integer) (IN)

recvtype
The data type of the receive buffer elements (handle) (IN)

root
The rank of the sending task (integer) (IN)

comm
The communicator (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

In the 64-bit library, this function uses a shared memory optimization among the tasks on a node. This optimization is discussed in the chapter *Using shared memory* of *IBM Parallel Environment Runtime Edition: MPI Programming Guide*, and is enabled by default. This optimization is not available to 32-bit programs.

Errors

Fatal errors:

Invalid communicator

Invalid counts

count < **0**

Invalid datatypes

Type not committed

Invalid root

For an intra-communicator: *root* < **0** or *root* >= *groupsize*

For an inter-communicator: *root* < **0** and is neither MPI_ROOT nor MPI_PROC_NULL, or *root* >= *groupsize* of the remote group

Unequal message lengths

Invalid use of MPI_IN_PLACE

MPI not initialized

MPI already finalized

Develop mode error if:

Inconsistent root

Related information

- MPI_GATHER
- MPI_SCATTER

MPI_SEND, MPI_Send

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Comm::Send(const void* buf, int count, const MPI::Datatype& datatype,
       int dest, int tag) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_SEND(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,
      INTEGER TAG,INTEGER COMM,INTEGER IERROR)
```

Description

This subroutine is a blocking standard mode send operation. MPI_SEND causes *count* elements of type *datatype* to be sent from *buf* to the task specified by *dest*. *dest* is a task rank that can be any value from 0 to (*n–1*), where *n* is the number of tasks in *comm*.

Parameters

buf
The initial address of the send buffer (choice) (IN)

count
The number of elements in the send buffer (non-negative integer) (IN)

datatype
The data type of each send buffer element (handle) (IN)

dest
The rank of the destination task in *comm*(integer) (IN)

tag
The message tag (positive integer) (IN)

comm
The communicator (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

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_BSEND
- MPI_ISEND
- MPI_RSEND
- MPI_SENDRECV
- MPI_SSEND

MPI_SEND_INIT, MPI_Send_init

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);
```

C++ synopsis

```
#include mpi.h
MPI::Prequest MPI::Comm::Send_init(const void* buf, int count,
       const MPI::Datatype& datatype,
       int dest, int tag) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_SEND_INIT(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,
      INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Description

This subroutine creates a persistent communication request for a standard mode send operation, and binds to it all arguments of a send operation. A communication started with MPI_SEND_INIT is completed by a call to one of the MPI wait or test operations. MPI_START or MPI_STARTALL is used to activate the send.

Parameters

buf
The initial address of the send buffer (choice) (IN)

count
The number of elements to be sent (integer) (IN)

datatype
The type of each element (handle) (IN)

dest
The rank of the destination task (integer) (IN)

tag
The message tag (positive integer) (IN)

comm
The communicator (handle) (IN)

request
The communication request (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

See “MPI_SEND, MPI_Send” on page 456 for more 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_ISEND
- MPI_START

MPI_SENDRECV, MPI_Sendrecv

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Comm::Sendrecv(const void* sendbuf, int sendcount,
    const MPI::Datatype& sendtype, int dest,
    int sendtag, void* recvbuf, int recvcount,
    const MPI::Datatype& recvtype, int source,
    int recvtag) const;
```

```
#include mpi.h
void MPI::Comm::Sendrecv(const void* sendbuf, int sendcount,
    const MPI::Datatype& sendtype, int dest,
    int sendtag, void* recvbuf, int recvcount,
    const MPI::Datatype& recvtype, int source,
    int recvtag, MPI::Status& status)
    const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
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)
```

Description

This subroutine 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 data types.

Passing MPI_STATUS_IGNORE for the *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status,* you can avoid having to allocate a status object in programs that do not need to examine the status fields.

Parameters

sendbuf
The initial address of the send buffer (choice) (IN)

sendcount
The number of elements to be sent (integer) (IN)

sendtype
The type of elements in the send buffer (handle) (IN)

dest
The rank of the destination task (integer) (IN)

sendtag
The send tag (integer) (IN)

recvbuf
The initial address of the receive buffer (choice) (OUT)

recvcount
The number of elements to be received (integer) (IN)

recvtype
The type of elements in the receive buffer (handle) (IN)

source
The rank of the source task or MPI_ANY_SOURCE (integer) (IN)

recvtag
The receive tag or MPI_ANY_TAG (integer) (IN)

comm
The communicator (handle) (IN)

status
The status object (Status) (INOUT). Note that in Fortran a single status object is an array of integers.

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid counts
count < **0**

Invalid datatypes

Type not committed

Invalid destination
dest < **0** or *dest* > = *groupsize*

Invalid source
source < 0 or *source* > = *groupsize*

Invalid communicator

Invalid tags
tag < 0

Invalid status ignore value

MPI not initialized

MPI already finalized

Related information

- MPI_RECV
- MPI_SEND
- MPI_SENDRECV_REPLACE

MPI_SENDRECV_REPLACE, MPI_Sendrecv_replace

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Comm::Sendrecv_replace(void* buf, int count,
     const MPI::Datatype& datatype,
     int dest, int sendtag, int source,
     int recvtag) const;
```

```
#include mpi.h
void MPI::Comm::Sendrecv_replace(void *buf, int count,
     const MPI::Datatype& datatype,
     int dest, int sendtag, int source,
     int recvtag, MPI::Status& status)
     const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
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)
```

Description

This subroutine 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.

Passing MPI_STATUS_IGNORE for the *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status,* you can avoid having to allocate a status object in programs that do not need to examine the status fields.

Parameters

buf
The initial address of the send and receive buffer (choice) (INOUT)

count
The number of elements to be sent and received (integer) (IN)

datatype
The type of elements in the send and receive buffer (handle) (IN)

dest
The rank of the destination task (integer) (IN)

sendtag
The send message tag (integer) (IN)

source
The rank of the source task or MPI_ANY_SOURCE (integer) (IN)

recvtag
The receive message tag or MPI_ANY_TAGE (integer) (IN)

comm
The communicator (handle) (IN)

status
The status object (Status) (INOUT). Note that in Fortran a single status object is an array of integers.

IERROR
The Fortran return code. It is always the last argument.

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 tags
tag < 0

Invalid status ignore value

Out of memory

MPI not initialized

MPI already finalized

Related information

- MPI_SENDRECV

MPI_SIZEOF

Returns the size in bytes of the machine representation of the given variable.

Fortran synopsis

```
USE MPI
MPI_SIZEOF(CHOICE X, INTEGER SIZE, INTEGER IERROR)
```

Description

This subroutine returns the size in bytes of the machine representation of the given variable. It is a generic Fortran routine and has only a Fortran binding . It requires information provided by the MPI module and will produce a runtime error if the program was coded with **include 'mpif.h'**. MPI_SIZEOF is most useful when variables are declared with KIND=SELECTED_*xxx*_KIND because the number of bytes for a variable may vary from one architecture to another.

Parameters

X A Fortran variable of numeric intrinsic type (choice) (IN)

SIZE
The size of the machine representation of that type (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_SIZEOF is similar to the C and C++ **sizeof** operator, but behaves slightly differently. If MPI_SIZEOF is given an array argument, it returns the size of the base element, not the size of the whole array.

Errors

Fatal errors:

MPI already finalized

MPI not initialized

No "USE MPI" statement in compilation unit

MPI_SSEND, MPI_Ssend

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Comm::Ssend(const void* buf, int count, const MPI::Datatype& datatype,
        int dest, int tag) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_SSEND(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,
   INTEGER TAG,INTEGER COMM,INTEGER IERROR)
```

Description

This subroutine is a blocking synchronous mode send operation. This is a nonlocal 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 processing the matching receive. If both sends and receives are blocking operations, the synchronous mode provides synchronous communication.

Parameters

buf
The initial address of the send buffer (choice) (IN)

count
The number of elements in the send buffer (integer) (IN)

datatype
The data type of each send buffer element (handle) (IN)

dest
The rank of the destination task (integer) (IN)

tag
The message tag (positive integer) (IN)

comm
The communicator (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

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

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);
```

C++ synopsis

```
#include mpi.h
MPI::Prequest MPI::Comm::Ssend_init(const void* buf, int count,
        const MPI::Datatype& datatype,
        int dest, int tag) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_SSEND_INIT(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,
        INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
```

Description

This subroutine creates a persistent communication object for a synchronous mode send operation. MPI_START or MPI_STARTALL can be used to activate the send.

Parameters

buf
The initial address of the send buffer (choice) (IN)

count
The number of elements to be sent (integer) (IN)

datatype
The type of each element (handle) (IN)

dest
The rank of the destination task (integer) (IN)

tag
The message tag (positive integer) (IN)

comm
The communicator (handle) (IN)

request
The communication request (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

See "MPI_SSEND, MPI_Ssend" on page 465 for more 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_ISSEND
- MPI_START

MPI_START, MPI_Start

Activates a persistent request operation.

C synopsis

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

C++ synopsis

```
#include mpi.h
void MPI::Prequest::Start();
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_START(INTEGER REQUEST,INTEGER IERROR)
```

Description

MPI_START activates a persistent request operation. A communication started with MPI_START is completed by a call to one of the MPI wait or test operations. **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.

Parameters

request
A communication request (handle) (INOUT)

IERROR
The Fortran return code. It is always the last argument.

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_BSEND_INIT
- MPI_RECV_INIT
- MPI_RSEND_INIT
- MPI_SEND_INIT
- MPI_SSEND_INIT
- MPI_STARTALL

MPI_STARTALL, MPI_Startall

Activates a collection of persistent request operations.

C synopsis

```
#include <mpi.h>
int MPI_Startall(int count,MPI_request *array_of_requests);
```

C++ synopsis

```
#include mpi.h
void MPI::Prequest::Startall(int count, MPI::Prequest array_of_requests[]);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_STARTALL(INTEGER COUNT,INTEGER ARRAY_OF_REQUESTS(*),INTEGER IERROR)
```

Description

MPI_STARTALL starts all communication 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, a matching receive must be posted before the call. If a request is for a buffered send, adequate buffer space must be available.

Parameters

count
The list length (integer) (IN)

array_of_requests
The array of requests (array of handle) (INOUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid count

Invalid request array

Request invalid

Request not persistent

Request active

Insufficient buffer space
Only if a buffered send

MPI not initialized

MPI already finalized

Related information

- MPI_START

MPI_Status_c2f

Translates a C status object into a Fortran status object.

C synopsis

```
#include <mpi.h>
int MPI_Status_c2f(MPI_Status *c_status, MPI_Fint *f_status);
```

Description

This function converts a C status object (which is a user-declared structure object) to a Fortran status object (which is a user-declared array of integers). The conversion occurs on all the information in status, including that which is hidden. That is, no status information is lost in the conversion.

The value of *c_status* must not be either MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE. Code that calls MPI_Status_c2f is responsible for checking that neither ignore value is used.

There is not a separate conversion function for arrays of statuses, because you can simply loop through the array, converting each status.

Parameters

c_status
The C status object (IN)

f_status
The Fortran status object (OUT)

Related information

- MPI_Status_f2c

MPI_Status_f2c

Converts a Fortran status object into a C status object.

C synopsis

```
#include <mpi.h>
int MPI_Status_f2c(MPI_Fint *f_status, MPI_Status *c_status);
```

Description

This function converts a Fortran status object (which is a user-declared array of integers) to a C status object (which is a user-declared structure object). The conversion occurs on all the information in status, including that which is hidden. That is, no status information is lost in the conversion.

If *f_status* is a valid Fortran status, but not the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE, MPI_Status_f2c returns in *c_status* a valid C status with the same content. If *f_status* is the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE, or if *f_status* is not a valid Fortran status, the call is erroneous.

The predeclared global variables MPI_F_STATUS_IGNORE and MPI_F_STATUSES_IGNORE can be used to test whether *f_status* is one of the ignore values.

The C status has the same source, tag and error code values as the Fortran status, and returns the same answers when queried for count, elements, and cancellation. The conversion function can be called with a Fortran status argument that has an undefined error field, in which case the value of the error field in the C status argument is undefined.

There is not a separate conversion function for arrays of statuses, because you can simply loop through the array, converting each status.

Parameters

f_status
The Fortran status object (IN)

c_status
The C status object (OUT)

Related information

- MPI_Status_c2f

MPI_STATUS_SET_CANCELLED, MPI_Status_set_cancelled

Defines cancellation information for a request.

C synopsis

```
#include <mpi.h>
int MPI_Status_set_cancelled(MPI_Status *status, int flag);
```

C++ synopsis

```
#include mpi.h
void MPI::Status::Set_cancelled(bool flag);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_STATUS_SET_CANCELLED(INTEGER STATUS(MPI_STATUS_SIZE),
    LOGICAL FLAG, INTEGER IERROR)
```

Description

This subroutine defines cancellation information for a generalized request and places it in a status object. If *flag* is set to **true**, a subsequent call to MPI_TEST_CANCELLED will also return *flag* = **true**; otherwise it will return **false**.

Users are advised not to reuse the status fields for values other than those for which they were intended. Doing so may lead to unexpected results when using the status object. For example, calling MPI_GET_ELEMENTS may cause an error if the value is out of range or it may be impossible to detect such an error. The *extra_state* argument provided with a generalized request can be used to return information that does not logically belong in *status*. Furthermore, modifying the values in a status set internally by MPI (that is, MPI_RECV), may lead to unpredictable results and is strongly discouraged.

Parameters

status
The status object to associate the cancel flag with (Status) (INOUT)

flag
The flag (Status) (logical) (IN). **true** indicates that the request was cancelled.

IERROR
The Fortran return code. It is always the last argument.

Errors

MPI not initialized

MPI already finalized

Related information

- MPI_STATUS_SET_ELEMENTS

MPI_STATUS_SET_ELEMENTS, MPI_Status_set_elements

Defines element information for a request.

C synopsis

```
#include <mpi.h>
int MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype, int count);
```

C++ synopsis

```
#include mpi.h
void MPI::Status::Set_elements(const MPI::Datatype& datatype, int count);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_STATUS_SET_ELEMENTS(INTEGER STATUS(MPI_STATUS_SIZE), INTEGER DATATYPE,
    INTEGER COUNT, INTEGER IERROR)
```

Description

This subroutine defines element information for a generalized request and places it in a status object.

Users are advised not to reuse the status fields for values other than those for which they were intended. Doing so may lead to unexpected results when using the status object. For example, calling MPI_GET_ELEMENTS may cause an error if the value is out of range or it may be impossible to detect such an error. The *extra_state* argument provided with a generalized request can be used to return information that does not logically belong in *status*. Furthermore, modifying the values in a status set internally by MPI (that is, MPI_RECV), may lead to unpredictable results and is strongly discouraged.

Parameters

status
The status object to associate **count** with (Status) (INOUT)

datatype
The data type associated with **count** (handle) (IN)

count
The number of elements to associate with *status* (integer) (IN)

IERROR
The Fortran return code. It is always the last argument.

Errors

MPI not initialized

MPI already finalized

Related information

- MPI_STATUS_SET_CANCELLED

MPI_TEST, MPI_Test

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);
```

C++ synopsis

```
#include mpi.h
bool MPI::Request::Test();
```

```
#include mpi.h
bool MPI::Request::Test(MPI::Status& status);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TEST(INTEGER REQUEST,LOGICAL FLAG,INTEGER STATUS(MPI_STATUS_SIZE),
    INTEGER IERROR)
```

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, MPI_Wait" on page 559.)

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 only by the return code.

Passing MPI_STATUS_IGNORE for the *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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 or test call does not contain meaningful source, tag, or message size information.

When you use this subroutine in a threads 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 *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

Parameters

request
The operation request (handle) (INOUT)

flag
Set to **true** if the operation completed (logical) (OUT)

status
The status object (Status) (INOUT). Note that in Fortran a single status object is an array of integers.

IERROR
The Fortran return code. It is always the last argument.

Errors

A GRequest free function returned an error

A GRequest query function returned an error

Invalid status ignore value

Invalid form of status ignore

Invalid request handle

Truncation occurred

MPI not initialized

MPI already finalized

Develop mode error if:

Illegal buffer update (ISEND)

Inconsistent datatype (MPE_I collectives)

Inconsistent message length (MPE_I collectives)

Inconsistent op (MPE_I collectives)

Match of blocking and non-blocking collectives (MPE_I collectives)

Related information

- MPI_TESTALL
- MPI_TESTANY
- MPI_TESTSOME
- MPI_WAIT

MPI_TEST_CANCELLED, MPI_Test_cancelled

Tests whether a nonblocking operation was cancelled.

C synopsis

```
#include <mpi.h>
int MPI_Test_cancelled(MPI_Status * status,int *flag);
```

C++ synopsis

```
#include mpi.h
bool MPI::Status::Is_cancelled() const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TEST_CANCELLED(INTEGER STATUS(MPI_STATUS_SIZE),LOGICAL FLAG,
    INTEGER IERROR)
```

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.

Parameters

status
A status object (Status) (IN). Note that in Fortran a single status object is an array of integers.

flag
Set to **true** if the operation was cancelled (logical) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

Nonblocking I/O operations are never cancelled successfully.

Errors

MPI not initialized

MPI already finalized

Related information

- MPI_CANCEL

MPI_TESTALL, MPI_Testall

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);
```

C++ synopsis

```
#include mpi.h
bool MPI::Request::Testall(int count, MPI::Request req_array[]);
```

```
#include mpi.h
bool MPI::Request::Testall(int count, MPI::Request req_array[],
      MPI::Status stat_array[]);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TESTALL(INTEGER COUNT,INTEGER ARRAY_OF_REQUESTS(*),LOGICAL FLAG,
      INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*),INTEGER IERROR)
```

Description

This subroutine 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.

Passing MPI_STATUSES_IGNORE for the *array_of_statuses* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *array_of_statuses*, you can avoid having to allocate a status object array in programs that do not need to examine the status fields.

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 or test call does not contain meaningful source, tag, or message size information.

When you use this subroutine in a threads 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 *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

Parameters

count
The number of requests to test (integer) (IN)

array_of_requests
An array of requests of length *count* (array of handles) (INOUT)

flag
(logical) (OUT)

array_of_statuses
An array of status of length *count* objects (array of status) (INOUT). Note that in Fortran a status object is itself an array.

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid count
count < **0**

Invalid request array

Invalid requests

Truncation occurred

A GRequest free function returned an error

A GRequest query function returned an error

Invalid status ignore value

Invalid form of status ignore

MPI not initialized

MPI already finalized

Develop mode error if:

Illegal buffer update (ISEND)

Inconsistent datatype (MPE_I collectives)

Inconsistent message length (MPE_I collectives)

Inconsistent op (MPE_I collectives)

Match of blocking and non-blocking collectives (MPE_I collectives)

Related information

- MPI_TEST
- MPI_WAITALL

MPI_TESTANY, MPI_Testany

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);
```

C++ synopsis

```
#include mpi.h
bool MPI::Request::Testany(int count, MPI::Request array[],
      int& index);
```

```
#include mpi.h
bool MPI::Request::Testany(int count, MPI::Request array[],
      int& index, MPI::Status& status);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TESTANY(INTEGER COUNT,INTEGER ARRAY_OF_REQUESTS(*),INTEGER INDEX,LOGICAL FLAG,
     INTEGER STATUS(MPI_STATUS_SIZE), INTEGER IERROR)
```

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, 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 invocation 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.

Passing MPI_STATUS_IGNORE for the *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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 or test call does not contain meaningful source, tag, or message size information.

When you use this subroutine in a threads 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 *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

Parameters

count
The list length (integer) (IN)

array_of_requests
The array of request (array of handles) (INOUT)

index
The index of the operation that completed, or MPI_UNDEFINED is no operation completed (OUT)

flag
Set to **true** if one of the operations is complete (logical) (OUT)

status
The status object (Status) (INOUT). Note that in Fortran a single status object is an array of integers.

IERROR
The Fortran return code. It is always the last argument.

Notes

The array is indexed from zero (0) in C and from one (1) in Fortran.

The use of this routine makes the order in which your application completes the requests nondeterministic. An application that processes messages in whatever order they complete must not make assumptions about that order. For example, if:

```
((msgA op msgB) op msgC)
```

can give a different answer than:

```
((msgB op msgC) op msgA)
```

the application must be prepared to accept either answer as *correct*, and must **not** assume a second run of the application will give the same answer.

Errors

Invalid count
count < **0**

Invalid request array

Invalid requests

Truncation occurred

A GRequest free function returned an error

A GRequest query function returned an error

Invalid status ignore value

Invalid form of status ignore

MPI not initialized

MPI already finalized

Develop mode error if:

Illegal buffer update (ISEND)

Inconsistent datatype (MPE_I collectives)

Inconsistent message length (MPE_I collectives)

Inconsistent op (MPE_I collectives)

Match of blocking and non-blocking collectives (MPE_I collectives)

Related information

- MPI_TEST
- MPI_WAITANY

MPI_TESTSOME, MPI_Testsome

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);
```

C++ synopsis

```
#include mpi.h
int MPI::Request::Testsome(int incount, MPI::Request req_array[],
      int array_of_indices[]);
```

```
#include mpi.h
int MPI::Request::Testsome(int incount, MPI::Request req_array[],
      int array_of_indices[],
      MPI::Status stat_array[]);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TESTSOME(INTEGER INCOUNT,INTEGER ARRAY_OF_REQUESTS(*),
      INTEGER OUTCOUNT,INTEGER ARRAY_OF_INDICES(*),
             INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*),INTEGER IERROR)
```

Description

This subroutine 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.

Passing MPI_STATUSES_IGNORE for the *array_of_statuses* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *array_of_statuses*, you can avoid having to allocate a status object array in programs that do not need to examine the status fields.

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 or test call does not contain meaningful source, tag, or message size information.

When you use this subroutine in a threads 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 *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

Parameters

incount
The length of *array_of_requests* (integer) (IN)

array_of_requests
The array of requests (array of handles) (INOUT)

outcount
The number of completed requests (integer) (OUT)

array_of_indices
The array of indices of operations that completed (array of integers) (OUT)

array_of_statuses
The array of status objects for operations that completed (array of status) (INOUT). Note that in Fortran a status object is itself an array.

IERROR
The Fortran return code. It is always the last argument.

Notes

The use of this routine makes the order in which your application completes the requests nondeterministic. An application that processes messages in whatever order they complete must not make assumptions about that order. For example, if:

```
((msgA op msgB) op msgC)
```

can give a different answer than:

```
((msgB op msgC) op msgA)
```

the application must be prepared to accept either answer as *correct*, and must **not** assume a second run of the application will give the same answer.

Errors

Invalid count
count < **0**

Invalid request array

Invalid request

Truncation occurred

A GRequest free function returned an error

A GRequest query function returned an error

Invalid status ignore value

Invalid form of status ignore

MPI not initialized

MPI already finalized

Develop mode error if:

Illegal buffer update (ISEND)

Inconsistent datatype (MPE_I collectives)

Inconsistent message length (MPE_I collectives)

Inconsistent op (MPE_I collectives)

Match of blocking and non-blocking collectives (MPE_I collectives)

Related information

- MPI_TEST
- MPI_WAITSOME

MPI_TOPO_TEST, MPI_Topo_test

Returns the type of virtual topology associated with a communicator.

C synopsis

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

C++ synopsis

```
#include mpi.h
int MPI::Comm::Get_topology() const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TOPO_TEST(INTEGER COMM,INTEGER STATUS,INTEGER IERROR)
```

Description

This subroutine 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_DIST_GRAPH
distributed graph topology

MPI_UNDEFINED
no topology

Parameters

comm
The communicator (handle) (IN)

status
The topology type of communicator *comm* (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

MPI not initialized

MPI already finalized

Invalid communicator

Related information

- MPI_CART_CREATE
- MPI_GRAPH_CREATE

MPI_Type_c2f

Translates a C data type handle into a Fortran handle to the same data type.

C synopsis

```
#include <mpi.h>
MPI_Fint MPI_Type_c2f(MPI_Type datatype);
```

Description

This function does not have C++ or Fortran bindings. MPI_Type_c2f translates a C data type handle into a Fortran handle to the same data type; it maps a null handle into a null handle and a non-valid handle into a non-valid handle. The converted handle is returned as the function's value. There is no error detection or return code.

Parameters

datatype
The data type (handle) (IN)

Related information

- MPI_Type_f2c

MPI_TYPE_COMMIT, MPI_Type_commit

Makes a data type ready for use in communication.

C synopsis

```
#include <mpi.h>
int MPI_Type_commit(MPI_Datatype *datatype);
```

C++ synopsis

```
#include mpi.h
void MPI::Datatype::Commit();
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_COMMIT(INTEGER DATATYPE,INTEGER IERROR)
```

Description

A data type object must be committed before you can use it in communication. You can use an uncommitted data type as an argument in data type constructors.

This subroutine makes a data type ready for use in communication. The data type is the formal description of a communication buffer. It is not the content of the buffer.

Once the data type is committed it can be repeatedly reused to communicate the changing contents of a buffer or buffers with different starting addresses.

Parameters

datatype
The data type that is to be committed (handle) (INOUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

Basic data types 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

Returns a new data type 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);
```

C++ synopsis

```
#include mpi.h
MPI::Datatype MPI::Datatype::Create_contiguous(int count) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_CONTIGUOUS(INTEGER COUNT,INTEGER OLDTYPE,INTEGER NEWTYPE,INTEGER IERROR)
```

Description

This subroutine returns a new data type that represents the concatenation of *count* instances of *oldtype*. MPI_TYPE_CONTIGUOUS allows replication of a data type into contiguous locations.

Parameters

count
The replication *count* (non-negative integer) (IN)

oldtype
The old data type (handle) (IN)

newtype
The new data type (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

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

Generates the data types 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);
```

C++ synopsis

```
#include mpi.h
MPI::Datatype MPI::Datatype::Create_darray(int size, int rank, int ndims,
        const int array_of_gsizes[],
        const int array_of_distribs[],
        const int array_of_dargs[],
        const int array_of_psizes[],
        int order) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
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)
```

Description

MPI_TYPE_CREATE_DARRAY generates the data types 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.

Parameters

size
The size of the task group (positive integer) (IN)

rank
The rank in the task group (nonnegative integer) (IN)

ndims
The number of array dimensions as well as task grid dimensions (positive integer) (IN)

array_of_gsizes
The number of elements of type *oldtype* in each dimension of the global array (array of positive integers) (IN)

array_of_distribs
The distribution of the global array in each dimension (array of state) (IN)

array_of_dargs
The distribution argument in each dimension of the global array (array of positive integers) (IN)

array_of_psizes
The size of the logical grid of tasks in each dimension (array of positive integers) (IN)

order
The array storage order *flag* (state) (IN)

oldtype
The old data type (handle) (IN)

newtype
The new data type (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

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 data type

Invalid datatype
oldtype cannot be: MPI_LB, MPI_PACKED, or MPI_UB

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_F90_COMPLEX, MPI_Type_create_f90_complex

Returns a predefined MPI data type that matches a COMPLEX variable of KIND selected_real_kind(*p, r*).

C synopsis

```
#include <mpi.h>
int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype);
```

C++ synopsis

```
#include mpi.h
static MPI::Datatype MPI::Datatype::Create_f90_complex(int p, int r);
```

Fortran synopsis

```
USE MPI
MPI_TYPE_CREATE_F90_COMPLEX(INTEGER P, INTEGER R, INTEGER NEWTYPE, INTEGER IERROR)
```

Description

This subroutine returns a predefined MPI data type that matches a COMPLEX variable of KIND=selected_real_kind(*p, r*). Either *p* or *r* may be omitted from calls to selected_real_kind(*p, r*), but not both. Analogously, either *p* or *r* may be set to MPI_UNDEFINED in this subroutine. In communication, an MPI data type A returned by MPI_TYPE_CREATE_F90_COMPLEX matches a data type B if and only if B was returned by MPI_TYPE_CREATE_F90_COMPLEX called with the same values for *p* and *r*, or B is a duplicate of such a data type.

Parameters

p The precision in decimal digits (integer) (IN)

r The decimal exponent range (integer) (IN)

newtype
The requested MPI data type (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

It is erroneous to supply values for *p* and *r* that are not supported by the compiler.

An MPI_Datatype returned by this subroutine is already committed. It cannot be freed with MPI_TYPE_FREE. It can be used with the MPI reduction functions.

Errors

Fatal errors:

MPI already finalized

MPI not initialized

p or r value outside range supported by compiler

Related information

- MPI_TYPE_CREATE_F90_INTEGER
- MPI_TYPE_CREATE_F90_REAL

MPI_TYPE_CREATE_F90_INTEGER, MPI_Type_create_f90_integer

Returns a predefined MPI data type that matches an INTEGER variable of KIND selected_integer_kind(*r*).

C synopsis

```
#include <mpi.h>
int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype);
```

C++ synopsis

```
#include mpi.h
static MPI::Datatype MPI::Datatype::Create_f90_integer(int r);
```

Fortran synopsis

```
USE MPI
MPI_TYPE_CREATE_F90_INTEGER(INTEGER R, INTEGER NEWTYPE, INTEGER IERROR)
```

Description

This subroutine returns a predefined MPI data type that matches an INTEGER variable of KIND=selected_integer_kind(*r*). In communication, an MPI data type A returned by MPI_TYPE_CREATE_F90_INTEGER matches a data type B if and only if B was returned by MPI_TYPE_CREATE_F90_INTEGER called with the same value for *r*, or B is a duplicate of such a data type.

Parameters

r The decimal exponent range, that is, the number of decimal digits (integer) (IN)

newtype
The requested MPI data type (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

It is erroneous to supply values for *r* that are not supported by the compiler.

An MPI_Datatype returned by this subroutine is already committed. It cannot be freed with MPI_TYPE_FREE. It can be used with the MPI reduction functions.

Errors

Fatal errors:

MPI already finalized

MPI not initialized

r value outside range supported by compiler

Related information

- MPI_TYPE_CREATE_F90_COMPLEX
- MPI_TYPE_CREATE_F90_REAL

MPI_TYPE_CREATE_F90_REAL, MPI_Type_create_f90_real

Returns a predefined MPI data type that matches a REAL variable of KIND selected_real_kind(*p, r*).

C synopsis

```
#include <mpi.h>
int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype);
```

C++ synopsis

```
#include mpi.h
static MPI::Datatype MPI::Datatype::Create_f90_real(int p, int r);
```

Fortran synopsis

```
USE MPI
MPI_TYPE_CREATE_F90_REAL(INTEGER P, INTEGER R, INTEGER NEWTYPE, INTEGER IERROR)
```

Description

This subroutine returns a predefined MPI data type that matches a REAL variable of KIND=selected_real_kind(*p, r*). In the model described in this manual page, it returns a handle for the element D(*p, r*). Either *p* or *r* may be omitted from calls to selected_real_kind(*p, r*), but not both. Analogously, either *p* or *r* may be set to MPI_UNDEFINED in calling this subroutine. In communication, an MPI data type A returned by MPI_TYPE_CREATE_F90_REAL matches a data type B if and only if B was returned by MPI_TYPE_CREATE_F90_REAL called with the same values for *p* and *r*, or B is a duplicate of such a data type.

Parameters

p The precision in decimal digits (integer) (IN)

r The decimal exponent range (integer) (IN)

newtype
The requested MPI data type (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

It is erroneous to supply values for *p* and *r* that are not supported by the compiler.

An MPI_Datatype returned by this subroutine is already committed. It cannot be freed with MPI_TYPE_FREE. It can be used with the MPI reduction functions.

Errors

Fatal errors:

MPI already finalized

MPI not initialized

p or r value outside range supported by compiler

Related information

- MPI_TYPE_CREATE_F90_COMPLEX
- MPI_TYPE_CREATE_F90_INTEGER

MPI_TYPE_CREATE_HINDEXED, MPI_Type_create_hindexed

Returns a new data type 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_create_hindexed(int count, int array_of_blocklengths[],
        MPI_Aint array_of_displacements[],
        MPI_Datatype oldtype,MPI_Datatype *newtype);
```

C++ synopsis

```
#include mpi.h
MPI::Datatype MPI::Datatype::Create_hindexed(int count,
          const int array_of_blocklengths[],
          const MPI::Aint array_of_displacements[])
          const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_CREATE_HINDEXED(INTEGER COUNT, INTEGER ARRAY_OF_BLOCKLENGTHS(*),
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*),
    INTEGER OLDTYPE, INTEGER NEWTYPE, INTEGER IERROR)
```

Description

This subroutine returns a new data type that represents *count* blocks. Each block 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 (the way they are expressed in MPI_TYPE_INDEXED).

Parameters

count
The number of blocks and the number of entries in *array_of_displacements* and *array_of_blocklengths* (non-negative integer) (IN)

array_of_blocklengths
The number of elements in each block (array of non-negative integers) (IN)

array_of_displacements
A byte displacement for each block (array of integer) (IN)

oldtype
The old data type (handle) (IN)

newtype
The new data type (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

newtype must be committed using MPI_TYPE_COMMIT before being used for communication.

MPI_TYPE_CREATE_HINDEXED is synonymous with MPI_TYPE_HINDEXED in C and C++, or in Fortran when default INTEGERs are address-sized.

(MPI_TYPE_HINDEXED is not available in C++.) In Fortran, MPI_TYPE_CREATE_HINDEXED accepts arguments of type INTEGER(KIND=MPI_ADDRESS_KIND), for *array_of_displacements* where type MPI_Aint is used in C.

If Fortran 64-bit applications must be written to be portable to systems that do not support Fortran 90 KIND declarations, it is also correct to declare the (KIND=MPI_ADDRESS_KIND) arguments as INTEGER*8. The KIND format has the advantage of allowing the same source code to compile for either 32-bit or 64-bit processing. The MPI_TYPE_HINDEXED binding is retained to support old codes but any new code, whether C or Fortran should use MPI_TYPE_CREATE_HINDEXED.

Note that the MPI-1 routines that use a Fortran INTEGER where C bindings specify MPI_Aint will work correctly as long as the values they represent fit in a 32-bit signed integer. It can be difficult to predict reliably when values will remain in range and the loss of high-order bits when overflow does occur will not raise an MPI error, so this may lead to obscure application failures.

Errors

Fatal 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_GET_ADDRESS
- MPI_TYPE_CREATE_HVECTOR
- MPI_TYPE_CREATE_STRUCT

MPI_TYPE_CREATE_HVECTOR, MPI_Type_create_hvector

Returns a new data type 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_create_hvector(int count, int blocklength, MPI_Aint stride,
       MPI_Datatype oldtype, MPI_Datatype *newtype);
```

C++ synopsis

```
#include mpi.h
MPI::Datatype MPI::Datatype::Create_hvector(int count, int blocklength,
        MPI::Aint stride) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_CREATE_HVECTOR(INTEGER COUNT, INTEGER BLOCKLENGTH,
   INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE,
   INTEGER OLDTYPE, INTEGER NEWTYPE, INTEGER IERROR)
```

Description

This subroutine returns a new data type 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.

Parameters

count
The number of blocks (non-negative integer) (IN)

blocklength
The number of elements in each block (non-negative integer) (IN)

stride
An integer specifying the number of bytes between start of each block (IN)

oldtype
The old data type (handle) (IN)

newtype
The new data type (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

newtype must be committed using MPI_TYPE_COMMIT before being used for communication.

MPI_TYPE_CREATE_HVECTOR is synonymous with MPI_TYPE_HVECTOR in C and C++, or in Fortran when default INTEGERs are address-sized. (MPI_TYPE_HVECTOR is not available in C++.) In Fortran, MPI_TYPE_CREATE_HVECTOR accepts an argument of type INTEGER(KIND=MPI_ADDRESS_KIND) for *stride* where type MPI_Aint is used in C.

If Fortran 64-bit applications must be written to be portable to systems that do not support Fortran 90 KIND declarations, it is also correct to declare the (KIND=MPI_ADDRESS_KIND) arguments as INTEGER*8. The KIND format has the advantage of allowing the same source code to compile for either 32-bit or 64-bit processing. The MPI_TYPE_HVECTOR binding is retained to support old codes but any new code, whether C or Fortran should use MPI_TYPE_CREATE_HVECTOR.

Note that the MPI-1 routines that use a Fortran INTEGER where C bindings specify MPI_Aint will work correctly as long as the values they represent fit in a 32-bit signed integer. It can be difficult to predict reliably when values will remain in range and the loss of high-order bits when overflow does occur will not raise an MPI error, so this may lead to obscure application failures.

Errors

Fatal 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_GET_ADDRESS
- MPI_TYPE_CREATE_HINDEXED
- MPI_TYPE_CREATE_STRUCT

MPI_TYPE_CREATE_INDEXED_BLOCK, MPI_Type_create_indexed_block

Returns a new data type that represents *count* blocks.

C synopsis

```
#include <mpi.h>
int MPI_Type_create_indexed_block(int count, int blocklength,
      int array_of_displacements[],
      MPI_Datatype oldtype, MPI_datatype *newtype);
```

C++ synopsis

```
#include mpi.h
MPI::Datatype MPI::Datatype::Create_indexed_block(int count, int blocklength,
        const int array_of_displacements[])
        const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_CREATE_INDEXED_BLOCK(INTEGER COUNT, INTEGER BLOCKLENGTH,
         INTEGER ARRAY_OF DISPLACEMENTS(*), INTEGER OLDTYPE,
         INTEGER NEWTYPE, INTEGER IERROR)
```

Description

This subroutine returns a new data type that represents *count* blocks. Each block is defined by an entry in *array_of_displacements*. Displacements are expressed in units of extent(*oldtype*).

Parameters

count
The length of *array_of_displacements* (non-negative integer) (IN)

blocklength
The size of the block (non-negative integer) (IN). All blocks are the same size.

array_of_displacements
The displacement of each block in units of extent(**oldtype**) (array of integer) (IN)

oldtype
The old data type (handle) (IN)

newtype
The new data type (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

newtype must be committed using MPI_TYPE_COMMIT before being used for communication.

Errors

Fatal errors:

Invalid blocklength
blocklength < **0**

Invalid count
count < **0**

Oldtype is MPI_LB, MPI_UB or MPI_PACKED

MPI already finalized

MPI not initialized

Undefined oldtype

Related information

- MPI_TYPE_COMMIT
- MPI_TYPE_INDEXED

MPI_TYPE_CREATE_KEYVAL, MPI_Type_create_keyval

Creates a new attribute key for a data type.

C synopsis

```
#include <mpi.h>
int MPI_Type_create_keyval (MPI_Type_copy_attr_function *type_copy_attr_fn,
   MPI_Type_delete_attr_function *type_delete_attr_fn,
                        int *type_keyval, void *extra_state);
```

C++ synopsis

```
#include mpi.h
int MPI::Datatype::Create_keyval(MPI::Datatype::Copy_attr_function*,
   type_copy_attr_fn,MPI::Datatype::Delete_attr_function*,
   type_delete_attr_fn, void* extra_state);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_CREATE_KEYVAL(EXTERNAL TYPE_COPY_ATTR_FN, EXTERNAL TYPE_DELETE_ATTR_FN,
   INTEGER TYPE_KEYVAL, INTERGER EXTRA_STATE, INTEGER IERROR)
```

Description

This subroutine creates a new attribute key for a data type and returns a handle to it in the *type_keyval* argument. A key is unique in a task and is opaque to the user. Once created, a key can be used to associate an attribute with a data type and access it within the local task. The copy function *type_copy_attr_fn* is invoked when a data type is duplicated by **MPI_TYPE_DUP**. Attribute copy functions are invoked in arbitrary order for each key value in *oldtype*. If the copy function sets its flag argument to **0**, the attribute is deleted in the new data type. Otherwise, the new attribute is set using the *attribute_val_out* argument of **MPI_Type_copy_attr_function**.

The attribute delete function *type_delete_attr_fn* is called by MPI_TYPE_FREE, MPI_TYPE_DELETE_ATTR, and MPI_TYPE_SET_ATTR. The delete function takes whatever steps are needed by the user code to remove an attribute. The predefined functions MPI_TYPE_NULL_COPY_FN and MPI_TYPE_DUP_FN can be used to never copy or to always copy, respectively. The predefined function MPI_TYPE_NULL_DELETE_FN can be used if no special handling of attribute deletions is required. The attribute copy and delete functions are defined as follows (only the C form is shown here):

```
int MPI_Type_copy_attr_function(MPI_Datatype oldtype, int type_keyval,
                                void *extra_state, void *attribute_val_in,
                                void *attribute_val_out, int *flag)

int MPI_Type_delete_attr_function(MPI_Datatype type, int type_keyval,
                                  void *attribute_val, void *extra_state)
```

The *attribute_val_in* parameter is the value of the attribute. The *attribute_val_out* parameter is the address of the value, so the function can set a new value. The *attribute_val_out* parameter is logically a **void****, but it is prototyped as **void***, to avoid the need for complex casting.

Parameters

type_copy_attr_fn
The copy callback function for *type_keyval* (function) (IN)

type_delete_attr_fn
The delete callback function for *type_keyval* (function) (IN)

type_keyval
The key value for future access (integer) (OUT)

extra_state
The extra state for callback functions (IN)

IERROR
The Fortran return code. It is always the last argument.

Errors

Fatal errors:

MPI not initialized (MPI_ERR_OTHER)

MPI already finalized (MPI_ERR_OTHER)

Related information

- MPI_KEYVAL_CREATE
- MPI_TYPE_FREE_KEYVAL

MPI_TYPE_CREATE_RESIZED, MPI_Type_create_resized

Duplicates a data type and changes the upper bound, lower bound, and extent.

C synopsis

```
#include <mpi.h>
int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint extent,
       MPI_Datatype *newtype);
```

C++ synopsis

```
#include mpi.h
MPI::Datatype MPI::Datatype::Create_resized(const MPI::Aint lb,
         const MPI::Aint extent)
         const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_CREATE_RESIZED(INTEGER OLDTYPE, INTEGER(KIND=MPI_ADDRESS_KIND) LB,
   INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, INTEGER NEWTYPE, INTEGER IERROR)
```

Description

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

Parameters

oldtype
The input data type (handle) (IN)

lb The new lower bound of the data type (integer) (IN)

extent
The new extent of the data type (integer) (IN)

newtype
The output data type (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

The new data type must be committed using MPI_TYPE_COMMIT before it can be used in communication.

Errors

Fatal errors:

Copy callback failed

Invalid datatype

MPI not initialized

MPI already finalized

Null datatype

MPI_TYPE_CREATE_STRUCT, MPI_Type_create_struct

Returns a new data type that represents *count* blocks. Each block 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_create_struct(int count, int array_of_blocklengths[],
    MPI_Aint array_of_displacements[], MPI_Datatype array_of_types[],
    MPI_datatype *newtype);
```

C++ synopsis

```
#include mpi.h
static MPI::Datatype MPI::Datatype::Create_struct(int count,
        const int array_of_blocklengths[],
        const MPI::Aint array_of_displacements[],
        const MPI::Datatype array_of_types[]);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_CREATE_STRUCT(INTEGER COUNT, INTEGER ARRAY_OF_BLOCKLENGTHS(*),
         INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF DISPLACEMENTS(*),
         INTEGER ARRAY_OF_TYPES(*), INTEGER NEWTYPE, INTEGER IERROR)
```

Description

This subroutine returns a new data type that represents *count* blocks. Each block is defined by an entry in *array_of_blocklengths*, *array_of_displacements* and *array_of_types*. Displacements are expressed in bytes.

Parameters

count
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
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
The byte displacement of each block (array of integer) (IN)

array_of_types
The type of the elements in each block. That is, block*(i)* is made of a concatenation of type *array_of_types(i)* (array of handles to data type objects) (IN)

newtype
The new data type (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

newtype must be committed using MPI_TYPE_COMMIT before being used for communication.

MPI_TYPE_CREATE_STRUCT is synonymous with MPI_TYPE_STRUCT in C and C++, or in Fortran when default INTEGERs are address-sized. (MPI_TYPE_STRUCT is not available in C++.) In Fortran, MPI_TYPE_CREATE_STRUCT accepts arguments of type INTEGER(KIND=MPI_ADDRESS_KIND) for *array_of_displacements* where type MPI_Aint is used in C.

If Fortran 64-bit applications must be written to be portable to systems that do not support Fortran 90 KIND declarations, it is also correct to declare the (KIND=MPI_ADDRESS_KIND) arguments as INTEGER*8. The KIND format has the advantage of allowing the same source code to compile for either 32-bit or 64-bit processing. The MPI_TYPE_STRUCT binding is retained to support old codes but any new code, whether C or Fortran should use MPI_TYPE_CREATE_STRUCT.

Note that the MPI-1 routines that use a Fortran INTEGER where C bindings specify MPI_Aint will work correctly as long as the values they represent fit in a 32-bit signed integer. It can be difficult to predict reliably when values will remain in range and the loss of high-order bits when overflow does occur will not raise an MPI error, so this may lead to obscure application failures.

Errors

Fatal 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_GET_ADDRESS
- MPI_TYPE_CREATE_HINDEXED
- MPI_TYPE_CREATE_HVECTOR

MPI_TYPE_CREATE_SUBARRAY, MPI_Type_create_subarray

Returns a new data type 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);
```

C++ synopsis

```
#include mpi.h
MPI::Datatype MPI::Datatype::Create_subarray(int ndims, const int array_of_sizes[],
          const int array_of_subsizes[],
          const int array_of_starts[],
          int order) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_CREATE_SUBARRAY (INTEGER NDIMS,INTEGER ARRAY_OF_SIZES(*),
     INTEGER ARRAY_OF_SUBSIZES(*),INTEGER ARRAY_OF_STARTS(*),
     INTEGER ORDER,INTEGER OLDTYPE,INTEGER NEWTYPE,INTEGER IERROR)
```

Description

MPI_TYPE_CREATE_SUBARRAY creates an MPI data type 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.

Parameters

ndims
The number of array dimensions, a positive integer (IN)

array_of_sizes
The number of elements of type *oldtype* in each dimension of the full array (array of positive integers) (IN)

array_of_subsizes
The number of type *oldtype* in each dimension of the subarray (array of positive integers) (IN)

array_of_starts
The starting coordinates of the subarray in each dimension (array of nonnegative integers) (IN)

order
The array storage order *flag* (state) (IN)

oldtype
The array element data type (handle) (IN)

newtype
The new data type (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

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 data type

Invalid datatype
oldtype cannot be: MPI_LB, MPI_PACKED, or MPI_UB

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_DELETE_ATTR, MPI_Type_delete_attr

Deletes an attribute from a data type.

C synopsis

```
#include <mpi.h>
int MPI_Type_delete_attr (MPI_Datatype type, int type_keyval);
```

C++ synopsis

```
#include mpi.h
void MPI::Datatype::Delete_attr(int type_keyval);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_DELETE_ATTR(INTEGER TYPE, INTEGER TYPE_KEYVAL, INTEGER IERROR)
```

Description

This subroutine deletes an attribute from data type *type*.

Parameters

type
The data type from which the attribute is deleted (handle) (INOUT)

type_keyval
The key value (integer) (IN)

IERROR
The Fortran return code. It is always the last argument.

Errors

Fatal errors:

Invalid datatype (MPI_ERR_TYPE)

Null datatype (MPI_ERR_TYPE)

Invalid attribute key (MPI_ERR_ARG) *type_keyval* **is undefined**

Wrong keytype (MPI_ERR_ARG) attribute key is not a datatype key

Predefined attribute key (MPI_ERR_ARG)

MPI not initialized (MPI_ERR_OTHER)

MPI already finalized (MPI_ERR_OTHER)

Related information

- MPI_TYPE_GET_ATTR
- MPI_TYPE_SET_ATTR

MPI_TYPE_DUP, MPI_Type_dup

Duplicates a data type, including any cached information.

C synopsis

```
#include <mpi.h>
int MPI_Type_dup (MPI_Datatype type, MPI_Datatype *newtype);
```

C++ synopsis

```
#include mpi.h
MPI::Datatype MPI::Datatype::Dup() const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_DUP(INTEGER TYPE, INTEGER NEWTYPE, INTEGER IERROR)
```

Description

This subroutine is a new type constructor that duplicates the existing type with associated key values. For each key value, the respective copy callback function determines the attribute value associated with this key in the new communicator. One particular action that a copy callback may take is to delete the attribute from the new data type. MPI_TYPE_DUP returns in *newtype* a new data type with exactly the same properties as *type* and any copied cached information. The new data type has an identical upper bound and lower bound and yields the same net result when fully decoded with the MPI_TYPE_GET_CONTENTS and MPI_TYPE_GET_ENVELOPE functions. The *newtype* has the same committed state as *type*.

Parameters

type
The data type (handle) (IN)

newtype
A copy of *type* (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_TYPE_DUP semantic is different from that of MPI_FILE_GET_VIEW and MPI_TYPE_GET_CONTENTS. The latter subroutines return a new reference to an existing data type object, while MPI_TYPE_DUP creates a new object. The distinction becomes important only when using data type attributes.

Errors

Fatal errors:

Invalid datatype (MPI_ERR_TYPE)

Null datatype (MPI_ERR_TYPE)

MPI not initialized (MPI_ERR_OTHER)

MPI already finalized (MPI_ERR_OTHER)

Related information

- MPI_TYPE_FREE
- MPI_TYPE_SET_NAME

MPI_TYPE_EXTENT, MPI_Type_extent

Returns the extent of any defined data type.

C synopsis

```
#include <mpi.h>
int MPI_Type_extent(MPI_Datatype datatype,MPI_Aint *size);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_EXTENT(INTEGER DATATYPE,INTEGER EXTENT,INTEGER IERROR)
```

Description

This subroutine returns the extent of a data type. The default extent of a data type is the span from the first byte to the last byte occupied by entries in this data type and rounded up to satisfy alignment requirements.

Parameters

datatype
The data type (handle) (IN)

size
The data type extent (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_TYPE_GET_EXTENT supersedes MPI_TYPE_EXTENT.

Rounding for alignment is not done when MPI_UB is used to define the data type. Types defined with MPI_LB, MP_UB, or with any type that itself contains MPI_LB or MPI_UB may return an extent that is not directly related to the layout of data in memory. Refer to "MPI_TYPE_STRUCT, MPI_Type_struct" on page 548 or "MPI_TYPE_CREATE_STRUCT, MPI_Type_create_struct" on page 509 for more information on MPI_LB and MPI_UB.

MPI_TYPE_CREATE_RESIZED can also alter default extent.

You can still use this subroutine in Fortran 64-bit applications if you know that all data type extents can be represented by an INTEGER, but you do so at your own risk. MPI_TYPE_GET_EXTENT should be used in new codes.

Errors

Invalid datatype

MPI not initialized

MPI already finalized

Related information

- MPI_TYPE_SIZE

MPI_Type_f2c

Returns a C handle to a data type.

C synopsis

```
#include <mpi.h>
MPI_Type MPI_Type_f2c(MPI_Fint datatype);
```

Description

This function does not have C++ or Fortran bindings. MPI_Type_f2c returns a C handle to a data type. If *datatype* is a valid Fortran handle to a data type, MPI_Type_f2c returns a valid C handle to that same data type. If *datatype* is set to the Fortran value MPI_DATATYPE_NULL, MPI_Type_f2c returns the equivalent null C handle. If *datatype* is not a valid Fortran handle, MPI_Type_f2c returns a non-valid C handle. The converted handle is returned as the function's value. There is no error detection or return code.

Parameters

datatype
The data type (handle) (IN)

Related information

- MPI_Type_c2f

MPI_TYPE_FREE, MPI_Type_free

Marks a data type for deallocation.

C synopsis

```
#include <mpi.h>
int MPI_Type_free(MPI_Datatype *datatype);
```

C++ synopsis

```
#include mpi.h
void MPI::Datatype::Free();
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_FREE(INTEGER DATATYPE,INTEGER IERROR)
```

Description

This subroutine marks the data type object associated with *datatype* for deallocation. It sets *datatype* to MPI_DATATYPE_NULL. All communication currently using this data type completes normally. Derived data types defined from the freed data type are not affected.

Parameters

datatype
The data type to be freed (handle) (INOUT)

IERROR
The Fortran return code. It is always the last argument.

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 data type 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_FILE_GET_VIEW
- MPI_TYPE_COMMIT
- MPI_TYPE_GET_CONTENTS
- MPI_TYPE_GET_ENVELOPE

MPI_TYPE_FREE_KEYVAL, MPI_Type_free_keyval

Frees a data type key value.

C synopsis

```
#include <mpi.h>
int MPI_Type_free_keyval (int *type_keyval);
```

C++ synopsis

```
#include mpi.h
void MPI::Datatype::Free_keyval(int& type_keyval);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_FREE_KEYVAL(INTEGER TYPE_KEYVAL, INTEGER IERROR)
```

Description

This subroutine frees the key referred to by the *type_keyval* argument and sets *keyval* to MPI_KEYVAL_INVALID.

Parameters

type_keyval
The key value (integer) (INOUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Fatal errors:

Invalid attribute key (MPI_ERR_ARG) *type_keyval* **is undefined**

Predefined attribute key (MPI_ERR_ARG)

Wrong keytype (MPI_ERR_ARG) attribute key is not a datatype key

MPI not initialized (MPI_ERR_OTHER)

MPI already finalized (MPI_ERR_OTHER)

Related information

- MPI_TYPE_CREATE_KEYVAL

MPI_TYPE_GET_ATTR, MPI_Type_get_attr

Attaches an attribute to a data type.

C synopsis

```
#include <mpi.h>
int MPI_Type_get_attr (MPI_Datatype type, int type_keyval,
                       void *attribute_val, int *flag);
```

C++ synopsis

```
#include mpi.h
bool MPI::Datatype::Get_attr(int type_keyval, void* attribute_val) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_GET_ATTR(INTEGER TYPE, INTEGER TYPE_KEYVAL, INTEGER(KIND=MPI_ADDRESS_KIND)
                  ATTRIBUTE_VAL, LOGICAL FLAG, INTEGER IERROR)
```

Description

This subroutine attaches an attribute to data type *type*.

Parameters

type
The data type to which the attribute is attached (handle) (IN)

type_keyval
The key value (integer) (IN)

attribute_val
The attribute value, unless *flag* = **false** (OUT)

flag
Set to **false** if no attribute is associated with the key (logical) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

The implementation of MPI_TYPE_SET_ATTR and MPI_TYPE_GET_ATTR involves saving a single word of information in the data type. The languages C and Fortran have different approaches to using this capability:

In C: As the programmer, you normally define a struct that holds arbitrary attribute information. Before calling MPI_TYPE_SET_ATTR, you allocate some storage for the attribute structure and then call MPI_TYPE_SET_ATTR to record the address of this structure. You must make sure 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_TYPE_GET_ATTR. Both MPI_TYPE_SET_ATTR and MPI_TYPE_GET_ATTR take a **void*** parameter, but this does not imply that the same parameter is passed to either one.

In Fortran:
MPI_TYPE_SET_ATTR records an address-size integer and MPI_TYPE_GET_ATTR returns the address-size integer. As the programmer, you can choose to encode all attribute information in this

integer or maintain some kind of database in which the integer can index. Either of these approaches will port to other MPI implementations.

Many of the Fortran compilers include an additional feature that allows some of the same functions a C programmer would use. These compilers support the POINTER type, often referred to as a *Cray pointer*. XL Fortran is one of the compilers that supports the POINTER type. For more information, see *IBM XL Fortran Compiler Reference*.

Errors

Fatal errors:

Invalid datatype (MPI_ERR_TYPE)

Null datatype (MPI_ERR_TYPE)

Invalid attribute key (MPI_ERR_ARG) *type_keyval* **is undefined**

Wrong keytype (MPI_ERR_ARG) attribute key is not a datatype key

MPI not initialized (MPI_ERR_OTHER)

MPI already finalized (MPI_ERR_OTHER)

Related information

- MPI_TYPE_DELETE_ATTR
- MPI_TYPE_SET_ATTR

MPI_TYPE_GET_CONTENTS, MPI_Type_get_contents

Obtains the arguments used in the creation of the data type.

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[]);
```

C++ synopsis

```
#include mpi.h
void MPI::Datatype::Get_contents(int max_integers, int max_addresses,
                                 int max_datatypes, int array_of_integers[],
                                 MPI::Aint array_of_addresses[],
                                 MPI::Datatype array_of_datatypes[])
     const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_GET_CONTENTS(INTEGER DATATYPE, INTEGER MAX_INTEGERS,INTEGER MAX_ADDRESSES,
        INTEGER MAX_DATATYPES,INTEGER ARRAY_of_INTEGERS(*),
        INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_ADDRESSES(*),
        INTEGER ARRAY_of_DATATYPES(*), INTEGER IERROR)
```

Description

MPI_TYPE_GET_CONTENTS identifies the combiner and returns the arguments that were used with this combiner to create the data type 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 data type is a predefined type that may not be decoded. The data type 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 data type handle that must eventually be freed if it represents a derived type.

Parameters

datatype
The data type to access (handle) (IN)

max_integers
The number of elements in array_of_integers (non-negative integer) (IN)

max_addresses
The number of elements in the array_of_addresses (non-negative integer) (IN)

max_datatypes
The number of elements in *array_of_datatypes* (non-negative integer) (IN)

array_of_integers
Contains integer arguments used in the constructing data type (array of integers) (OUT)

array_of_addresses
Contains address arguments used in the constructing data type (array of integers) (OUT)

array_of_datatypes
Contains data type arguments used in the constructing data type (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. Combiners and constructor arguments

Constructor argument	C location	Fortran location	ni na nd
MPI_COMBINER_DUP			
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

Table 4. Combiners and constructor arguments (continued)

Constructor argument	C location	Fortran location	ni na nd
MPI_COMBINER_INDEXED_BLOCK			
count	i[0]	I(1)	count+2
blocklength	i[1]	I(2)	0
array_of_displacements	i[2] to i[i[0]+1]	I(3) to I(I(1)+2)	1
oldtype	d[0]	D(1)	
MPI_COMBINER_STRUCT MPI_COMBINER_STRUCT_INTEGER			
count	i[0]	I(1)	count+1
array_of_blocklengths	i[1] to i[i[0]]	I(2) to I(I(1)+1)	count
array_of_displacements	a[0] to a[i[0]-1]	A(1) to A(I(1))	count
array_of_types	d[0] to d[i[0]-1]	D(1)	
MPI_COMBINER_SUBARRAY			
ndims	i[0]	I(1)	3*ndims+2
array_of_sizes	i[1] to i[i[0]]	I(2) to I(I(1)+1)	0
array_of_subsizes	i[i[0]+1] to i[2*i[0]]	I(I(1)+2) to I(2*I(1)+1)	1
array_of_starts	i[2*i[0]+1] to i[3*i[0]]	I(2*I(1)+2) to I(3*I(1)+1)	
order	i[3*i[0]+1]	I(3*I(1)+2)	
oldtype	d[0]	D(1)	
MPI_COMBINER_DARRAY			
size	i[0]	I(1)	4*ndims+4
rank	i[1]	I(2)	0
ndims	i[2]	I(3)	1
array_of_gsizes	i[3] to i[i[2]+2]	I(4) to I(I(3)+3)	
array_of_distribs	i[i[2]+3] to i[2*i[2]+2]	I(I(3)+4) to I(2*I(3)+3)	
array_of_dargs	i[2*i[2]+3] to i[3*i[2]+2]	I(2*I(3)+4) to I(3*I(3)+3)	
array_of_psizes	i[3*i[2]+3] to i[4*i[2]+2]	I(3*I(3)+4) to I(4*I(3)+3)	
order	i[4*i[2]+3]	I(4*I(3)+4)	
oldtype	d[0]	D(1)	
MPI_COMBINER_F90_REAL MPI_COMBINER_F90_COMPLEX			
p	i[0]	I(1)	2
r	i[1]	I(2)	0
			0
MPI_COMBINER_F90_INTEGER			
r	i[0]	I(1)	
			1
			0
			0
MPI_COMBINER_RESIZED			

Table 4. Combiners and constructor arguments (continued)

Constructor argument	C location	Fortran location	ni na nd
lb	a[0]	A(1)	0
extent	a[1]	A(2)	2
oldtype	d[0]	D(1)	1

Notes

An MPI type constructor, such as MPI_TYPE_CONTIGUOUS, creates a data type object within MPI and gives a handle for that object to the caller. This handle represents one reference to the object. In IBM PE MPI, the MPI data types obtained with calls to MPI_TYPE_GET_CONTENTS are new handles for the existing data type objects. The number of handles (references) given to the user is tracked by a reference counter in the object. MPI cannot discard a data type 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 aatype 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 bbtype. Because both handles identify the same data type 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 aatype and bbtype. 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

Determines the constructor that was used to create the data type and the amount of data that will be returned by a call to MPI_TYPE_GET_CONTENTS for the same data type.

C synopsis

```
#include <mpi.h>
int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers,
    int *num_addresses, int *num_datatypes, int *combiner);
```

C++ synopsis

```
#include mpi.h
void MPI::Datatype::Get_envelope(int& num_integers, int& num_addresses,
     int& num_datatypes, int& combiner)
     const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_GET_ENVELOPE(INTEGER DATATYPE, INTEGER NUM_INTEGERS,
    INTEGER NUM_ADDRESSES, INTEGER NUM_DATATYPES, INTEGER COMBINER,
    INTEGER IERROR)
```

Description

MPI_TYPE_GET_ENVELOPE provides information about an unknown data type that 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 data type handle returned by MPI_TYPE_GET_CONTENTS or MPI_FILE_GET_VIEW is for a predefined, named data type. When the combiner is MPI_COMBINER_NAMED, it is an error to call MPI_TYPE_GET_CONTENTS or MPI_TYPE_FREE with the data type.

Parameters

datatype
The data type to access (handle) (IN)

num_integers
The number of input integers used in the call constructing combiner (non-negative integer) (OUT)

num_addresses
The number of input addresses used in the call constructing combiner (non-negative integer) (OUT)

num_datatypes
The number of input data types used in the call constructing combiner (non-negative integer) (OUT)

combiner
The combiner (state) (OUT)

This is a list of the combiners and the calls associated with them.

Combiner
What it represents

MPI_COMBINER_NAMED
A named, predefined data type

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

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

Errors

Invalid datatype

MPI already finalized

MPI not initialized

Related information

- MPI_TYPE_FREE
- MPI_TYPE_GET_CONTENTS

MPI_TYPE_GET_EXTENT, MPI_Type_get_extent

Returns the lower bound and the extent of any defined data type.

C synopsis

```
#include <mpi.h>
int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb, MPI_Aint *extent);
```

C++ synopsis

```
#include mpi.h
void MPI::Datatype::Get_extent(MPI::Aint& lb, MPI::Aint& extent)
          const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_GET_EXTENT(INTEGER DATATYPE, INTEGER(KIND=MPI_ADDRESS_KIND) LB,
      INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, INTEGER IERROR)
```

Description

This subroutine returns the lower bound and the extent of a data type. By default, the extent of a data type is the span from the first byte to the last byte occupied by entries in this data type and rounded up to satisfy alignment requirements.

Parameters

datatype
The data type (handle) (IN)

lb The lower bound of the data type (integer) (OUT)

extent
The extent of the data type (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

Rounding for alignment is not done when MPI_UB is used to define the data type. Types defined with MPI_LB, MP_UB, or with any type that itself contains MPI_LB or MPI_UB may return an extent that is not directly related to the layout of data in memory. Refer to “MPI_TYPE_STRUCT, MPI_Type_struct” on page 548 or “MPI_TYPE_CREATE_STRUCT, MPI_Type_create_struct” on page 509 for more information on MPI_LB and MPI_UB.

MPI_TYPE_CREATE_RESIZED can also alter default extent.

In Fortran, MPI_TYPE_GET_EXTENT accepts arguments of type INTEGER(KIND=MPI_ADDRESS_KIND) for *lb* and *extent* arguments where type MPI_Aint is used in C.

If Fortran 64-bit applications must be written to be portable to systems that do not support Fortran 90 KIND declarations, it is also correct to declare the (KIND=MPI_ADDRESS_KIND) arguments as INTEGER*8. The KIND format has the advantage of allowing the same source code to compile for either 32-bit or 64-bit processing. The MPI_TYPE_xxxx binding is retained to support old codes but any new code, whether C or Fortran should use MPI_TYPE_CREATE_xxxxx.

Note that the MPI-1 routines that use a Fortran INTEGER where C bindings specify MPI_Aint will work correctly as long as the values they represent fit in a 32-bit signed integer. It can be difficult to predict reliably when values will remain in range and the loss of high-order bits when overflow does occur will not raise an MPI error, so this may lead to obscure application failures.

Errors

Fatal errors:

Invalid datatype

MPI not initialized

MPI already finalized

Related information

- MPI_TYPE_SIZE

MPI_TYPE_GET_NAME, MPI_Type_get_name

Returns the name that was last associated with a data type.

C synopsis

```
#include <mpi.h>
int MPI_Type_get_name(MPI_Datatype type, char *type_name, int *resultlen);
```

C++ synopsis

```
#include mpi.h
void MPI::Datatype::Get_name(char* type_name, int& resultlen)
        const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_GET_NAME(INTEGER TYPE, CHARACTER*(*) TYPE_NAME, INTEGER RESULTLEN,
    INTEGER IERROR)
```

Description

This subroutine returns the name that was last associated with the specified data type. The name can be set and retrieved from any language. The same name is returned independent of the language used. The name should be allocated so it can hold a resulting string that is the length of MPI_MAX_OBJECT_NAME. For IBM PE MPI, the value of MPI_MAX_OBJECT_NAME is 256.
MPI_TYPE_GET_NAME returns a copy of the set name in *type_name*.

Parameters

type
The data type with the name to be returned (handle) (IN)

type_name
The name previously stored on the data type, or an empty string if no such name exists (string) (OUT)

resultlen
The length of the returned name (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

If you did not associate a name with a data type, or if an error occurs, MPI_TYPE_GET_NAME returns an empty string (all spaces in Fortran or "" in C and C++). Named predefined data types have the default names of the data type name. For example, MPI_WCHAR has the default name of MPI_WCHAR.

It is safe simply to print the string returned by MPI_TYPE_GET_NAME, as it is always a valid string even if there was no name.

Errors

Fatal errors:

Invalid datatype

MPI already finalized

MPI not initialized

Related information

- MPI_TYPE_DUP
- MPI_TYPE_SET_NAME

MPI_TYPE_GET_TRUE_EXTENT, MPI_Type_get_true_extent

Returns the true extent of any defined data type.

C synopsis

```
#include <mpi.h>
int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb,
        MPI_Aint *true_extent);
```

C++ synopsis

```
#include mpi.h
void MPI::Datatype::Get_true_extent(MPI::Aint& true_lb,
        MPI::Aint& true_extent)
        const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_GET_TRUE_EXTENT(INTEGER DATATYPE, INTEGER(KIND=MPI_ADDRESS_KIND) TRUE_LB,
           INTEGER(KIND=MPI_ADDRESS_KIND) TRUE_EXTENT, INTEGER IERROR)
```

Description

This subroutine returns the true extent of a data type. *true_lb* returns the offset of the lowest unit of storage that is addressed by the data type. *true_extent* returns the true size of the data type. The true extent of a data type is the minimum number of bytes of memory that are needed to hold it (the data type), uncompressed.

Parameters

datatype
The data type about which to get information (handle) (IN)

true_lb
The true lower bound of the data type (integer) (OUT)

true_extent
The true size of the data type (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

All other MPI subroutines that refer to *extent* use that term to identify the stride at which the data type is applied when used more that once. That stride is often the same as the footprint in address space, but because MPI allows default extent (stride) to be modified, a data type's *extent* and *true extent* may not always be the same.

The *true extent* tells you how much space is required in the address space to store one instance of the data type. However, for two or more instances, multiplying *true extent* by the number of instances is not useful. To determine the footprint in address space for two or more instances of the data type, you must also use MPI_TYPE_GET_EXTENT to learn the stride.

Errors

Fatal errors:

Invalid datatype

MPI not initialized

MPI already finalized

MPI_TYPE_HINDEXED, MPI_Type_hindexed

Returns a new data type 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' or USE MPI
MPI_TYPE_HINDEXED(INTEGER COUNT,INTEGER ARRAY_OF_BLOCKLENGTHS(*),
    INTEGER ARRAY_OF DISPLACEMENTS(*),INTEGER OLDTYPE,
    INTEGER NEWTYPE,INTEGER IERROR)
```

Description

This subroutine returns a new data type 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.

Parameters

count
The number of blocks and the number of entries in *array_of_displacements* and *array_of_blocklengths* (non-negative integer) (IN)

array_of_blocklengths
The number of instances of **oldtype** for each block (array of non-negative integers) (IN)

array_of_displacements
A byte displacement for each block (array of integer) (IN)

oldtype
The old data type (handle) (IN)

newtype
The new data type (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_TYPE_CREATE_HINDEXED supersedes MPI_TYPE_HINDEXED.

For Fortran 64-bit codes, an INTEGER may not be enough to represent a displacement. When displacements are known to be small enough, this subroutine remains usable at your own risk. New codes should use MPI_TYPE_CREATE_HINDEXED.

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

Returns a new data type 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' or USE MPI
MPI_TYPE_HVECTOR(INTEGER COUNT,INTEGER BLOCKLENGTH,INTEGER STRIDE,
   INTEGER OLDTYPE,INTEGER NEWTYPE,INTEGER IERROR)
```

Description

This subroutine returns a new data type 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.

Parameters

count
The number of blocks (non-negative integer) (IN)

blocklength
The number of **oldtype** instances in each block (non-negative integer) (IN)

stride
An integer specifying the number of bytes between start of each block. (IN)

oldtype
The old data type (handle) (IN)

newtype
The new data type (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_TYPE_CREATE_HVECTOR supersedes MPI_TYPE_HVECTOR.

For Fortran 64-bit codes, an INTEGER may not be enough to represent the stride. When the stride is known to be small enough, this subroutine remains usable at your own risk. New codes should always use MPI_TYPE_CREATE_HVECTOR.

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

Returns a new data type 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);
```

C++ synopsis

```
#include mpi.h
MPI::Datatype MPI::Datatype::Create_indexed(int count,
         const int array_of_blocklengths[],
         const int array_of_displacements[])
         const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_INDEXED(INTEGER COUNT,INTEGER ARRAY_OF_BLOCKLENGTHS(*),
   INTEGER ARRAY_OF DISPLACEMENTS(*),INTEGER OLDTYPE,
   INTEGER NEWTYPE,INTEGER IERROR)
```

Description

This subroutine returns a new data type 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*).

Parameters

count
The number of blocks and the number of entries in *array_of_displacements* and *array_of_blocklengths* (non-negative integer) (IN)

array_of_blocklengths
The number of instances of **oldtype** in each block (array of non-negative integers) (IN)

array_of_displacements
The displacement of each block in units of extent(**oldtype**) (array of integer) (IN)

oldtype
The old data type (handle) (IN)

newtype
The new data type (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

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

Returns the lower bound of a data type.

C synopsis

```
#include <mpi.h>
int MPI_Type_lb(MPI_Datatype datatype,MPI_Aint *displacement);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_LB(INTEGER DATATYPE,INTEGER DISPLACEMENT,INTEGER IERROR)
```

Description

This subroutine returns the lower bound of a specific data type.

In general, the lower bound is the offset of the lowest address byte in the data type. Data type constructors with explicit MPI_LB and vector constructors with negative stride can produce lb < 0. The lower bound cannot be greater than the 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, MPI_Type_struct" on page 548 for more information on MPI_LB and MPI_UB.

Parameters

datatype
The data type (handle) (IN)

displacement
The displacement of lower bound from the origin in bytes (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_TYPE_GET_EXTENT supersedes MPI_TYPE_LB.

For Fortran 64-bit codes, an INTEGER may not be enough to represent the lower bound. When the lower bound is known to be representable by an INTEGER, this subroutine remains usable at your own risk. New codes should always use MPI_TYPE_GET_EXTENT.

Errors

Invalid datatype

MPI not initialized

MPI already finalized

Related information

- MPI_TYPE_STRUCT
- MPI_TYPE_UB

MPI_TYPE_MATCH_SIZE, MPI_Type_match_size

Returns a reference (handle) to one of the predefined named data types, not a duplicate.

C synopsis

```
#include <mpi.h>
int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *type);
```

C++ synopsis

```
#include mpi.h
static MPI::Datatype MPI::Datatype::Match_size(int typeclass, int size);
```

Fortran synopsis

```
USE MPI
MPI_TYPE_MATCH_SIZE(INTEGER TYPECLASS, INTEGER SIZE, INTEGER TYPE, INTEGER IERROR)
```

Description

This subroutine returns an MPI data type matching a local variable of type (*typeclass, size*). The value of *typeclass* is one of these: MPI_TYPECLASS_REAL, MPI_TYPECLASS_INTEGER, or MPI_TYPECLASS_COMPLEX, corresponding to the desired type class. This type cannot be freed. MPI_TYPE_MATCH_SIZE can be used to obtain a size-specific type that matches a Fortran numeric intrinsic type by first calling MPI_SIZEOF in order to compute the variable size, and then calling MPI_TYPE_MATCH_SIZE to find a suitable data type. In C and C++, you can use the C function sizeof(), instead of MPI_SIZEOF. In addition, for variables of default kind, the variable's size can be computed by a call to MPI_TYPE_GET_EXTENT, if the typeclass is known. It is erroneous to specify a size not supported by the compiler.

Parameters

typeclass
The generic type specifier (integer) (IN)

size
The size, in bytes, of the representation (integer) (IN)

type
The data type with the correct type and size (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Fatal errors:

MPI already finalized

MPI not initialized

No matching MPI intrinsic type

Related information

- MPI_SIZEOF
- MPI_TYPE_GET_EXTENT

MPI_TYPE_SET_ATTR, MPI_Type_set_attr

Attaches the data type attribute value to the data type and associates it with the key.

C synopsis

```
#include <mpi.h>
int MPI_Type_set_attr (MPI_Datatype type, int type_keyval, void *attribute_val);
```

C++ synopsis

```
#include mpi.h
void MPI::Datatype::Set_attr(int type_keyval, const void* attribute_val);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_SET_ATTR(INTEGER TYPE, INTEGER TYPE_KEYVAL, INTEGER(KIND=MPI_ADDRESS_KIND)
                  ATTRIBUTE_VAL, INTEGER IERROR)
```

Description

This subroutine stores the attribute *attribute_val* for subsequent retrieval by MPI_TYPE_GET_ATTR. If an attribute already exists for *type_keyval* on *type*, the attribute delete function is called before the new attribute is stored.

Parameters

type
The data type to which the attribute will be attached (handle) (INOUT)

type_keyval
The key value (integer) (IN)

attribute_val
The attribute value (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

The implementation of MPI_TYPE_SET_ATTR and MPI_TYPE_GET_ATTR involves saving a single word of information in the data type. The languages C and Fortran have different approaches to using this capability:

In C: As the programmer, you normally define a struct that holds arbitrary attribute information. Before calling MPI_TYPE_SET_ATTR, you allocate some storage for the attribute structure and then call MPI_TYPE_SET_ATTR to record the address of this structure. You must make sure 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_TYPE_GET_ATTR. Both MPI_TYPE_SET_ATTR and MPI_TYPE_GET_ATTR take a **void*** parameter, but this does not imply that the same parameter is passed to either one.

In Fortran:
MPI_TYPE_SET_ATTR records an address-size integer and MPI_TYPE_GET_ATTR returns the address-size integer. As the programmer, you can choose to encode all attribute information in this

integer or maintain some kind of database in which the integer can index. Either of these approaches will port to other MPI implementations.

Many of the Fortran compilers include an additional feature that allows some of the same functions a C programmer would use. These compilers support the POINTER type, often referred to as a *Cray pointer*. XL Fortran is one of the compilers that supports the POINTER type. For more information, see *IBM XL Fortran Compiler Reference*

Errors

Fatal errors:

Invalid datatype (MPI_ERR_TYPE)

Null datatype (MPI_ERR_TYPE)

Invalid attribute key (MPI_ERR_ARG) *type_keyval* **is undefined**

Predefined attribute key (MPI_ERR_ARG)

Wrong keytype (MPI_ERR_ARG) attribute key is not a datatype key

MPI not initialized (MPI_ERR_OTHER)

MPI already finalized (MPI_ERR_OTHER)

Related information

- MPI_TYPE_DELETE_ATTR
- MPI_TYPE_GET_ATTR

MPI_TYPE_SET_NAME, MPI_Type_set_name

Associates a name string with a data type.

C synopsis

```
#include <mpi.h>
int MPI_Type_set_name (MPI_Datatype type, char *type_name);
```

C++ synopsis

```
#include mpi.h
void MPI::Datatype::Set_name(const char* type_name);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_SET_NAME(INTEGER TYPE, CHARACTER*(*) TYPE_NAME, INTEGER IERROR)
```

Description

This subroutine lets you associate a name string with a data type. Because the purpose of this name is as an identifier, when the data type is copied or duplicated, the name does not propagate.

MPI_TYPE_SET_NAME is a local (non-collective) operation, which affects only the name of the data type as specified in the task that made the MPI_TYPE_SET_NAME call. There is no requirement that the same (or any) name be assigned to a data type in every task where that data type exists. However, to avoid confusion, it is a good idea to give the same name to a data type in all of the tasks where it exists.

Parameters

type
The data type with the identifier to be set (handle) (INOUT)

type_name
The character string that is saved as the data type's name (string) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

The length of the name that can be stored is limited to the value of MPI_MAX_OBJECT_NAME in Fortran and MPI_MAX_OBJECT_NAME-1 in C and C++ to allow for the null terminator. Attempts to use a longer name will result in truncation of the name. For IBM PE MPI, the value of MPI_MAX_OBJECT_NAME is 256.

Under circumstances of storage exhaustion, an attempt to use a name of any length could fail, therefore the value of MPI_MAX_OBJECT_NAME should be viewed only as a strict upper bound on the name length, not a guarantee that setting names of less than this length will always succeed.

Associating a name with a data type has no effect on the semantics of an MPI program, and (necessarily) increases the storage requirement of the program, because the names must be saved. Therefore, there is no requirement that you use this subroutine to associate names with data types. However, debugging and

profiling MPI applications can be made easier if names are associated with data types, as the debugger or profiler should then be able to present information in a less cryptic manner.

Errors

Fatal errors:

Invalid datatype

MPI already finalized

MPI not initialized

Related information

- MPI_TYPE_DUP
- MPI_TYPE_GET_NAME

MPI_TYPE_SIZE, MPI_Type_size

Returns the number of bytes represented by any defined data type.

C synopsis

```
#include <mpi.h>
int MPI_Type_size(MPI_Datatype datatype,int *size);
```

C++ synopsis

```
#include mpi.h
int MPI::Datatype::Get_size() const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_SIZE(INTEGER DATATYPE,INTEGER SIZE,INTEGER IERROR)
```

Description

This subroutine returns the total number of bytes in the type signature associated with *datatype*. Entries with multiple occurrences in the data type are counted.

Parameters

datatype
The data type (handle) (IN)

size
The data type size (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

This function must be used with some care in 64-bit applications because *size* is an integer and could be subject to overflow.

Errors

Invalid datatype

MPI not initialized

MPI already finalized

Related information

- MPI_TYPE_EXTENT

MPI_TYPE_STRUCT, MPI_Type_struct

Returns a new data type 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' or USE MPI
MPI_TYPE_STRUCT(INTEGER COUNT,INTEGER ARRAY_OF_BLOCKLENGTHS(*),
    INTEGER ARRAY_OF DISPLACEMENTS(*),INTEGER ARRAY_OF_TYPES(*),
    INTEGER NEWTYPE,INTEGER IERROR)
```

Description

This subroutine returns a new data type 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 of constructor. It allows each block to consist of replications of different data types. It is the only constructor that allows MPI pseudo types MPI_LB and MPI_UB. Without these pseudo types, the extent of a data type 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 two characters, it will still have an extent of 8 because it is padded to meet the boundary constraints of an integer. 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 data type. 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 data type 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, suppose 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.

Parameters

count
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
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
The byte displacement of each block (array of integer) (IN)

array_of_types
The data type comprising each block. That is, block*(i)* is made of a concatenation of type *array_of_types(i)*. (array of handles to data type objects) (IN)

newtype
The new data type (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_TYPE_CREATE_STRUCT supersedes MPI_TYPE_STRUCT.

For Fortran 64-bit codes, an array of integer may not be enough to represent *array_of_displacements*. When *array_of_displacements* is known to be representable by an array of integer, this subroutine remains usable at your own risk. New codes should always use MPI_TYPE_CREATE_STRUCT.

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

Returns the upper bound of a data type.

C synopsis

```
#include <mpi.h>
int MPI_Type_ub(MPI_Datatype datatype,MPI_Aint *displacement);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_UB(INTEGER DATATYPE,INTEGER DISPLACEMENT,
     INTEGER IERROR)
```

Description

This subroutine returns the upper bound of a specific data type.

The upper bound is the displacement you use in locating the origin byte of the next instance of *datatype* for operations that use *count* and *datatype*. In the normal case, *ub* represents the displacement of the highest address byte of the data type + *e* (where *e* >= 0 and results in (*ub* - *lb*) being a multiple of the boundary requirement for the most boundary constrained type in the data type). 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, MPI_Type_struct" on page 548 for more information on MPI_LB and MPI_UB.

Parameters

datatype
The data type (handle) (IN)

displacement
The displacement of the upper bound from the origin, in bytes (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_TYPE_GET_EXTENT supersedes MPI_TYPE_UB.

For Fortran 64-bit codes, an INTEGER may not be enough to represent the upper bound. When the upper bound is known to be representable by an INTEGER, this subroutine remains usable at your own risk. New codes should always use MPI_TYPE_GET_EXTENT.

Errors

Invalid datatype

MPI not initialized

MPI already finalized

Related information

- MPI_TYPE_LB
- MPI_TYPE_STRUCT

MPI_TYPE_VECTOR, MPI_Type_vector

Returns a new data type 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);
```

C++ synopsis

```
#include mpi.h
MPI::Datatype MPI::Datatype::Create_vector(int count, int blocklength,
        int stride) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_TYPE_VECTOR(INTEGER COUNT,INTEGER BLOCKLENGTH,INTEGER STRIDE,
  INTEGER OLDTYPE,INTEGER NEWTYPE,INTEGER IERROR)
```

Description

This subroutine returns a new data type 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*).

Parameters

count
The number of blocks (non-negative integer) (IN)

blocklength
The number of **oldtype** instances in each block (non-negative integer) (IN)

stride
The number of units between the start of each block (integer) (IN)

oldtype
The old data type (handle) (IN)

newtype
The new data type (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

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

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Datatype::Unpack(const void* inbuf, int insize, void* outbuf,
      int outcount, int& position,
      const MPI::Comm& comm) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_UNPACK(CHOICE INBUF,INTEGER INSIZE,INTEGER POSITION,CHOICE OUTBUF,
    INTEGER OUTCOUNT,INTEGER DATATYPE,INTEGER COMM,INTEGER IERROR)
```

Description

This subroutine 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.

Parameters

inbuf
The input buffer start (choice) (IN)

insize
An integer specifying the size of input buffer in bytes (IN)

position
An integer specifying the current packed buffer offset in bytes (INOUT)

outbuf
The output buffer start (choice) (OUT)

outcount
An integer specifying the number of instances of *datatype* to be unpacked (IN)

datatype
The data type of each output data item (handle) (IN)

comm
The communicator for the packed message (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

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

Negative length or position for buffer
outsize < 0 or *position* < 0

MPI not initialized

MPI already finalized

Related information

- MPI_PACK

MPI_UNPACK_EXTERNAL, MPI_Unpack_external

Unpacks the message into the specified receive buffer from the specified packed buffer, using the external32 data format.

C synopsis

```
#include <mpi.h>
int MPI_Unpack_external(char *datarep, void *inbuf, MPI_Aint insize,
   MPI_Aint *position, void *outbuf, int outcount,
   MPI_Datatype datatype);
```

C++ synopsis

```
#include mpi.h
void MPI::Datatype::Unpack_external(const char* datarep, const void* inbuf,
        MPI::Aint insize, MPI::Aint& position,
        void* outbuf, int outcount) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_UNPACK_EXTERNAL(CHARACTER*(*) DATAREP, CHOICE INBUF(*),
      INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE,
      INTEGER(KIND=MPI_ADDRESS_KIND) POSITION,
      CHOICE OUTBUF(*), INTEGER OUTCOUNT, INTEGER DATATYPE, INTEGER IERROR)
```

Description

This subroutine 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_EXTERNAL.

If you are using IBM PE for Linux, note that MPI_UNPACK_EXTERNAL is currently not supported on IBM System x servers.

Parameters

datarep
The data representation (string) (IN)

inbuf
The input buffer start (choice) (IN)

insize
An integer specifying the size of input buffer in bytes (IN)

position
An integer specifying the current position in the buffer, in bytes (INOUT)

outbuf
The output buffer start (choice) (OUT)

outcount
An integer specifying the number of output data items (IN)

datatype
The data type of each output data item (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

In MPI_UNPACK_EXTERNAL, 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 datarep

Invalid datatype

Type is not committed

Inbuf too small

Negative length or position for buffer
outsize < 0 or *position* < 0

MPI not initialized

MPI already finalized

Related information

- MPI_PACK_EXTERNAL

MPI_UNPUBLISH_NAME, MPI_Unpublish_name

Unpublishes a service name that was previously published.

C synopsis

```
#include <mpi.h>
int MPI_Unpublish_name(char *service_name, MPI_Info info, char *port_name);
```

C++ synopsis

```
#include <mpi.h>
void MPI::Unpublish_name(const char* service_name, const MPI::Info& info,
const char* port_name);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
INTEGER INFO, IERROR
CHARACTER*(*) SERVICE_NAME, PORT_NAME
```

Description

This subroutine unpublishes a service name that was previously published. Attempting to unpublish a name that has not been published or has already been unpublished is erroneous, and returns an error in class **MPI_ERR_SERVICE**.

All published names must be unpublished before the corresponding port is closed and before the publishing task exits.

Parameters

service_name
A service name (string) (IN)

info
An info is an object containing {key,value} pairs. IBM PE MPI **MPI_UNPUBLISH_NAME** does not recognize any info keys. **MPI_INFO_NULL** is always valid (IN)

port_name
A port name (string) (IN)

Errors

Invalid service name passed to MPI_UNPUBLISH_NAME

Invalid port name (NULL)

Related information

- **MPI_PUBLISH_NAME**

MPI_WAIT, MPI_Wait

Waits for a nonblocking operation to complete.

C synopsis

```
#include <mpi.h>
int MPI_Wait(MPI_Request *request,MPI_Status *status);
```

C++ synopsis

```
#include mpi.h
void MPI::Request::Wait();
```

```
#include mpi.h
void MPI::Request::Wait(MPI::Status& status);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_WAIT(INTEGER REQUEST,INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
```

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 nonlocal 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 only by the return code.

Passing MPI_STATUS_IGNORE for the *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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 or test call does not contain meaningful source, tag, or message size information.

When you use this subroutine in a threads 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 *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

Parameters

request
The request to wait for (handle) (INOUT)

status
The status object (Status) (INOUT). Note that in Fortran a single status object is an array of integers.

IERROR
The Fortran return code. It is always the last argument.

Errors

A GRequest free function returned an error

A GRequest query function returned an error

Invalid status ignore value

Invalid form of status ignore

Invalid request handle

Truncation occurred

MPI not initialized

MPI already finalized

Develop mode error if:

Illegal buffer update (ISEND)

Inconsistent datatype (MPE_I collectives)

Inconsistent message length (MPE_I collectives)

Inconsistent op (MPE_I collectives)

Match of blocking and non-blocking collectives (MPE_I collectives)

Related information

- MPI_TEST
- MPI_WAITALL
- MPI_WAITANY
- MPI_WAITSOME

MPI_WAITALL, MPI_Waitall

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);
```

C++ synopsis

```
#include mpi.h
void MPI::Request::Waitall(int count, MPI::Request req_array[]);
```

```
#include mpi.h
void MPI::Request::Waitall(int count, MPI::Request req_array[],
      MPI::Status stat_array[]);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_WAITALL(INTEGER COUNT,INTEGER ARRAY_OF_ REQUESTS(*),
      INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*),INTEGER IERROR)
```

Description

This subroutine 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 invocation 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.

Passing MPI_STATUSES_IGNORE for the *array_of statuses* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *array_of statuses*, you can avoid having to allocate a status object array in programs that do not need to examine the status fields.

When you use this subroutine in a threads 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 *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

Parameters

count
The lists length (integer) (IN)

array_of_requests
An array of requests of length *count* (array of handles) (INOUT)

array_of_statuses
An array of status objects of length *count* (array of status) (INOUT). Note that in Fortran a status object is itself an array.

IERROR
The Fortran return code. It is always the last argument.

Errors

A GRequest free function returned an error

A GRequest query function returned an error

Invalid status ignore value

Invalid form of status ignore

Invalid count
count < **0**

Invalid request array

Invalid request

Truncation occurred

MPI not initialized

MPI already finalized

Develop mode error if:

Illegal buffer update (ISEND)

Inconsistent datatype (MPE_I collectives)

Inconsistent message length (MPE_I collectives)

Inconsistent op (MPE_I collectives)

Match of blocking and non-blocking collectives (MPE_I collectives)

Related information

- MPI_TESTALL
- MPI_WAIT

MPI_WAITANY, MPI_Waitany

Waits for any single nonblocking operation in the array of requests to complete.

C synopsis

```
#include <mpi.h>
int MPI_Waitany(int count,MPI_Request *array_of_requests,
    int *index,MPI_Status *status);
```

C++ synopsis

```
#include mpi.h
int MPI::Request::Waitany(int count, MPI::Request array[]);
```

```
#include mpi.h
int MPI::Request::Waitany(int count, MPI::Request array[],
     MPI::Status& status);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_WAITANY(INTEGER COUNT,INTEGER ARRAY_OF_REQUESTS(*),INTEGER INDEX,
      INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
```

Description

This subroutine 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 invocation 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.

Passing MPI_STATUS_IGNORE for the *status* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *status*, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

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 or test call does not contain meaningful source, tag, or message size information.

When you use this subroutine in a threads 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 *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

Parameters

count
The list length (integer) (IN)

array_of_requests
The array of requests (array of handles) (INOUT)

index
The index of the handle for the operation that completed (integer) (OUT)

status
A status object (Status) (INOUT). Note that in Fortran a single status object is an array of integers.

IERROR
The Fortran return code. It is always the last argument.

Notes

The array is indexed from **0** in C and from **1** in Fortran.

The use of this routine makes the order in which your application completes the requests nondeterministic. An application that processes messages in whatever order they complete must not make assumptions about that order. For example, if:

```
((msgA op msgB) op msgC)
```

can give a different answer than:

```
((msgB op msgC) op msgA)
```

the application must be prepared to accept either answer as *correct*, and must **not** assume a second run of the application will give the same answer.

Errors

A GRequest free function returned an error

A GRequest query function returned an error

Invalid status ignore value

Invalid form of status ignore

Invalid count
count < **0**

Invalid requests array

Invalid requests

Truncation occurred

MPI not initialized

MPI already finalized

Develop mode error if:

Illegal buffer update (ISEND)

Inconsistent datatype (MPE_I collectives)

Inconsistent message length (MPE_I collectives)

Inconsistent op (MPE_I collectives)

Match of blocking and non-blocking collectives (MPE_I collectives)

Related information

- MPI_TESTANY
- MPI_WAIT

MPI_WAITSOME, MPI_Waitsome

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);
```

C++ synopsis

```
#include mpi.h
int MPI::Request::Waitsome(int incount, MPI::Request req_array[],
             int array_of_indices[]);
```

```
#include mpi.h
int MPI::Request::Waitsome(int incount, MPI::Request req_array[],
             int array_of_indices[],
             MPI::Status stat_array[]);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_WAITSOME(INTEGER INCOUNT,INTEGER ARRAY_OF_REQUESTS,INTEGER OUTCOUNT,
      INTEGER ARRAY_OF_INDICES(*),INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*),
      INTEGER IERROR)
```

Description

This subroutine 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.

Passing MPI_STATUSES_IGNORE for the *array_of_statuses* argument causes IBM PE MPI to skip filling in the status fields. By passing this value for *array_of_statuses*, you can avoid having to allocate a status object array in programs that do not need to examine the status fields.

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 or test call does not contain meaningful source, tag, or message size information.

When you use this subroutine in a threads 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 *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information on programming with MPI in a threads environment.

Parameters

incount
The length of *array_of_requests*, *array_of_indices*, and *array_of_statuses* (integer) (IN)

array_of_requests
An array of requests (array of handles) (INOUT)

outcount
The number of completed requests (integer) (OUT)

array_of_indices
The array of indices of operations that completed (array of integers) (OUT)

array_of_statuses
The array of status objects for operations that completed (array of status) (INOUT). Note that in Fortran a status object is itself an array.

IERROR
The Fortran return code. It is always the last argument.

Notes

In C, the index within the array *array_of_requests*, is indexed from zero and from one in Fortran.

The use of this routine makes the order in which your application completes the requests nondeterministic. An application that processes messages in whatever order they complete must not make assumptions about that order. For example, if:

```
((msgA op msgB) op msgC)
```

can give a different answer than:

```
((msgB op msgC) op msgA)
```

the application must be prepared to accept either answer as *correct*, and must **not** assume a second run of the application will give the same answer.

Errors

A GRequest free function returned an error

A GRequest query function returned an error

Invalid status ignore value

Invalid form of status ignore

Invalid count
count < **0**

Invalid requests

Invalid index array

Truncation occurred

MPI not initialized

MPI already finalized

Develop mode error if:

Illegal buffer update (ISEND)

Inconsistent datatype (MPE_I collectives)

Inconsistent message length (MPE_I collectives)

Inconsistent op (MPE_I collectives)

Match of blocking and non-blocking collectives (MPE_I collectives)

Related information

- MPI_TESTSOME
- MPI_WAIT

MPI_Win_c2f

Translates a C window handle into a Fortran handle to the same window.

C synopsis

```
#include <mpi.h>
MPI_Fint MPI_Win_c2f(MPI_Win win);
```

Description

This function does not have C++ or Fortran bindings. MPI_Win_c2f translates a C window handle into a Fortran handle to the same window; it maps a null handle into a null handle and a non-valid handle into a non-valid handle. The converted handle is returned as the function's value. There is no error detection or return code.

Parameters

win
The window (handle) (IN)

Related information

- MPI_Win_f2c

MPI_WIN_CALL_ERRHANDLER, MPI_Win_call_errhandler

Calls the error handler assigned to the window with the error code supplied.

C synopsis

```
#include <mpi.h>
int MPI_Win_call_errhandler (MPI_Win win, int errorcode);
```

C++ synopsis

```
#include mpi.h
void MPI::Win::Call_errhandler(int errorcode) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_WIN_CALL_ERRHANDLER(INTEGER WIN, INTEGER ERRORCODE, INTEGER IERROR)
```

Description

This subroutine calls the error handler assigned to the window with the error code supplied.

Parameters

win
The window with the error handler (handle) (IN)

errorcode
The error code (integer) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_WIN_CALL_ERRHANDLER returns MPI_SUCCESS in C and C++ and the same value in IERROR if the error handler was successfully called (assuming the error handler itself is not fatal).

The default error handler for MPI_Win is MPI_ERRORS_ARE_FATAL. Thus, calling MPI_WIN_CALL_ERRHANDLER will terminate the job if the default error handler has not been changed for this window. When a predefined error handler is used on *win*, the error message printed by IBM PE MPI will indicate the error code that is passed in. You cannot force IBM PE MPI to issue a specific predefined error by passing its error code to this subroutine.

Error handlers should not be called recursively with MPI_WIN_CALL_ERRHANDLER. Doing this can create a situation where an infinite recursion is created. This can occur if MPI_WIN_CALL_ERRHANDLER is called inside an error handler.

Error codes and classes are associated with a task, so they can be used in any error handler. An error handler should be prepared to deal with any error code it is given. Furthermore, it is good practice to call an error handler only with the appropriate error codes. For example, window errors would normally be sent to the window error handler.

Errors

Invalid error code

Invalid window handle

MPI not initialized

MPI already finalized

Related information

- MPI_ERRHANDLER_FREE
- MPI_WIN_CREATE_ERRHANDLER
- MPI_WIN_GET_ERRHANDLER
- MPI_WIN_SET_ERRHANDLER

MPI_WIN_COMPLETE, MPI_Win_complete

Completes an RMA access epoch on a window object.

C synopsis

```
#include <mpi.h>
int MPI_Win_complete (MPI_Win win);
```

C++ synopsis

```
#include mpi.h
void MPI::Win::Complete() const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_WIN_COMPLETE(INTEGER WIN, INTEGER IERROR)
```

Description

This subroutine completes an RMA access epoch on *win* started by a call to MPI_WIN_START. All RMA communication calls issued on *win* during this epoch will have completed at the origin when the call returns.

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

The target must use corresponding MPI_WIN_POST and MPI_WIN_WAIT. It is the return from MPI_WIN_WAIT at the target that enforces completion at the target.

Parameters

win
The window object (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid window handle (*handle*)

No access epoch to terminate

RMA communication call in progress

RMA synchronization call in progress

MPI not initialized

MPI already finalized

Related information

- MPI_WIN_POST
- MPI_WIN_START
- MPI_WIN_TEST
- MPI_WIN_WAIT

MPI_WIN_CREATE, MPI_Win_create

Allows each task in an intra-communicator group to specify a window in its memory that is made accessible to accesses by remote tasks.

C synopsis

```
#include <mpi.h>
int MPI_Win_create (void *base, MPI_Aint size, int disp_unit,
    MPI_Info info, MPI_Comm comm, MPI_Win *win);
```

C++ synopsis

```
#include mpi.h
static MPI::Win MPI::Win::Create(const void* base, MPI::Aint size,
     int disp_unit, const MPI::Info& info,
     const MPI::Intracomm& comm);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_WIN_CREATE(CHOICE BASE, INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, INTEGER DISP_UNIT,
               INTEGER INFO, INTEGER COMM, INTEGER WIN, INTEGER IERROR)
```

Description

This subroutine returns a handle that represents the window set and the group of tasks that own and access the windows.

This is a collective operation issued by all tasks in the group of *comm*. It creates a window object that can be used by these tasks to perform RMA operations.

Each task specifies a buffer of existing memory that it exposes to RMA accesses by the tasks in the group of *comm*. The buffer consists of *size* number of bytes, starting at address *base*. A task may elect to expose no memory by specifying a *size* value of 0.

The displacement unit argument facilitates address arithmetic in RMA operations. The target displacement argument of an RMA operation is scaled by the factor *disp_unit* specified by the target task, at window creation.

Parameters

base
The initial address of the window (choice) (IN)

size
The size of the window in bytes (nonnegative integer) (IN)

disp_unit
The local unit size for displacements, in bytes (positive integer) (IN)

info
The Info argument (handle) (IN)

comm
The communicator (handle) (IN)

win
The window object returned by the call (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

Common choices for *disp_unit* are: **1** (no scaling), and (in C syntax) *sizeof(type)*, for a window that consists of an array of elements of type *type*. With the latter choice, you can use array indices in RMA calls, and have those scaled correctly to byte displacements, even in a heterogeneous environment.

IBM PE MPI includes support for the *IBM_win_cache* hint, which specifies the amount of memory (in kilobytes) reserved for MPI one-sided RMA communication caching at an origin. Caching occurs whenever several short (in general, those delivering significantly less than 4 KB of data) RMA communications are initiated at the origin against a particular target during a single epoch. If *n* bytes are reserved for this purpose, the resulting *aggregation potential* (maximum number of messages to all targets that may be cached at any given time) is approximately *n*/24 for a 32-bit application and *n*/32 for a 64-bit application. The maximum number of bytes reserved for caching is limited by the number of tasks in the window multiplied by 24000 for a 32-bit application (or for a 64-bit application, multiplied by 32000). Hint values that ask for more bytes than these values are effectively truncated.

A hint in MPI is a (key,value) pair put in an Info object. See "MPI_INFO_CREATE, MPI_Info_create" on page 359. The Info object is then passed to this function, MPI_WIN_CREATE.

The best setting for the *IBM_win_cache* hint is application-dependant. If you know the task never originates more than one RMA per remote task in an epoch, you might prefer to shut off caching. Setting the hint to 0 prevents caching and memory allocation altogether. If you expect the task to originate more than one small RMA per remote task, and can estimate the total number of small RMAs in a typical epoch you can use that estimate as a guide. If there will be *n* small RMAs per epoch, any cache greater than *n**24 (or *n**32 for a 64-bit application) is wasted. If *n* is a large number, such that *n**24 (or *n**32 for a 64-bit application) would require too much memory, the choice of a smaller cache will provide enough aggregation potential to yield most of the possible performance benefit.

If the *IBM_win_cache* hint is not present, 64 KB is reserved.

The various tasks in the group of *comm* may specify completely different target windows, in location, size, displacement units and Info arguments. As long as all the get, put, and accumulate accesses to a particular task fit their specific target window this should not pose a problem. The same area in memory may appear in multiple windows, each associated with a different window object. However, concurrent communications to distinct, overlapping windows may lead to erroneous results.

A window can be created in any part of the task memory. However, on some systems, the performance of windows in memory allocated by MPI_ALLOC_MEM will be better. MPI_ALLOC_MEM has no advantage in IBM PE MPI, but may be used to improve the portability of your code to a system where MPI_ALLOC_MEM does do special memory allocation.

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

In an environment that uses dynamic process management, MP_WIN_CREATE can take an input communicator that covers two or more worlds. The additional connection among these worlds, created by an MPI_WIN_CREATE, is undone by the MPI_WIN_FREE (similar to using MPI_COMM_DISCONNECT on a communicator that spans worlds).

Errors

Can't create RMA window in single threaded environment
MP_SINGLE_THREAD is set to **yes**

Invalid info argument (*handle***)**

Invalid intra-communicator (*handle***)**

Invalid window displacement unit (*value***)**
the *value* of the window displacement unit is less than **1**

Invalid window size (*value***)**
the *value* of the window size is less than **0**

MPI not initialized

MPI already finalized

Related information

- MPI_WIN_FREE
- MPI_WIN_GET_GROUP

MPI_WIN_CREATE_ERRHANDLER, MPI_Win_create_errhandler

Creates an error handler that can be attached to windows.

C synopsis

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

C++ synopsis

```
#include mpi.h
MPI::Errhandler MPI::Win::Create_errhandler(MPI::Win::Errhandler_fn* function);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_WIN_CREATE_ERRHANDLER(EXTERNAL FUNCTION, INTEGER ERRHANDLER,
                          INTEGER IERROR)
```

Description

In C, the user subroutine should be a function of type **MPI_Win_errhandler_fn**, which is defined as:

```
typedef void MPI_Win_errhandler_fn(MPI_Win *, int *, ...);
```

The first argument is the window in use, the second is the error code to be returned.

In C++, the user subroutine should be of the form:

```
typedef void MPI::Win::Errhandler_fn(MPI::Win &, int *, ...);
```

In Fortran, the user subroutine should be of the form:

```
SUBROUTINE WIN_ERRHANDLER_FN(WIN, ERROR_CODE, ...)
INTEGER WIN, ERROR_CODE
```

Parameters

function
The user-defined error-handling procedure (function) (IN)

errhandler
The MPI error handler (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

The MPI standard specifies a **varargs** error handler prototype. A correct user error handler would be coded as:

```
void my_handler(MPI_Win *win, int *errcode, ...){}
```

IBM PE 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. The extra *errhandler* arguments can be accessed by using the C **varargs** (or **stdargs**) facility, but programs that do so will not port cleanly to other MPI implementations that might have different additional arguments.

The effective prototype for an error handler in IBM PE MPI is:

```
typedef void (MPI_Handler_function)
  (MPI_Win *win, int *code, char *routine_name, int *flag,
   MPI_Aint *badval)
```

The additional arguments are:

routine_name
: the name of the MPI routine in which the error occurred

flag **true** if *badval* is meaningful, otherwise **false**

badval
: the non-valid integer or long 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 not allowed

MPI not initialized

MPI already finalized

Related information

- MPI_ERRHANDLER_CREATE
- MPI_WIN_CALL_ERRHANDLER
- MPI_WIN_GET_ERRHANDLER
- MPI_WIN_SET_ERRHANDLER

MPI_WIN_CREATE_KEYVAL, MPI_Win_create_keyval

Generates a new window attribute key.

C synopsis

```
#include <mpi.h>
int MPI_Win_create_keyval (MPI_Win_copy_attr_function *win_copy_attr_fn,
                           MPI_Win_delete_attr_function *win_delete_attr_fn,
                           int *win_keyval, void *extra_state);
```

C++ synopsis

```
#include mpi.h
static int MPI::Win::Create_keyval(MPI::Win::Copy_attr_function* win_copy_attr_fn,
                           MPI::Win::Delete_attr_function* win_delete_attr_fn,
                           void* extra_state);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_WIN_CREATE_KEYVAL(EXTERNAL WIN_COPY_ATTR_FN,
                      EXTERNAL WIN_DELETE_ATTR_FN, INTEGER WIN_KEYVAL,
                      INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, INTEGER IERROR)
```

Description

This subroutine creates a new attribute key for a window and returns a handle to it in the *win_keyval* argument. A key is unique in a task and is opaque to the user. Once created, a key can be used to associate an attribute with a window and access it within the local task.

The argument *win_copy_attr_fn* can be specified as MPI_WIN_NULL_COPY_FN or MPI_WIN_DUP_FN in C, C++, or Fortran. The MPI_WIN_NULL_COPY_FN function returns *flag* = **0** and MPI_SUCCESS. MPI_WIN_DUP_FN is a simple copy function that sets *flag* = **1**, returns the value of *attribute_val_in* in *attribute_val_out*, and returns MPI_SUCCESS.

The argument *win_delete_attr_fn* can be specified as MPI_WIN_NULL_DELETE_FN in C, C++, or Fortran. The MPI_WIN_NULL_DELETE_FN function returns MPI_SUCCESS.

The attribute copy and delete functions are defined as follows (only the C form is shown here):

```
int MPI_Win_copy_attr_fn(MPI_Datatype oldtype, int type_keyval,
                         void *extra_state, void *attribute_val_in,
                         void *attribute_val_out, int *flag)

int MPI_Win_delete_attr_fn(MPI_Datatype type, int type_keyval,
                           void *attribute_val, void *extra_state)
```

The *attribute_val_in* parameter is the value of the attribute. The *attribute_val_out* parameter is the address of the value, so the function can set a new value. The *attribute_val_out* parameter is logically a **void****, but it is prototyped as **void***, to avoid the need for complex casting.

Parameters

extra_state
The extra state for callback functions (integer) (IN)

win_copy_attr_fn
The copy callback function for *win_keyval* (IN)

win_delete_attr_fn
The delete callback function for *win_keyval* (IN)

win_keyval
The key value for future access (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

MPI not initialized

MPI already finalized

Related information

- MPI_KEYVAL_CREATE
- MPI_WIN_FREE_KEYVAL

MPI_WIN_DELETE_ATTR, MPI_Win_delete_attr

Deletes an attribute from a window.

C synopsis

```
#include <mpi.h>
int MPI_Win_delete_attr (MPI_Win win, int win_keyval);
```

C++ synopsis

```
#include mpi.h
void MPI::Win::Delete_attr(int win_keyval);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_WIN_DELETE_ATTR(INTEGER WIN, INTEGER WIN_KEYVAL, INTEGER IERROR)
```

Description

This subroutine deletes an attribute from window *win*.

Parameters

win
The window from which the attribute is deleted (handle) (INOUT)

win_keyval
The key value (integer) (IN)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid window handle (*handle*)

Invalid keyval (*value*)

Invalid use of predefined key (*handle*)

Invalid key type (*value*)

MPI not initialized

MPI already finalized

Related information

- MPI_GET_ATTR
- MPI_SET_ATTR

MPI_Win_f2c

Returns a C handle to a window.

C synopsis

```
#include <mpi.h>
MPI_Win MPI_Win_f2c(MPI_Fint win);
```

Description

This function does not have C++ or Fortran bindings. MPI_Win_f2c returns a C handle to a window. If *win* is a valid Fortran handle to a window, MPI_Win_f2c returns a valid C handle to that same window. If *win* is set to the Fortran value MPI_WIN_NULL, MPI_Win_f2c returns the equivalent null C handle. If *win* is not a valid Fortran handle, MPI_Win_f2c returns a non-valid C handle. The converted handle is returned as the function's value. There is no error detection or return code.

Parameters

win
The window (handle) (IN)

Related information

- MPI_Win_c2f

MPI_WIN_FENCE, MPI_Win_fence

Synchronizes RMA calls on a window.

C synopsis

```
#include <mpi.h>
int MPI_Win_fence (int assert, MPI_Win win);
```

C++ synopsis

```
#include mpi.h
void MPI::Win::Fence(int assert) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_WIN_FENCE(INTEGER ASSERT, INTEGER WIN, INTEGER IERROR)
```

Description

This subroutine, which is collective on the group of *win*, synchronizes RMA calls on window *win*. All RMA operations on *win* originating at a given task and started before the fence call will complete at that task before the fence call returns. They will be completed at their target before the fence call returns at the target. RMA operations on *win* started by a task after the fence call returns will access their target window only after MPI_WIN_FENCE has been called by the target task.

The call completes an RMA access epoch if it was preceded by another fence call and the local task issued RMA communication calls on *win* between these two calls. The call completes an RMA exposure epoch if it was preceded by another fence call and the local window was the target of RMA accesses between these two calls. The call starts an RMA access epoch if it is followed by another fence call and by RMA communication calls issued between these two fence calls. The call starts an exposure epoch if it is followed by another fence call and the local window is the target of RMA accesses between these two fence calls. Thus, the fence call is equivalent to calls to a subset of post, start, complete, and wait.

A fence call usually entails a barrier synchronization: a task completes a call to MPI_WIN_FENCE only after all other tasks in the group entered their matching call. However, a call to MPI_WIN_FENCE that is known not to end any epoch (in particular, a call with *assert* set to **MPI_MODE_NOPRECEDE**) does not necessarily act as a barrier.

Parameters

assert
The program assertion (integer) (IN)

win
The window object (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

Calls to MPI_WIN_FENCE should both precede and follow calls to put, get, or accumulate that are synchronized with fence calls.

The *assert* argument provides assertions on the context of the call that can be used to optimize performance. A value of *assert* set to **0** is always valid. Other valid *assert* values are:

- **MPI_MODE_NOPRECEDE**
- **MPI_MODE_NOPUT**
- **MPI_MODE_NOSTORE**
- **MPI_MODE_NOSUCCEED**

When the *assert* value is set to **MPI_MODE_NOPRECEDE**, the function does not enforce the completion of prior RMA operations. Because the **MPI_MODE_NOPRECEDE** assertion promises that there have been no prior RMA operations, it allows MPI_WIN_FENCE to avoid the cost of confirming that prior RMA operations have completed both locally and remotely. If **MPI_MODE_NOPRECEDE** is used, it must be used on all calls to MPI_WIN_FENCE in the group.

When the *assert* value is set to **MPI_MODE_NOSUCCEED**, the function skips some re-initialization of window state because it can assume that either there is no more RMA and the window will be closed, or some epoch initiating synchronization call will be made before additional RMA operations.

If **MPI_MODE_NOPRECEDE** or **MPI_MODE_NOSUCCEED** is used on only some callers, or if there have been RMA operations prior to a call with **MPI_MODE_NOPRECEDE**, the effect on the application is undefined. Use any assertion with care.

A logical use for these assertions is when an application has a loop containing a load/store epoch and an RMA epoch in every iteration. The first MPI_WIN_FENCE in the loop might assert **MPI_MODE_NOSUCCEED** and be followed by code that does computation reading and updating the window memory. After this computation, another MPI_WIN_FENCE which asserts **MPI_MODE_NOPRECEDE** opens an epoch of RMA operations. When the RMA operations are done, the loop goes back to the top where the MPI_WIN_FENCE with the assertion **MPI_MODE_NOSUCCEED** completes the RMA operations from the prior iteration, and readies another load/store epoch.

An assert value of **MPI_MODE_NOPUT** is a promise the application will not do an MPI_PUT or MPI_ACCUMULATE with the local window as target, until after the next MPI_WIN_FENCE. IBM PE MPI ignores an assert value of **MPI_MODE_NOPUT**, but permits the user to specify this value in order to write applications that are portable to other environments, where this assert is meaningful.

An *assert* value of **MPI_MODE_NOSTORE** is a promise that the application has not attempted to update the local window using local store, MPI_GET, or any form of message receive since the previous MPI_WIN_FENCE. IBM PE MPI ignores an *assert* value of **MPI_MODE_NOPUT**, but permits the user to specify this value in order to write applications that are portable to other environments, where this assert is meaningful.

If an assert is used on a call that does not support that particular assert, the call will raise an error in class **MPI_ERR_ASSERT**. If an assert that is supported for a call is used, but the application structure makes the assert incorrect in the context of this particular call, there will be no error raised at the call and the kinds of failure that a user will experience are not always predictable. Because an assert is a

statement about the structure of your application, a properly chosen assert will be valid for any MPI implementation. An improperly chosen assert may do no harm on one MPI implementation and cause unexplainable failures on another.

Errors

Invalid window handle (*handle*)

Access epoch already in effect

Exposure epoch already in effect

Can't start exposure epoch on a locked target

RMA communication call in progress

RMA synchronization call in progress

MPI not initialized

MPI already finalized

Assertion is not valid for MPI_WIN_FENCE

MPI_WIN_FREE, MPI_Win_free

Frees the window object and returns a null handle (equal to MPI_WIN_NULL).

C synopsis

```
#include <mpi.h>
int MPI_Win_free (MPI_Win *win);
```

C++ synopsis

```
#include mpi.h
void MPI::Win::Free();
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_WIN_FREE(INTEGER WIN, INTEGER IERROR)
```

Description

This is a collective operation issued by all tasks in the group associated with *win*. MPI_WIN_FREE(win) can be invoked by a task only after it has completed its involvement in RMA communication on window *win*. That is, the task has called MPI_WIN_FENCE, or called MPI_WIN_WAIT to match a previous call to MPI_WIN_POST, or called MPI_WIN_COMPLETE to match a previous call to MPI_WIN_START, or called MPI_WIN_UNLOCK to match a previous call to MPI_WIN_LOCK. When the call returns, the window memory can be freed.

Parameters

win
The window object (handle) (INOUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid window handle (*handle*)

Pending origin activity when freeing a window

Pending target activity when freeing a window

RMA communication call in progress

RMA synchronization call in progress

MPI not initialized

MPI already finalized

Related information

- MPI_WIN_CREATE

MPI_WIN_FREE_KEYVAL, MPI_Win_free_keyval

Marks a window attribute key for deallocation.

C synopsis

```
#include <mpi.h>
int MPI_Win_free_keyval (int *win_keyval);
```

C++ synopsis

```
#include mpi.h
void MPI::Win::Free_keyval(int& win_keyval);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_WIN_FREE_KEYVAL(INTEGER WIN_KEYVAL, INTEGER IERROR)
```

Description

This subroutine 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_WIN_DELETE_ATTR to free one attribute instance. To free all attribute instances associated with a communicator, use MPI_WIN_FREE.

Parameters

win_keyval
The key value (integer) (INOUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid use of predefined key (*handle*)

MPI not initialized

MPI already finalized

Related information

- MPI_KEYVAL_CREATE
- MPI_WIN_CREATE_KEYVAL

MPI_WIN_GET_ATTR, MPI_Win_get_attr

Retrieves the window attribute value identified by the key.

C synopsis

```
#include <mpi.h>
int MPI_Win_get_attr (MPI_Win win, int win_keyval,
                      void *attribute_val, int *flag);
```

C++ synopsis

```
#include mpi.h
bool MPI::Win::Get_attr(int win_keyval, void* attribute_val) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_WIN_GET_ATTR(INTEGER WIN, INTEGER WIN_KEYVAL, INTEGER(KIND=MPI_ADDRESS_KIND)
                 ATTRIBUTE_VAL, LOGICAL FLAG, INTEGER IERROR)
```

Description

This subroutine 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**.

Parameters

win
The window to which the attribute is attached (handle) (IN)

win_keyval
The key value (integer) (IN)

attribute_val
The attribute value, unless *flag* = **false** (OUT)

flag
Set to **false** if there is no attribute associated with the key (logical) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

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

In C: As the programmer, you normally define a struct that holds arbitrary attribute information. Before calling MPI_WIN_SET_ATTR, you allocate some storage for the attribute structure and then call MPI_WIN_SET_ATTR to record the address of this structure. You must make sure 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_WIN_GET_ATTR. Both MPI_WIN_SET_ATTR and MPI_WIN_GET_ATTR take a **void*** parameter, but this does not imply that the same parameter is passed to either one.

In Fortran:
MPI_WIN_SET_ATTR records an address-size integer and

MPI_WIN_GET_ATTR returns the address-size integer. As the programmer, you can choose to encode all attribute information in this integer or maintain 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 that will allow some of the same functions a C programmer would use. This is the POINTER type, which is described in the *IBM XL Fortran Compiler Reference*. Use of this feature will impact the program's portability.

Errors

Invalid window handle (*handle***)**

Invalid keyval (*value***)**

Invalid key type (*value***)**

MPI not initialized

MPI already finalized

Related information

- MPI_DELETE_ATTR
- MPI_SET_ATTR

MPI_WIN_GET_ERRHANDLER, MPI_Win_get_errhandler

Retrieves the error handler currently associated with a window.

C synopsis

```
#include <mpi.h>
int MPI_Win_get_errhandler (MPI_Win win, MPI_Errhandler *errhandler);
```

C++ synopsis

```
#include mpi.h
MPI::Errhandler MPI::Win::Get_errhandler() const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_WIN_GET_ERRHANDLER(INTEGER WIN, INTEGER ERRHANDLER, INTEGER IERROR)
```

Description

This subroutine returns the error handler *errhandler* currently associated with window *win*.

Parameters

win
The window (handle) (IN)

errhandler
The error handler currently associated with the window (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid window handle (*handle*)

MPI not initialized

MPI already finalized

Related information

- MPI_WIN_CREATE_ERRHANDLER
- MPI_WIN_SET_ERRHANDLER

MPI_WIN_GET_GROUP, MPI_Win_get_group

Returns a duplicate of the group of the communicator used to create a window.

C synopsis

```
#include <mpi.h>
int MPI_Win_get_group (MPI_Win *win, MPI_Group *group);
```

C++ synopsis

```
#include mpi.h
MPI::Group MPI::Win::Get_group() const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_WIN_GET_GROUP(INTEGER WIN, INTEGER GROUP, INTEGER IERROR)
```

Description

This subroutine returns a duplicate of the group of the communicator used to create the window associated with *win*. The group is returned in *group*. The user is responsible for freeing *group* when it is no longer needed.

It is necessary to know the group associated with a window to be able to create the subset groups needed by MPI_WIN_POST and MPI_WIN_START.

Parameters

win
The window object (handle) (IN)

group
The group of tasks that share access to the window (handle) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid window handle (*handle*)

MPI not initialized

MPI already finalized

Related information

- MPI_GROUP_FREE
- MPI_WIN_CREATE
- MPI_WIN_POST
- MPI_WIN_START

MPI_WIN_GET_NAME, MPI_Win_get_name

Returns the name that was last associated with a window.

C synopsis

```
#include <mpi.h>
int MPI_Win_get_name (MPI_Win win, char *win_name, int *resultlen);
```

C++ synopsis

```
#include mpi.h
void MPI::Win::Get_name(char* win_name, int& resultlen) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_WIN_GET_NAME(INTEGER WIN, CHARACTER*(*) WIN_NAME, INTEGER RESULTLEN,
   INTEGER IERROR)
```

Description

This subroutine returns the last name that was associated with the specified window. The name can be set and retrieved from any language. The same name is returned independent of the language used. The name should be allocated so it can hold a resulting string that is the length of MPI_MAX_OBJECT_NAME. For IBM PE MPI, the value of MPI_MAX_OBJECT_NAME is 256. MPI_WIN_GET_NAME returns a copy of the set name in *win_name*.

Parameters

win
The window with the name to be returned (handle) (IN)

win_name
The name previously stored on the window, or an empty string if no such name exists (string) (OUT)

resultlen
The length of the returned name (integer) (OUT)

IERROR
The Fortran return code. It is always the last argument.

Notes

If you did not associate a name with a window, or if an error occurs, MPI_WIN_GET_NAME returns an empty string (all spaces in Fortran or "" in C and C++).

It is safe simply to print the string returned by MPI_WIN_GET_NAME, as it is always a valid string even if there was no name.

Errors

Fatal errors:

Invalid window handle

MPI already finalized

MPI not initialized

Related information

- MPI_WIN_SET_NAME

MPI_WIN_LOCK, MPI_Win_lock

Starts an RMA access epoch at the target task.

C synopsis

```
#include <mpi.h>
int MPI_Win_lock (int lock_type, int rank, int assert, MPI_Win win);
```

C++ synopsis

```
#include mpi.h
void MPI::Win::Lock(int lock_type, int rank, int assert) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_WIN_LOCK(INTEGER LOCK_TYPE, INTEGER RANK, INTEGER ASSERT,
             INTEGER WIN, INTEGER IERROR)
```

Description

This subroutine starts an RMA access epoch at the target task. Only the window at the task with rank *rank* can be accessed by RMA operations on *win* during that epoch.

Parameters

lock_type
Dictates whether another process can access the target window at the same time (if MPI_LOCK_SHARED) or not (MPI_LOCK_EXCLUSIVE) (IN)

rank
The rank of the locked window (nonnegative integer) (IN)

assert
The program assertion (integer) (IN)

win
The window object (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

The *assert* value on MPI_WIN_LOCK does not affect optimization of IBM PE MPI. A value of *assert* set to **0** is always valid. Other valid *assert* values are:

- **MPI_MODE_NOCHECK**

RMA operations can be started immediately. IBM PE MPI permits the user to specify an *assert* value of **MPI_MODE_NOCHECK**, in order to write applications portable to other environments, where this assert is meaningful.

If an assert is used on a call that does not support that particular assert, the call will raise an error in class **MPI_ERR_ASSERT**. If an assert that is supported for a call is used, but the application structure makes the assert incorrect in the context of this particular call, there will be no error raised at the call and the kinds of failure that a user will experience are not always predictable. Because an assert is a statement about the structure of your application, a properly chosen assert will be valid for any MPI implementation. An improperly chosen assert may do no harm

on one MPI implementation and cause unexplainable failures on another.

Errors

Invalid lock type (*value*)

Invalid window handle (*handle*)

Target outside window group (*rank*)

Access epoch already in effect

RMA communication call in progress

RMA synchronization call in progress

MPI not initialized

MPI already finalized

Assertion is not valid for MPI_WIN_LOCK

Related information

- MPI_WIN_UNLOCK

MPI_WIN_POST, MPI_Win_post

Starts an RMA exposure epoch for a local window.

C synopsis

```
#include <mpi.h>
int MPI_Win_post (MPI_Group group, int assert, MPI_Win win);
```

C++ synopsis

```
#include mpi.h
void MPI::Win::Post(const MPI::Group& group, int assert) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_WIN_POST(INTEGER GROUP, INTEGER ASSERT, INTEGER WIN, INTEGER IERROR)
```

Description

This subroutine starts an RMA exposure epoch for the local window associated with *win*. Only tasks in *group* should access the window with RMA calls on *win* during this epoch. Each task in *group* must issue a matching call to MPI_WIN_START. MPI_WIN_POST does not block. The exposure epoch is closed by a call to MPI_WIN_TEST or MPI_WIN_WAIT.

Parameters

group
The group of target tasks (handle) (IN)

assert
The program assertion (integer) (IN)

win
The window object (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

It is erroneous to have a window locked and exposed (in an exposure epoch) concurrently. That is, a task may not call MPI_WIN_LOCK to lock a target window if the target task has called MPI_WIN_POST and has not yet called MPI_WIN_WAIT. It is erroneous to call MPI_WIN_POST while the local window is locked.

Users need to use explicit synchronization code in order to enforce mutual exclusion between locking periods and exposure epochs on a window.

The use of MPI_WIN_POST and MPI_WIN_WAIT at a task requires that MPI_WIN_POST identify a subset of the tasks in the window group, each of which will do a corresponding MPI_WIN_START.

The *assert* argument provides assertions on the context of the call that can be used to optimize performance. A value of *assert* set to **0** is always valid. Other valid *assert* values are:

- **MPI_MODE_NOCHECK**

- **MPI_MODE_NOPUT**
- **MPI_MODE_NOSTORE**

When the *assert* value is set to **MPI_MODE_NOCHECK**, the function skips the sending of POST notification to the corresponding callers of MPI_WIN_START. If **MPI_MODE_NOCHECK** is used, it must be used on all associated calls to MPI_WIN_POST and MPI_WIN_START. The **MPI_MODE_NOCHECK** assertion is only to be used when the structure of the application code provides an absolute guarantee that the post occurs before any task tries to do an RMA with the MPI_WIN_POST caller as target. If the application structure cannot provide this guarantee, there will be a race condition. Sometime the race will go the wrong way and the application will terminate with a fatal error. The behavior of an application is undefined when it uses an assertion incorrectly.

An *assert* value of **MPI_MODE_NOSTORE** is a promise that the application has not caused the local window to be updated by local store functions (or local get or receive calls) since last synchronization. On some MPI implementations, this assertion might remove the need for a cache synchronization at the post call. IBM PE MPI ignores an *assert* value of **MPI_MODE_NOSTORE**, but permits the user to specify this value in order to write applications that are portable to other environments, where this assert is meaningful.

An *assert* value of **MPI_MODE_NOPUT** is a promise that the application will not cause the local window to be updated by put or accumulate calls after the post call, until the ensuing (wait) synchronization. On some MPI implementations, this assertion might remove the need for a cache synchronization at the wait call. IBM PE MPI ignores an *assert* value of **MPI_MODE_NOPUT**, but permits the user to specify this value in order to write applications that are portable to other environments, where this assert is meaningful.

If an assert is used on a call that does not support that particular assert, the call will raise an error in class **MPI_ERR_ASSERT**. If an assert that is supported for a call is used, but the application structure makes the assert incorrect in the context of this particular call, there will be no error raised at the call and the kinds of failure that a user will experience are not always predictable. Because an assert is a statement about the structure of your application, a properly chosen assert will be valid for any MPI implementation. An improperly chosen assert may do no harm on one MPI implementation and cause unexplainable failures on another.

Errors

Invalid group (*handle*)

Invalid window handle (*handle*)

Group is not a subset of window group

Exposure epoch already in effect

Can't start exposure epoch on a locked target

RMA communication call in progress

RMA synchronization call in progress

MPI not initialized

MPI already finalized

Assertion is not valid for MPI_WIN_POST

Related information

- MPI_WIN_COMPLETE
- MPI_WIN_START
- MPI_WIN_TEST
- MPI_WIN_WAIT

MPI_WIN_SET_ATTR, MPI_Win_set_attr

Attaches the window attribute value to the window and associates it with the key.

C synopsis

```
#include <mpi.h>
int MPI_Win_set_attr (MPI_Win win, int win_keyval, void *attribute_val);
```

C++ synopsis

```
#include mpi.h
void MPI::Win::Set_attr(int win_keyval, const void* attribute_val);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_WIN_SET_ATTR(INTEGER WIN, INTEGER WIN_KEYVAL, INTEGER(KIND=MPI_ADDRESS_KIND)
                 ATTRIBUTE_VAL, INTEGER IERROR)
```

Description

This subroutine stores the attribute value for retrieval by MPI_WIN_GET_ATTR. 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.

Parameters

win
The window to which the attribute will be attached (handle) (INOUT)

win_keyval
The key value (integer) (IN)

attribute_val
The attribute value (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

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

In C: As the programmer, you normally define a struct that holds arbitrary attribute information. Before calling MPI_WIN_SET_ATTR, you allocate some storage for the attribute structure and then call MPI_WIN_SET_ATTR to record the address of this structure. You must make sure 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_WIN_GET_ATTR. Both MPI_WIN_SET_ATTR and MPI_WIN_GET_ATTR take a **void*** parameter, but this does not imply that the same parameter is passed to either one.

In Fortran:
MPI_WIN_SET_ATTR records an address-size integer and MPI_WIN_GET_ATTR returns the address-size integer. As the programmer, you can choose to encode all attribute information in this integer or maintain some kind of database in which the integer can index. Either of these approaches will port to other MPI implementations.

Many of the Fortran compilers include an additional feature that allows some of the same functions a C programmer would use. These compilers support the POINTER type, often referred to as a *Cray pointer*. XL Fortran is one of the compilers that supports the POINTER type. For more information, see *IBM XL Fortran Compiler Reference*.

Errors

Invalid window handle (*handle***)**

Invalid keyval (*value***)**

Invalid use of predefined key (*handle***)**

Invalid key type (*value***)**

MPI not initialized

MPI already finalized

Related information

- MPI_WIN_DELETE_ATTR
- MPI_WIN_GET_ATTR

MPI_WIN_SET_ERRHANDLER, MPI_Win_set_errhandler

Attaches a new error handler to a window.

C synopsis

```
#include <mpi.h>
int MPI_Win_set_errhandler (MPI_Win win, MPI_Errhandler errhandler);
```

C++ synopsis

```
#include mpi.h
void MPI::Win::Set_errhandler(const MPI::Errhandler& errhandler);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_WIN_SET_ERRHANDLER(INTEGER WIN, INTEGER ERRHANDLER, INTEGER IERROR)
```

Description

This subroutine attaches the error handler *errhandler* to window *win*.

Parameters

win
The window (handle) (INOUT)

errhandler
The new error handler for the window (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

The error handler must be either a predefined error handler, or an error handler created by a call to MPI_WIN_CREATE_ERRHANDLER. Any previously-attached error handler is replaced.

For information about a predefined error handler for C++, see *IBM Parallel Environment Runtime Edition: MPI Programming Guide*.

Errors

Invalid error handler

Invalid window handle (*handle*)

MPI not initialized

MPI already finalized

Related information

- MPI_ERRHANDLER_FREE
- MPI_WIN_CREATE_ERRHANDLER
- MPI_WIN_GET_ERRHANDLER

MPI_WIN_SET_NAME, MPI_Win_set_name

Associates a name string with a window.

C synopsis

```
#include <mpi.h>
int MPI_Win_set_name (MPI_Win win, char *win_name);
```

C++ synopsis

```
#include mpi.h
void MPI::Win::Set_name(const char* win_name);
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_WIN_SET_NAME(INTEGER WIN, CHARACTER*(*) WIN_NAME, INTEGER IERROR)
```

Description

This subroutine lets you associate a name string with a window.

The character string that is passed to MPI_WIN_SET_NAME is copied to space managed by the MPI library (so it can be freed by the caller immediately after the call, or allocated on the stack). Leading spaces in the name are significant, but trailing spaces are not.

Parameters

win
The window with the identifier to be set (handle) (INOUT)

win_name
The character string that is saved as the window's name (string) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

MPI_WIN_SET_NAME is a local (noncollective) operation, which affects only the name of the window as specified in the task that made the MPI_WIN_SET_NAME call. There is no requirement that the same (or any) name be assigned to a window in every task where that window exists. However, to avoid confusion, it is a good idea to give the same name to a window in all of the tasks where it exists.

The length of the name that can be stored is limited to the value of MPI_MAX_OBJECT_NAME in Fortran and MPI_MAX_OBJECT_NAME-1 in C and C++ to allow for the null terminator. Attempts to use a longer name will result in truncation of the name. For IBM PE MPI, the value of MPI_MAX_OBJECT_NAME is 256.

Associating a name with a window has no effect on the semantics of an MPI program, and (necessarily) increases the storage requirement of the program, because the names must be saved. Therefore, there is no requirement that you use this subroutine to associate names with windows. However, debugging and profiling MPI applications can be made easier if names are associated with data types, as the debugger or profiler should then be able to present information in a less cryptic manner.

Errors

Fatal errors:

Invalid window handle

MPI already finalized

MPI not initialized

Related information

- MPI_WIN_GET_NAME

MPI_WIN_START, MPI_Win_start

Starts an RMA access epoch for a window object.

C synopsis

```
#include <mpi.h>
int MPI_Win_start (MPI_Group group, int assert, MPI_Win win);
```

C++ synopsis

```
#include mpi.h
void MPI::Win::Start(const MPI::Group& group, int assert) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_WIN_START(INTEGER GROUP, INTEGER ASSERT, INTEGER WIN, INTEGER IERROR)
```

Description

This subroutine starts an RMA access epoch for *win*. RMA calls issued on *win* during this epoch must access only windows at tasks in *group*. Each task in *group* must issue a matching call to MPI_WIN_POST. RMA accesses to each target window are delayed, if necessary, until the target task issues the matching call to MPI_WIN_POST. MPI_WIN_START is allowed to block until the corresponding MPI_WIN_POST calls are processed, but is not required to.

Parameters

group
The group of target tasks (handle) (IN)

assert
The program assertion (integer) (IN)

win
The window object (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Notes

The use of MPI_WIN_START and MPI_WIN_COMPLETE at a task requires that MPI_WIN_START identify a subset of the tasks in the window group, each of which will do a corresponding MPI_WIN_POST.

The *assert* argument provides assertions on the context of the call that can be used to optimize performance. A value of *assert* set to **0** is always valid. Other valid *assert* values are:

- **MPI_MODE_NOCHECK**

When the *assert* value is set to **MPI_MODE_NOCHECK**, the RMA functions skip waiting for a POST notification from the target and simply assume the target has called MPI_WIN_POST. If **MPI_MODE_NOCHECK** is used, it must be used on all associated calls to MPI_WIN_POST and MPI_WIN_START. The **MPI_MODE_NOCHECK** assertion is only to be used when the structure of the application code provides an absolute guarantee that the post occurs before any task tries to do an RMA with the MPI_WIN_POST caller as target. If the application structure cannot provide this guarantee, there will be a race condition.

Sometime the race will go the wrong way, and the application will terminate with a fatal error. The behavior of an application is undefined when it uses an assertion incorrectly.

If an assert is used on a call that does not support that particular assert, the call will raise an error in class **MPI_ERR_ASSERT**. If an assert that is supported for a call is used, but the application structure makes the assert incorrect in the context of this particular call, there will be no error raised at the call and the kinds of failure that a user will experience are not always predictable. Because an assert is a statement about the structure of your application, a properly chosen assert will be valid for any MPI implementation. An improperly chosen assert may do no harm on one MPI implementation and cause unexplainable failures on another.

Errors

Invalid group (*handle***)**

Invalid window handle (*handle***)**

Group is not a subset of window group (*handle***)**

Access epoch already in effect

RMA communication call in progress

RMA synchronization call in progress

MPI not initialized

MPI already finalized

Assertion is not valid for MPI_WIN_START

Related information

- MPI_WIN_COMPLETE
- MPI_WIN_POST
- MPI_WIN_TEST
- MPI_WIN_WAIT

MPI_WIN_TEST, MPI_Win_test

Tries to complete an RMA exposure epoch.

C synopsis

```
#include <mpi.h>
int MPI_Win_test (MPI_Win win, int *flag);
```

C++ synopsis

```
#include mpi.h
bool MPI::Win::Test() const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_WIN_TEST(INTEGER WIN, LOGICAL FLAG, INTEGER IERROR)
```

Description

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

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

Parameters

flag
The success flag (logical) (OUT)

win
The window object (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid window handle (*handle*)

No exposure epoch to terminate

Unsolicited access of local window while exposed

RMA communication call in progress

RMA synchronization call in progress

MPI not initialized

MPI already finalized

Related information

- MPI_WIN_COMPLETE
- MPI_WIN_POST
- MPI_WIN_START
- MPI_WIN_WAIT

MPI_WIN_UNLOCK, MPI_Win_unlock

Completes an RMA access epoch at the target task.

C synopsis

```
#include <mpi.h>
int MPI_Win_unlock (int rank, MPI_Win win);
```

C++ synopsis

```
#include mpi.h
void MPI::Win::Unlock(int rank) const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_WIN_UNLOCK(INTEGER RANK, INTEGER WIN, INTEGER IERROR)
```

Description

This subroutine completes an RMA access epoch started by a call to MPI_WIN_LOCK(...,win). RMA operations issued during this period will have completed both at the origin and at the target when the call returns.

Parameters

rank
The rank of the window (nonnegative integer) (IN)

win
The window object (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid window handle (*handle*)

Target outside window group (*rank*)

Origin holds no lock on the target (*rank*)

No access epoch to terminate

Unsolicited access of target window while locked

RMA communication call in progress

RMA synchronization call in progress

MPI not initialized

MPI already finalized

Related information

- MPI_WIN_LOCK

MPI_WIN_WAIT, MPI_Win_wait

Completes an RMA exposure epoch.

C synopsis

```
#include <mpi.h>
int MPI_Win_wait (MPI_Win win);
```

C++ synopsis

```
#include mpi.h
void MPI::Win::Wait() const;
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
MPI_WIN_WAIT(INTEGER WIN, INTEGER IERROR)
```

Description

This subroutine completes an RMA exposure epoch started by a call to MPI_WIN_POST on *win*. MPI_WIN_WAIT matches calls to MPI_WIN_COMPLETE(win) issued by each of the origin tasks that were granted access to the window during this epoch. The call to MPI_WIN_WAIT will block until all matching calls to MPI_WIN_COMPLETE have occurred. This guarantees that all these origin tasks have completed their RMA accesses to the local window. When the call returns, all these RMA accesses will have completed at the target window.

Parameters

win
The window object (handle) (IN)

IERROR
The Fortran return code. It is always the last argument.

Errors

Invalid window handle (*handle*)

No exposure epoch to terminate

Unsolicited access of local window while exposed

RMA communication call in progress

RMA synchronization call in progress

MPI not initialized

MPI already finalized

Related information

- MPI_WIN_COMPLETE
- MPI_WIN_POST
- MPI_WIN_START
- MPI_WIN_TEST

MPI_WTICK, MPI_Wtick

Returns the resolution of MPI_WTIME in seconds.

C synopsis

```
#include <mpi.h>
double MPI_Wtick(void);
```

C++ synopsis

```
#include mpi.h
double MPI::Wtick();
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
DOUBLE PRECISION MPI_WTICK()
```

Description

This subroutine 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

Returns the current value of *time* as a floating-point value.

C synopsis

```
#include <mpi.h>
double MPI_Wtime(void);
```

C++ synopsis

```
#include mpi.h
double MPI::Wtime();
```

Fortran synopsis

```
include 'mpif.h' or USE MPI
DOUBLE PRECISION MPI_WTIME()
```

Description

This subroutine 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, MPI_Attr_get" on page 77 for more information.

If you are using IBM PE for AIX, the environment variable **MP_CLOCK_SOURCE** lets you specify the source from which MPI_WTIME gets its time values. See *IBM Parallel Environment Runtime Edition: MPI Programming Guide* for more information.

Errors

MPI not initialized

MPI already finalized

Related information

- MPI_ATTR_GET
- MPI_WTICK
- MPI_COMM_GET_ATTR

Chapter 4. Parallel utility subroutines

There are a number of user-callable, thread-safe subroutines that take advantage of the parallel operating environment (POE). There is a C version and a Fortran version for most of the subroutines.

Included are subroutines for:

- Controlling distribution of STDIN and STDOUT.
- Synchronizing parallel tasks without using the message passing library.
- Improving control of interrupt driven programs.
- Printing and clearing statistical data.
- Controlling the checkpoint/restart function (IBM PE for AIX only).

For descriptions of these subroutines, see *IBM Parallel Environment Runtime Edition: MPI Programming Guide*.

Note: MPICH2 does not support the parallel utility subroutines.

The parallel utility subroutines are:

mpc_isatty
Determines if a device is a terminal on the home node.

MP_BANDWIDTH, mpc_bandwidth (IBM PE for AIX only)
Obtains user space switch bandwidth statistics. Note that **MP_BANDWIDTH** can only be used with applications that use MPI. LAPI programs that used an earlier version of this routine should be recoded to make the relevant calls directly. For more information, see the *IBM Parallel Environment Runtime Edition: MPI Programming Guide*.

MP_DISABLEINTR, mpc_disableintr (IBM PE for AIX only)
Disables message arrival interrupts for the MPI task from which it was called.

MP_ENABLEINTR, mpc_enableintr (IBM PE for AIX only)
Enables message arrival interrupts for the MPI task from which it was called.

MP_FLUSH, mpc_flush
Flushes output buffers to STDOUT. This is a synchronizing call across all parallel tasks.

MP_INIT_CKPT, mpc_init_ckpt (IBM PE for AIX only)
Starts user-initiated checkpointing.

MP_QUERYINTR, mpc_queryintr
Returns the state of interrupts on a task.

MP_SET_CKPT_CALLBACKS, mpc_set_ckpt_callbacks (IBM PE for AIX only)
Registers functions to be called when an application is checkpointed, resumed, and restarted.

MP_STATISTICS_WRITE, mpc_statistics_write
Prints both MPCI and LAPI transmission statistics.

MP_STATISTICS_ZERO, mpc_statistics_zero
Resets (zeros) the **MPCI_stats_t** structure. It has no effect on LAPI.

MP_STDOUT_MODE, mpc_stdout_mode
Sets the mode (single, ordered, unordered) for STDOUT.

MP_STDOUTMODE_QUERY, mpc_stdoutmode_query
Returns the mode to which STDOUT is currently set.

MP_UNSET_CKPT_CALLBACKS, mpc_unset_ckpt_callbacks (IBM PE for AIX only) Unregisters checkpoint, resume, and restart application callbacks.

pe_dbg_breakpoint (IBM PE for AIX only)
Provides a communication mechanism between POE and an attached third party debugger (TPD).

Chapter 5. Parallel task identification API subroutines (IBM PE for AIX only)

There are a number of parallel task identification API subroutines that are available for parallel programming. These subroutines take advantage of the parallel operating environment (POE).

The POE API subroutines are:

poe_master_tasks
: Retrieves the list of process IDs of master POE processes currently running on this system.

poe_task_info
: Returns a NULL-terminated array of pointers to structures of type POE_TASKINFO.

For descriptions of these subroutines, see *IBM Parallel Environment Runtime Edition: MPI Programming Guide*.

Accessibility features for IBM PE Runtime Edition

Accessibility features help users who have a disability, such as restricted mobility or limited vision, to use information technology products successfully.

This product uses standard Microsoft Windows navigation keys.

Accessibility features

The following list includes the major accessibility features in Parallel Environment Runtime Edition:

- Keyboard-only operation
- Interfaces that are commonly used by screen readers
- Keys that are discernible by touch but do not activate just by touching them
- Industry-standard devices for ports and connectors
- The attachment of alternative input and output devices

The *IBM Cluster information center*, and its related publications, are accessibility-enabled. The accessibility features of the information center are described in the IBM Cluster information center (http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/topic/com.ibm.cluster.addinfo.doc/access.html).

IBM and accessibility

See the IBM Human Ability and Accessibility Center (http://www.ibm.com/able) for more information about the commitment that IBM has to accessibility.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
2455 South Road,
Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample

programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

All implemented function in the IBM PE MPI product is designed to comply with the requirements of the Message Passing Interface Forum, MPI: A Message-Passing Interface Standard. The standard is documented in two volumes, 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. The second volume includes a section identified as MPI 1.2 with clarifications and limited enhancements to MPI 1.1. It also contains the extensions identified as MPI 2.0. The three sections, MPI 1.1, MPI 1.2 and MPI 2.0 taken together constitute the current standard for MPI.

IBM PE MPI provides full support for all of MPI 2.2.

If you believe that IBM PE MPI does not comply with the MPI standard for the portions that are implemented, please contact IBM Service.

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml.

Intel is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Glossary

This glossary defines technical terms used in the IBM Parallel Environment documentation. If you do not find the term you are looking for, refer to the IBM Terminology site on the World Wide Web (**http://www.ibm.com/software/globalization/terminology/index.html**).

A

address
A unique code or identifier for a register, device, workstation, system, or storage location.

API application programming interface (API): An interface that allows an application program that is written in a high-level language to use specific data or functions of the operating system or another program.

application
One or more computer programs or software components that provide a function in direct support of a specific business process or processes.

argument
A value passed to or returned from a function or procedure at run time.

authentication
The process of validating the identity of a user or server.

authorization
The process of obtaining permission to perform specific actions.

B

bandwidth
A measure of frequency range, typically measured in hertz. Bandwidth also is commonly used to refer to data transmission rates as measured in bits or bytes per second.

blocking operation
An operation that has not completed 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
The simultaneous transmission of data to more than one destination.

C

C A programming language designed by Bell Labs in 1972 for use as the systems language for the UNIX operating system.

C++ An enhancement of the C language that adds features supporting object-oriented programming.

client A software program or computer that requests services from a server.

cluster
A group of processors interconnected through a high-speed network that can be used for high-performance computing.

collective communication
A communication operation that involves more than two processes or tasks. Broadcasts and reductions are examples of collective communication operations. All tasks in a communicator must participate.

communicator
A Message Passing Interface (MPI) object that describes the communication context and an associated group of processes.

compile
translate all or part of a program expressed in a high-level language into a computer program expressed in an intermediate language, an assembly language, or a machine language.

condition
One of a set of specified values that a data item can assume.

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 a severe user error. A programmer can use the core dump to diagnose and correct the problem.

core file
A file that preserves the state of a program, usually just before a program is terminated because of an unexpected error. See also *core dump.*

D

data parallelism
A situation in which parallel tasks perform the same computation on different sets of data.

debugger
A tool used to detect and trace errors in computer programs.

E

environment variable
(1) A variable that defines an aspect of the operating environment for a process. For example, environment variables can define the home directory, the command search path, the terminal in use, or 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.

Ethernet
A packet-based networking technology for local area networks (LANs) that supports multiple access and handles contention by using Carrier Sense Multiple Access with Collision Detection (CSMA/CD) as the access method. Ethernet is standardized in the IEEE 802.3 specification.

executable program
A program that can be run as a self-contained procedure. It consists of a main program and, optionally, one or more subprograms.

execution
The process of carrying out an instruction or instructions of a computer program by a computer.

F

fairness
A policy in which tasks, threads, or processes must eventually gain access to a resource for which they are competing. For example, if multiple threads are simultaneously seeking a lock, no set of circumstances can cause any thread to wait indefinitely for access to the lock.

Fiber Distributed Data Interface (FDDI)
An American National Standards Institute (ANSI) standard for a 100-Mbps LAN using fiber optic cables.

file 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, and updates correct an error in, or enhance, a previously installed program. (2) One or more separately-installable, logically-grouped units in an installation package. See also *licensed program* and *package.*

FORTRAN
A high-level programming language used primarily for scientific, engineering, and mathematical applications.

G

GDB An open-source portable debugger supporting Ada, C, C++, and FORTRAN. GDB is a useful tool for determining why a program crashes and where, in the program, the problem occurs.

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 symbol defined in one program module that is used in other program modules that are independently compiled.

graphical user interface (GUI)
A type of computer interface that presents a visual metaphor of a real-world scene, often of a desktop, by combining high-resolution graphics, pointing devices,

menu bars and other menus, overlapping windows, icons and the object-action relationship.

GUI See graphical user interface.

H

high performance switch
A high-performance message-passing network that connects all processor nodes.

home node
The node from which an application developer compiles and runs a program. The home node can be any workstation on the LAN.

host A computer that is connected to a network and provides an access point to that network. The host can be a client, a server, or both a client and server simultaneously.

host list file
A file that contains a list of host names, and possibly other information. The host list file is defined by the application that reads it.

host name
The name used to uniquely identify any computer on a network.

I

installation image
A copy of the software, in backup format, that the user is installing, as well as copies of other files the system needs to install the software product.

Internet
The collection of worldwide networks and gateways that function as a single, cooperative virtual network.

Internet Protocol (IP)
A protocol that routes data through a network or interconnected networks. This protocol acts as an intermediary between the higher protocol layers and the physical network.

IP Internet Protocol.

K

kernel The part of an operating system that contains programs for such tasks as input/output, management and control of hardware, and the scheduling of user tasks.

L

latency
The time from the initiation of an operation until something actually starts happening (for example, data transmission begins).

licensed program
A separately priced program and its associated materials that bear a copyright and are offered to customers under the terms and conditions of a licensing agreement.

lightweight core files
An alternative to standard AIX core files. Core files produced in the Standardized Lightweight Corefile Format provide simple process stack traces (listings of function calls that led to the error) and consume fewer system resources than traditional core files.

LoadLeveler® pool
A group of resources with similar characteristics and attributes.

local variable
A symbol defined in one program module or procedure that can only be used within that program module or procedure.

M

management domain
A set of nodes that are configured for management by Cluster Systems Management. Such a domain has a management server that is used to administer a number of managed nodes. Only management servers have knowledge of the domain. Managed nodes only know about the servers managing them.

menu A displayed list of items from which a user can make a selection.

message catalog
An indexed table of messages. Two or more catalogs can contain the same index values. The index value in each table refers to a different language version of the same message.

message passing
The process by which parallel tasks explicitly exchange program data.

Message Passing Interface (MPI)
A library specification for message passing. MPI is a standard application programming interface (API) that can be used by parallel applications.

MIMD
multiple instruction stream, multiple data stream.

multiple instruction stream, multiple data stream (MIMD)
A parallel programming model in which different processors perform different instructions on different sets of data.

MPMD
Multiple program, multiple data.

Multiple program, multiple data (MPMD)
A parallel programming model in which different, but related, programs are run on different sets of data.

N

network
In data communication, a configuration in which two or more locations are physically connected for the purpose of exchanging data.

network information services (NIS)
A set of network services (for example, a distributed service for retrieving information about the users, groups, network addresses, and gateways in a network) that resolve naming and addressing differences among computers in a network.

NIS See *network information services*.

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, that returns immediately whether or not the operation has completed. For example, a nonblocking receive does not wait until a message arrives. A nonblocking receive must be completed by a later test or wait.

O

object code
Machine-executable instructions, usually generated by a compiler from source code written in a higher level language. Object code might itself be executable or it might require linking with other object code files.

optimization
The process of achieving improved run-time performance or reduced code size of an application. Optimization can be performed by a compiler, by a preprocessor, or through hand tuning of 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
1) In AIX, a number of filesets that have been collected into a single installable image of licensed programs. See also fileset and licensed program. 2) In Linux, a collection of files, usually used to install a piece of software. The equivalent AIX term is fileset.

parallelism
The degree to which parts of a program may be concurrently executed.

parallelize
To convert a serial program for parallel execution.

parameter
A value or reference passed to a function, command, or program that serves as input or controls actions. The value is supplied by a user or by another program or process.

peer domain
A set of nodes configured for high availability. Such a domain has no distinguished or master node. All nodes are aware of all other nodes, and administrative commands can be issued from any node in the domain. All nodes also have a consistent view of the domain membership. Contrast with *management domain*.

point-to-point communication
A communication operation that 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
In a programming language, a block, with or without formal parameters, that is initiated 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. A process 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 with 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.

profiling
A performance analysis process that is based on statistics for the resources that are used by a program or application.

pthread
A shortened name for the i5/OS™ threads API set that is based on a subset of the POSIX standard.

R

reduction operation
An operation, usually mathematical, that reduces a collection of data by one or more dimensions. For example, an operation that reduces an array to a scalar value.

remote host
Any host on a network except the host at which a particular operator is working.

remote shell (rsh)
A variant of the remote login (rlogin) command that invokes a command interpreter on a remote UNIX machine and passes the command-line arguments to the command interpreter, omitting the login step completely.

S

Secure Shell (SSH)
A network protocol for secure data exchange between two networked devices. The client can use public-key and private-key authentication, or password authentication, to access the remote server.

shell script
A program, or script, that is interpreted by the shell of an operating system.

segmentation fault
A system-detected error, usually caused by a reference to a memory address that is not valid.

server A software program or a computer that provides services to other software programs or other computers.

single program, multiple data (SPMD)
A parallel programming model in which different processors run the same program on different sets of data.

source code
A computer program in a format that is readable by people. Source code is converted into binary code that can be used by a computer.

source line
A line of source code.

SPMD
single program, multiple data.

standard error (STDERR)
The output stream to which error messages or diagnostic messages are sent.

standard input (STDIN)
An input stream from which data is

retrieved. Standard input is normally associated with the keyboard, but if redirection or piping is used, the standard input can be a file or the output from a command.

standard output (STDOUT)
The output stream to which data is directed. Standard output is normally associated with the console, but if redirection or piping is used, the standard output can be a file or the input to a command.

STDERR
standard error.

STDIN
standard input.

STDOUT
standard output.

subroutine
A sequence of instructions within a larger program that performs a particular task. A subroutine can be accessed repeatedly, can be used in more than one program, and can be called at more than one point in a program.

synchronization
The action of forcing certain points in the execution sequences of two or more asynchronous procedures to coincide in time.

system administrator
The person who controls and manages a computer system.

T

task In a parallel job, there are two or more concurrent tasks working together through message passing. Though it is common to allocate one task per processor, the terms *task* and *processor* are not interchangeable.

thread A stream of computer instructions. In some operating systems, a thread is the smallest unit of operation in a process. Several threads can run concurrently, performing different jobs.

trace A record of the processing of a computer program or transaction. The information collected from a trace can be used to assess problems and performance.

U

user (1) An individual who uses license-enabled software products. (2) Any individual, organization, process, device, program, protocol, or system that uses the services of a computing system.

User Space
A version of the message passing library that is optimized for direct access to the high performance switch (IBM PE for AIX) or communication adapter (IBM PE for Linux). User Space maximizes performance by not involving the kernel in sending or receiving a message.

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
A representation of a changeable value.

X

X Window System
A software system, developed by the Massachusetts Institute of Technology, that enables the user of a display to concurrently use multiple application programs through different windows of the display. The application programs can execute on different computers.

Index

E

F

G

I

M

N

P

S

T

W

Product Number: 5641-PR1
5641-PR2
5725-G00

Printed in USA

SC23-6784-07