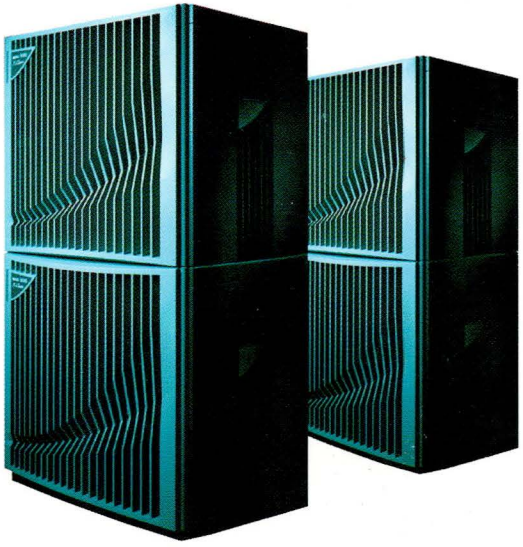
HEWLETT® PACKARD



HP MPI User's Guide

First Edition

Hewlett-Packard Company
Convex Division
3000 Waterview Parkway
P.O. Box 833851
Richardson, TX 75083-3851
United States of America

HP MPI User's Guide

B6011-90001

First Edition

January 1997

Hewlett-Packard Company
Convex Division
Richardson, Texas
United States of America

HP MPI User's Guide

B6011-90001

Parts of this book came from Cornell Theory Center's web document. That document is copyrighted by the Cornell Theory Center.

Parts of this book came from *MPI: A Message Passing Interface*. That book is copyrighted by the University of Tennessee. These sections were copied by permission of the University of Tennessee.

Parts of this book came from *MPI Primer / Developing with LAM*. That document is copyrighted by the Ohio Supercomputer Center. These sections were copied by permission of the Ohio Supercomputer Center.

Notice

The information contained in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

This entire book is recyclable.

Printed in the United States of America

Revision Information for

HP MPI User's Guide

Edition	HP Part No.No.	Description
First	B6011-90001	Released with HP MPI V1.1, January 1997. This edition supersedes 770-011230-002, released with HP MPI V1.1, October, 1996.

Contents

Figures

Tables

How to use this guide

This guide describes the HP MPI implementation of the Message Passing Interface (MPI) standard. The guide will help you use HP MPI to develop parallel applications.

You should already have experience developing UNIX applications. You should also understand the basic concepts behind parallel processing and be familiar with MPI.

This guide is intended to supplement *MPI: The Complete Reference* (B6011-90003) with information specific to the HP implementation of MPI.

Note

Some of the sections in this book contain examples used to illustrate HP MPI concepts. These examples use the /bin/csh syntax.

HP MPI features

HP MPI provides a wide range of features that offer you flexibility in developing parallel applications. These features include:

- Single program multiple data (SPMD) and multiple program multiple data (MPMD) styles of programming—Allow you to create an application that consists of a single program that is executed by each process or two or more programs where each process can execute a different program. In both cases, processes normally act on different data.
- XMPI tracing utility—Allows you to monitor trace files (one per process) during application computation and communication.
- Multilevel parallelism—Supports MPI processes that call multithreaded libraries to perform computations. This approach is beneficial for problems that can be decomposed into logical parts for parallel execution.
- Multiprotocol support —Supports diferent communication protocols depending upon where the processes are located and what type of platform is used. The supported protocols include shared memory within a host and TCP/IP between hosts.

System platforms

HP MPI runs under the SPP-UX and HP-UX operating systems.

The SPP-UX operating system is used on:

- SPP1600 servers
- S-Class servers
- X-Class servers

The HP-UX operating system is used on:

- K-Class servers
- D-Class servers

Using this book

Use Table 1 to select which portions of the book to read.

Table 1 Book organization

Read ...	To learn more about ...
Introduction	Message passing and basic MPI concepts such as latency and bandwidth, sending and receiving messages, and point-to-point and collective communication.
Getting started	Configuring your environment and running an application.
Understanding HP MPI	Directory structure, compiling applications, and running applications.
Analyzing MPI applications	Using the MPI profiling interface and the XMPI utility to profile, debug, and trace your applications. The CXpa, CXdb, and CXtrace tools are also briefly described.
Tuning application performance	Tuning considerations such as message latency and bandwidth, computation to communication ratio, CPU subscription, MPI routine selection, multilevel parallelism, and process placement.
Troubleshooting applications	Troubleshooting HP MPI errors that occur during building, starting, running, and completing applications.
MPI library routines	Interface information for the MPI routines and the HP MPI library extensions.
Example programs	Example programs in C, C++, Fortran 77, and Fortran 90 that supplement the description of MPI concepts.
XMPI resource file	Resource settings that determine how XMPI displays on your workstation.

Notational conventions

Notational conventions used in this book are described below.

`bold monospace`

In command examples, text shown in bold monospace identifies input that must be typed exactly as shown.

`monospace`

In paragraph text, `monospace` identifies command names, system calls, and data structures and types. In command examples, `monospace` identifies command output, including error messages.

Italic

In paragraph text, *italic* identifies titles of documents.

In command syntax diagrams, *italic* identifies variables that you must provide. The following command example uses brackets to indicate that the variable *output_file* is optional:

`command` *input_file* [*output_file*]

Note

A note highlights important supplemental information.

Associated documents

Associated documents include:

- *MPI: The Complete Reference*, published by MIT Press (B6011-90003)
- *Exemplar Programming Guide* (B5600-90001)
- *SPP-UX System Administration Guide* (B5655-90002)
- *HP PVM User's Guide* (B5885-90001)
- *HP PVM User's Guide for K-Class and EPS Servers* (B4774-90002)

Use Table 2 to access World Wide Web sites that contain additional MPI information.

Table 2 Useful MPI World Wide Web sites

Access ...	To learn more about ...
http://www.mcs.anl.gov/Projects/mpi/index.html	Argonne National Laboratory's MPICH implementation of MPI.
http://www.tc.cornell.edu/Edu/Tutor/MPI/	Cornell Theory Center's MPI tutorial and lab exercises.
http://www.osc.edu/lam.html	Ohio Supercomputer Center's LAM implementation of MPI.
http://www.erc.msstate.edu/mpi/	Mississippi State University's MPI web page.

Ordering documents

To order additional copies of this document or other documents listed in "Associated documents," send requests to:

Hewlett-Packard Company
Convex Division
Customer Service
P.O. Box 833851
Richardson TX 75083-3851 USA

Please include the order number (xxxxx-9xxxx number) or the exact title of the document.

Technical assistance

If you have questions that are not answered in this book, contact the Hewlett-Packard Convex Technical Assistance Center (TAC) at the following locations:

- Within the continental U.S., call 1 (800) 952-0379.
- From Canada, call 1 (800) 345-2384.
- All other locations, contact your local Hewlett-Packard office.

You can also use the contact utility, if you would like to report any problems you may have with HP MPI or its associated documentation.

Credits

HP MPI V1.1 is based on MPICH from Argonne National Laboratory and Mississippi State University and LAM from Ohio Supercomputing Center. The XMPI utility is based on LAM's version, available at http://www.osc.edu/lam.html.

Introduction 1

This chapter provides introductory information about MPI. The topics covered include:

- Parallel computational models
- Message passing
- MPI concepts

Parallel computational models

Computational models represent a way to look at the types of operations that are available to parallel applications.

These models are independent of the hardware used to support them. They can be implemented on any machine designed to run parallel applications. Model performance, however, depends on how optimally a model is implemented on a particular hardware architecture.

The models are categorized by how memory is used (shared versus distributed), what the unit of execution is, and how communication occurs (software versus hardware).

The models include:

- Shared memory—Each process can access a shared address space.
- Message passing—An application runs as a collection of autonomous processes, each with its own local memory.
- Remote memory operations—A process accesses the memory of another without its participation, but it does so explicitly, not the same way it accesses its local memory.
- Threads—In a multi-threaded process, the values of application variables are shared by all the threads.
- Hybrid models—Two or more of the models are used together.

Message passing

In message passing, a parallel application consists of a number of processes that run concurrently. Each process has its own local memory and communicates with other processes by sending and receiving messages. When data is passed in a message, both processes must work to transfer the data from the local memory of one to the local memory of the other.

Message passing is currently one of the most popular computation models for designing parallel applications. The advantages of using message passing include:

- Portability—MPI is implemented on most parallel platforms.
- Universality—Model makes minimal assumptions about underlying parallel hardware. Message-passing libraries exist on computers linked by networks and shared and distributed multiprocessors.
- Simplicity—Model supports explicit control of memory references for easier debugging.

On the other hand, creating message-passing applications may require more effort than letting a parallelizing compiler produce parallel applications.

Applications based on the message-passing model are nondeterministic by default (that is, the arrival order of messages sent from two processes to a third process is not defined). However, when one process sends two or more messages to another process, the transfer is deterministic in the sense that the messages are always received in the order sent. You must ensure that process communication and computation are deterministic when required within your application.

MPI concepts

MPI is a standard specification for interfaces to a library of message-passing routines. The goals of MPI are efficient communication, portability, and rich functionality.

Although several message-passing libraries exist on different systems (for example, Intel Paragon's NX, Connection Machine's CMMD, or PVM), MPI has emerged as a favorite for the following reasons:

- Support for full asynchronous communication—Process communication can overlap process computation.
- Group membership—Processes may be grouped based on context.
- Synchronization variables that protect process messaging—When sending and receiving messages, synchronization is enforced by source and destination information, message labeling, and context information.
- Portability—All implementations are based on a published standard that specifies the semantics for usage.

The MPI library contains 128 routines. These routines provide a rich set of functions that support point-to-point communications, collective operations, process groups and communication contexts, process topologies, and data type manipulation. HP MPI adds a small number of extension routines to this set. See "MPI library routines and extensions" on page 65 for the interface description of each routine.

Although the MPI library contains a large number of routines to choose from, you can design a surprisingly large number of applications by using only the six listed in Table 3.

Table 3 Six commonly used MPI routines

MPI routine	Description
`MPI_Init`	Initializes the MPI environment
`MPI_Finalize`	Terminates the MPI environment
`MPI_Comm_rank`	Determines the rank of the calling process within a group
`MPI_Comm_size`	Determines the size of the group
`MPI_Send`	Sends messages
`MPI_Recv`	Receives messages

As your application grows in complexity, you can introduce other routines from the library. For example, `MPI_Bcast` is an often-used routine for sending data from one process to other processes in a single operation. This is much more efficient in terms of performance than using `MPI_Send` to transfer data from the sending process to each receiving process one by one.

Point-to-point communication

Point-to-point communication involves sending and receiving messages between two processes. This is the simplest form of data transfer in a message-passing model.

The performance of point-to-point communication is measured in terms of total transfer time. The total transfer time is defined as

total_transfer_time = *latency* + (*message_size*/*bandwidth*)

Latency is the time between the initiation of the data transfer in the sending process and the arrival of the first byte in the receiving process. Bandwidth is the reciprocal of the time needed to transfer a byte. Bandwidth is normally expressed in megabytes per second.

Obviously, low latencies and high bandwidths lead to better performance. In some cases, you can improve performance more by optimizing the placement of the sending and receiving processes.

Communicators

A communicator is an object that represents a group of processes and their communication medium or context. These processes exchange messages with each other to transfer data. In this context, communicators encapsulate their processes such that communication is restricted to processes only within the group.

Communicators are also used to synchronize communications. In this context, they prevent process messages from being mistakenly intercepted by other program layers (for example, a user-defined library that the application calls).

The default communicators provided by MPI are MPI_COMM_WORLD and MPI_COMM_SELF. MPI_COMM_WORLD consists of all processes that are running when an application begins execution. Each process is the single member of its own MPI_COMM_SELF.

Communicators that allow processes within a single group to exchange data are termed intracommunicators. Communicators that allow processes between two different groups to exchange data are called intercommunicators.

Many MPI applications depend upon knowing how many processes there are and the process rank within a given communicator.

To determine the number of processes in a communicator named `comm`, use

```
MPI_Comm_size (MPI_Comm comm, int *size);
```

To determine the rank of each process in `comm`, use

```
MPI_Comm_rank (MPI_Comm comm, int *rank);
```

where the return value of rank is an integer between zero and (size - 1).

Refer to example "communicator.c" on page 105 for more information about using communicators.

Sending and receiving messages

There are two methods for sending and receiving data: blocking and nonblocking.

Blocking communication means the sending process does not return until the send buffer is available for reuse.

Nonblocking communication means the sending process returns immediately, but the send buffer is not safe for reuse. In this case:

1. The sending routine begins message transfer and returns immediately.
2. The application performs some computation.
3. The application calls a completion routine (for example, `MPI_Test` or `MPI_Wait`) to test or wait for completion of the send operation.

Blocking communication

Blocking communication consists of four send modes and one receive mode. The same receive mode is used regardless of the send mode used during process communication.

The four send modes include:

- Standard mode (`MPI_Send`)—The sending process returns when the system can buffer the message or when the message is received.
- Buffered mode (`MPI_Bsend`)—The sending process returns when the message is buffered in application-supplied space.
- Synchronous mode (`MPI_Ssend`)—The sending process returns only if a matching receive is posted and the receiving process has started to receive the message.
- Ready mode (`MPI_Rsend`)—The sending process assumes that a matching receive is posted. The sending process returns after the message is sent.

The four send modes are all invoked in a similar manner and all pass the same arguments. The only difference is in the routine name used to send the message (that is, `MPI_Send` versus `MPI_Bsend`).

To code a standard blocking send, use

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

where

`buf`

Specifies the starting address of the buffer.

`count`

Indicates the number of buffer elements.

dtype

Denotes the data type of the buffer elements.

dest

Specifies the rank of the destination process in the group associated with the communicator comm.

tag

Denotes the message label.

comm

Communication context that identifies a group of processes.

To code a blocking receive, use

```
MPI_Recv (void *buf, int count,
          MPI_datatype dtype, int source,
          int tag, MPI_Comm comm,
          MPI_Status *status);
```

where

buf

Specifies the starting address of the buffer.

count

Indicates the number of buffer elements.

dtype

Denotes the data type of the buffer elements.

dest

Specifies the rank of the destination process in the group associated with the communicator comm.

tag

Denotes the message label.

comm

Communication context that identifies a group of processes.

source

Specifies the rank of the source process in the group associated with the communicator comm.

status

Used to return information about the received message. It is useful when wildcards are used or the received message is smaller than expected. It may also contain error codes.

Refer to examples "send_receive.f" on page 90, "ping_pong.c" on page 92, and "master_worker.f90" on page 98 for more information about using blocking communications.

Nonblocking communication

MPI provides nonblocking versions of the blocking send and receive calls. Table 4 lists these calls.

Table 4 MPI blocking and nonblocking calls

Blocking mode	Nonblocking mode
MPI_Send	MPI_Isend
MPI_Bsend	MPI_Ibsend
MPI_Ssend	MPI_Issend
MPI_Rsend	MPI_Irsend
MPI_Recv	MPI_Irecv

The nonblocking calls have the same arguments as their blocking counterparts plus an additional argument for a request.

To code a standard nonblocking send, use

```
MPI_Isend (void *buf, int count,
           MPI_datatype dtype, int dest,
           int tag, MPI_Comm comm,
           MPI_Request *req);
```

where `req` specifies the request used by a completion routine when called by the application to complete the send operation.

Collective operations

Applications may require coordinated operations among multiple processes. For example, all processes need to cooperate in order to multiply two matrices or sum sets of numbers distributed among them.

MPI provides a set of collective operations to coordinate operations among processes. These operations are implemented such that all processes call the same operation with the same arguments. Thus, when sending and receiving messages, one collective operation can replace multiple sends and receives, resulting in lower overhead and higher performance.

Collective operations consist of routines for communication, computation, and synchronization. These routines all specify a communicator argument that defines the group of participating processes and the context of the operation.

Note

Collective operations are valid only for intracommunicators. Intercommunicators are not allowed as arguments.

Communication

Collective communication involves the exchange of data among all processes in a group. The communication can be one-to-many, many-to-one, or many-to-many.

The single originating or receiving process in the one-to-many and many-to-one routines is called the root.

Examples of such communication routines are:

Broadcast (`MPI_Bcast`)

A one-to-many operation where the root sends its data to all other processes in the communicator, including itself. Figure 1 shows the broadcast operation for a four-process application. Each row of boxes represents data locations in one process.

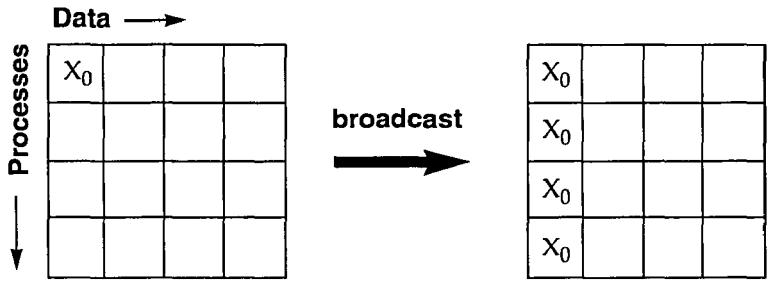


Figure 1 MPI broadcast operation

Scatter (`MPI_Scatter`)

A one-to-many operation where the root's data is split among all processes in the communicator. Figure 2 shows the scatter operation for a four-process application. As before, each row of boxes represents data locations in one process.

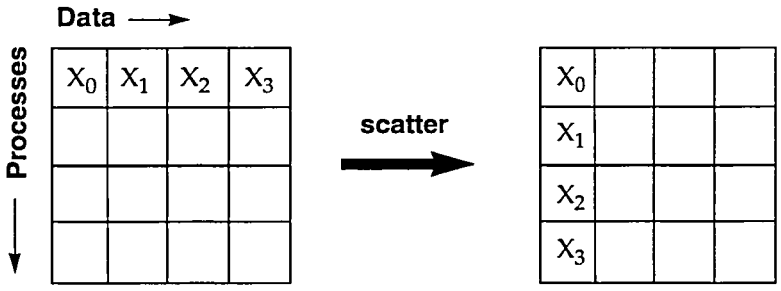


Figure 2 MPI scatter operation

To code a broadcast, use

```
MPI_Bcast (void *buf, int count,
           MPI_Datatype dtype, int root,
           MPI_Comm comm);
```

where

`buf`

Specifies the starting address of the buffer.

`count`

Indicates the number of buffer entries.

`dtype`

Denotes the data type of the buffer entries.

`root`

Specifies the rank of the root.

`comm`

Communication context that identifies a group of processes.

To code a scatter, use

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

where

`sendbuf`

Specifies the starting address of the send buffer.

`sendcount`

Specifies the number of elements sent to each process.

sendtype

Denotes the data type of the send buffer.

recvbuf

Specifies the address of the receive buffer.

recvcount

Indicates the number of elements in the receive buffer.

recvtype

Indicates the data type of the receive buffer elements.

root

Denotes the rank of the sending process.

comm

Communication context that identifies a group of processes.

Computation

Computation uses MPI_Reduce to apply reduction operations across all processes in a communicator. Reduction operations are binary and are only valid on numeric data. Also, reductions are always associative but may or may not be commutative.

You can select a reduction operation from a predefined list or define your own operation. Examples of predefined operations include MPI_SUM and MPI_PROD, which apply a summation and a multiplication across all processes respectively.

To implement a reduction, use

```
MPI_Reduce (void *sendbuf, void *recvbuf,
            int count, MPI_Datatype dtype,
            MPI_Op op, int root, MPI_Comm comm);
```

where

sendbuf

Specifies the address of the send buffer.

recvbuf

Denotes the address of the receive buffer.

count

Indicates the number of elements in the send buffer.

dtype

Specifies the data type of the send and receive buffers.

op

Specifies the reduction operation.

`root`

Indicates the rank of the root process.

`comm`

Communication context that identifies a group of processes.

Synchronization

Collective routines return as soon as their participation in a communication is complete. However, the return of the calling process does not guarantee that the receiving processes have completed or even started the operation.

To synchronize the execution of processes, call `MPI_Barrier`. `MPI_Barrier` blocks the calling process until all processes in the communicator have called it. This is a useful approach for separating two stages of a computation so messages from each stage are not overlapped.

To implement a barrier, use

```
MPI_Barrier (MPI_Comm comm);
```

where `comm` identifies a group of processes and a communication context.

Refer to examples "compute_pi.f" on page 96 and "sort.C" on page 100 for more information about using collective operations.

Noncontiguous data

`MPI_Send` and `MPI_Recv` are useful for transferring data between two processes using point-to-point communication. This transfer is based on the assumption that the data transferred is stored in contiguous memory (for example, sending an array in a C application).

What happens, however, when you want to transfer data that is not stored contiguously? In this case, you can create a derived data type or use `MPI_Pack` and `MPI_Unpack`.

A derived data type specifies a sequence of basic data types and integer displacements describing the data layout in memory.

You create derived data types through the use of type-constructor functions. Once a derived data type is created, it can be used repeatedly in all communicating calls. This allows MPI to pack and unpack the data as necessary and further optimize the data transfer.

The main types of constructor functions include:

- Contiguous—Allows replication of a data type into contiguous locations
- Vector—Allows replication of a data type into locations that consist of equally spaced blocks

- Indexed—Allows replication of a data type into a sequence of blocks where each block can contain a different number of copies and have a different displacement.
- Structure—Allows replication of a data type into a sequence of blocks such that each block consists of replications of different data types, copies, and displacements.

To create a vector data type, use

```
MPI_Type_Vector (int count, int blocklength,
                 int stride,
                 MPI_Datatype oldtype,
                 MPI_Datatype *newtype);
```

where

`count`

Indicates the number of blocks.

`blocklength`

Specifies the number of elements in each block.

`stride`

Denotes the number of elements between the start of two consecutive blocks.

`oldtype`

Specifies the old data type.

`newtype`

Specifies the new data type.

You must now commit the derived data type you created by calling `MPI_Type_commit`.

`MPI_Pack` allows you to store noncontiguous data in contiguous memory locations. `MPI_Unpack` copies data from a contiguous buffer into noncontiguous memory locations. Used together and when appropriate, these routines allow you to transfer heterogeneous data in a single message.

To code a pack, use

```
MPI_Pack (void *inbuf, int incount,
          MPI_Datatype dtype, void *outbuf,
          int outsize, int *position,
          MPI_Comm, comm);
```

where

`inbuf`

Specifies the start of the input buffer.

`incount`

Indicates the number of input data items.

`dtype`

Denotes the data type of each input data item.

`outbuf`

Specifies the start of the output buffer.

`outsize`

Indicates the output buffer size in bytes.

`position`

Specifies the current position in the buffer in bytes.

`comm`

Communication context that identifies a group of processes.

Derived data types are more efficient than using `MPI_Pack` and `MPI_Unpack`, but they cannot handle the case where the data layout varies and is not known before running the application.

Multilevel parallelism

By default, processes in MPI applications can only perform one task at a time. Such processes are known as single-threaded processes. This means that each process has an address space together with a single program counter, set of registers, and stack.

Multithreaded processes have one address space, but each process thread contains its own counter, registers, and stack.

Multilevel parallelism refers to MPI processes that call multithreaded libraries to perform computations. This approach is beneficial for problems that can be decomposed into logical parts for parallel execution (for example, a looping construct that spawns multiple threads to perform a computation and joins after the computation is complete).

Refer to example "multi_par.c" on page 106 for more information about using multilevel parallelism.

Advanced topics

This chapter provides information only about basic MPI concepts. Other advanced MPI topics include:

- Error handling
- Process topologies
- User-defined data types
- Process grouping
- Attribute caching

To learn more about these advanced topics, see *MPI: The Complete Reference*, the companion to this guide.

Getting started

2

This chapter describes how to get started with HP MPI. The topics covered are:

- Configuring your environment
- Building and running your first application

Configuring your environment

Use the following steps to configure your environment before running your first HP MPI application:

Step 1 Verify that HP MPI is installed on your system in the /opt/mpi directory.

Step 2 Add /opt/mpi/bin to the PATH variable by entering:

```
% setenv PATH /opt/mpi/bin:$PATH
```

Step 3 Add /opt/mpi/share/man to the MANPATH variable by entering:

```
% setenv MANPATH /opt/mpi/share/man:$MANPATH
```

Building and running your first application

To quickly gain experience with HP MPI, start by working with a C version of the familiar hello-world program. This program is called hello_world.c and prints out the text string "Hello world! I'm *r* of *s* on *host*" where *r* is a process's rank, *s* is the size of the communicator, and *host* is the host on which the program is run.

The source code for hello_world.c is stored in /opt/mpi/help and is shown below.

```
/* hello_world.c */
#include <stdio.h>
#include <mpi.h>

main(argc, argv)

int                     argc;
char                    *argv[];

{
        int             rank, size, len;
        char            name[MPI_MAX_PROCESSOR_NAME];

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        MPI_Get_processor_name(name, &len);
        printf ("Hello world! I'm %d of %d on %s\n", rank, size, name);

        MPI_Finalize();
        exit(0);
}
```

Building and running on a single host

To build and run hello_world.c on a local host named jawbone:

Step 1 Change to a writable directory.

Step 2 Enter

```
% mpicc -o hello_world /opt/mpi/help/hello_world.c
```

This step builds the hello_world executable.

Step 3 Enter

```
% mpirun -j -np 4 hello_world
```

This step runs the hello_world executable using four processes.

hello_world output

The output from running the hello_world executable is printed in nondeterministic order and is shown below.

```
Hello world! I'm 1 of 4 on jawbone
Hello world! I'm 3 of 4 on jawbone
Hello world! I'm 0 of 4 on jawbone
Hello world! I'm 2 of 4 on jawbone
```

Building and running on multiple hosts

To build and run hello_world.c on a local host named jawbone and a remote host named wizard (assuming that both machines run under either SPP-UX or HP-UX or hello_world.c is built on HP-UX so the same binary can run on both hosts):

Step 1 Change to a writable directory.

Step 2 Enter

```
% mpicc -o hello_world /opt/mpi/help/hello_world.c
```

This step builds the hello_world executable.

Step 3 Copy the hello_world executable from jawbone to your home directory on wizard.

Step 4 Create a text file called appfile and add the following two lines:

```
-np 2 hello_world
-h wizard -np 2 hello_world
```

The appfile file contains a separate line for each host, which specifies the name of the executable and the number of processes to run on that host. The `-h` option identifies remote hosts.

Step 5 Enter

```
% mpirun -f appfile
```

This step runs hello_world on the hosts specified in the appfile file.

hello_world output

The output from running the hello_world executable is printed in nondeterministic order and is shown below.

```
Hello world! I'm 1 of 4 on wizard
Hello world! I'm 3 of 4 on jawbone
Hello world! I'm 0 of 4 on wizard
Hello world! I'm 2 of 4 on jawbone
```

Understanding HP MPI

3

This chapter provides information about the HP MPI implementation of MPI 1.1. The topics covered are:

- Directory structure
- Compiling applications
- Running applications

Directory structure

All HP MPI files are stored in the /opt/mpi directory. The directory structure is organized as shown in Table 5.

Table 5 Organization of the /opt/mpi directory

Subdirectory	Contents
bin	Binary files for the HP MPI utilities
help	Source files for the example programs shipped with HP MPI
include	Header files
lib/X11/app-defaults	Application default settings for the XMPI trace utility
lib/pa1.1/libfmpi.a	MPI library for Fortran applications
lib/pa1.1/libmpi.a	MPI library for C and C++ applications
lib/pa1.1/libpmpi.a	MPI profiling interface library
newconfig/	Configuration files and release notes
share/man/man1.Z	Man pages for the HP MPI utilities
share/man/man3.Z	Man pages for HP MPI library extensions

The man pages located in the /opt/mpi/share/man/man1.Z subdirectory can be grouped into three categories: compilation, general, and runtime. The compilation and run-time categories correspond to available types of HP MPI utilities. All three categories are described in Table 6.

Table 6 Man page categories

Man page category	Description
Compilation	Describes the available compilation utilities. Refer to Compiling applications for more information.
General	Describes the general features of HP MPI. The man page is called MPI.1.
Runtime	Describes the available run-time utilities. Refer to "Run-time utility commands" on page 30 for more information.

Compiling applications

The compiler used to build HP MPI applications depends upon which programming language you use. HP MPI provides separate compilation utilities and default compilers for the languages shown in Table 7.

Table 7 Compilation utilities

Language	Utility	Default compiler
C	`mpicc`	/opt/ansic/bin/cc
C++	`mpiCC`	/opt/CC/bin/CC
Fortran 77	`mpif77`	/opt/fortran/bin/f77
Fortran 90	`mpif90`	/opt/fortran90/bin/f90

Note

Even though the mpiCC and mpif90 compilation utilities are shipped with HP MPI, all C++ and Fortran 90 applications currently use C and Fortran 77 bindings respectively.

If you want to use a compiler other than the default one assigned to each utility, you can set the environment variables shown in Table 8.

Table 8 Compilation environment variables

Utility	Environment variable
mpicc	MPI_CC
mpiCC	MPI_CXX
mpif77	MPI_F77
mpif90	MPI_F90

To set an environment variable, enter

% **setenv** ***environment_variable path***

where *environment_variable* is the name of the variable you want to set and *path* specifies the path to the compiler you want to use.

If you ever move the HP MPI installation directory from its default location (/opt/mpi), set the MPI_ROOT environment variable to point to the new location. To set MPI_ROOT, enter

```
% setenv MPI_ROOT /usr/local/mpi
```

Running applications

Most HP MPI applications are run using the mpirun command. You should invoke the mpirun command with the -j option. This option displays the job ID of your job. The job ID is useful during troubleshooting if you want to check for a hung job using the mpijob command or want to terminate your job using the mpiclean command.

In some cases, you can use the *executable* -np # command syntax to start your application. For example, to start an executable named hello_world with four processes, enter:

```
% hello_world -np 4
```

For multiprotocol applications that span multiple subcomplexes or multiple hosts, you must use mpirun together with an appfile. For applications that run on a single host and have a single executable, you can use *executable* -np #, although mpirun is still recommended.

Types of applications

HP MPI supports two programming styles: SPMD applications and MPMD applications.

Running SPMD applications

A SPMD application consists of a single program that is executed by each process in the application. Each process normally acts upon different data. Even though this style simplifies the execution of an application, using SPMD can also make the executable larger and more complicated.

Each process calls `MPI_Comm_rank` to distinguish itself from all other processes in the application. It then determines what processing to perform.

To run a SPMD application, use the `mpirun` command like this:

```
% mpirun -j -np # hello_world
```

where # is the number of processors and hello_world is the name of your application.

Suppose you want to build a C application called poisson and run it using five processes to perform the computation. To do this, use the following command sequence:

```
% mpicc -o poisson poisson.c
% mpirun -j -np 5 poisson
```

Running MPMD applications

A MPMD application uses two or more separate programs to functionally decompose a problem.

This style can be used to simplify the application source and reduce the size of spawned processes. Each process can execute a different program.

To run an MPMD application, the `mpirun` command must reference an appfile that contains the number of processes to be created from each program and the list of programs to be run.

A simple invocation of an MPMD application looks like this:

```
% mpirun -j -f appfile
```

where *appfile* is the path name to a file that contains process counts and a list of programs.

Suppose you decompose the poisson application into two source files: poisson_master (uses a single master process) and poisson_child (uses four child processes).

The appfile for the example application contains the two lines shown below:

```
-np 1 poisson_master
-np 4 poisson_child
```

To build and run the example application, use the following command sequence:

% `mpicc -o poisson_master poisson_master.c`

% `mpicc -o poisson_child poisson_child.c`

% `mpirun -j -f` *appfile*

See "Creating an appfile" on page 31 for more information about using appfiles.

Multiprotocol messaging

Multiprotocol messaging refers to process communication that uses different protocols depending upon where the processes are located and what type of Exemplar system is used.

An example configuration for an X-Class server is shown in Figure 3.

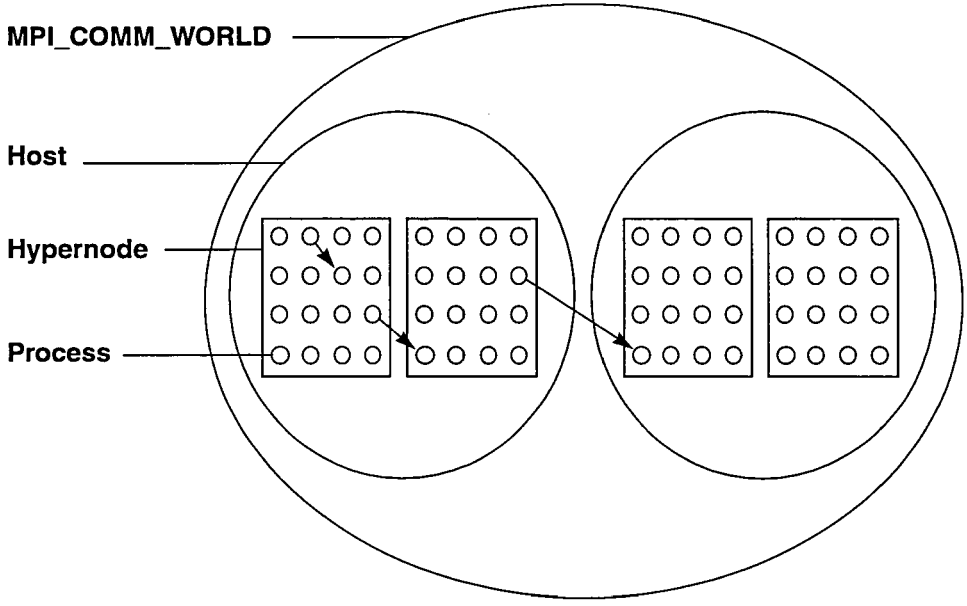


Figure 3 Multiprotocol messaging with an X-Class server

The circles within each hypernode represent processes. The arrows represent message passing. An arrow originates from the sending process and terminates at the receiving process.

Point-to-point and collective protocols on an X-Class server support messaging between:

- Processes on the same hypernode—Data is transferred directly using an optimized bcopy operation.
- Processes on different hypernodes in the same host—Data is transferred using global shared memory.
- Processes on different hosts—Data is transferred using a TCP/IP socket.

The communication speed of protocols for S- and X-Class servers is fastest for processes on the same hypernode, slower for processes on different hypernodes in the same host, and slowest for processes on different hosts.

An example configuration for a K-Class server is shown in Figure 4.

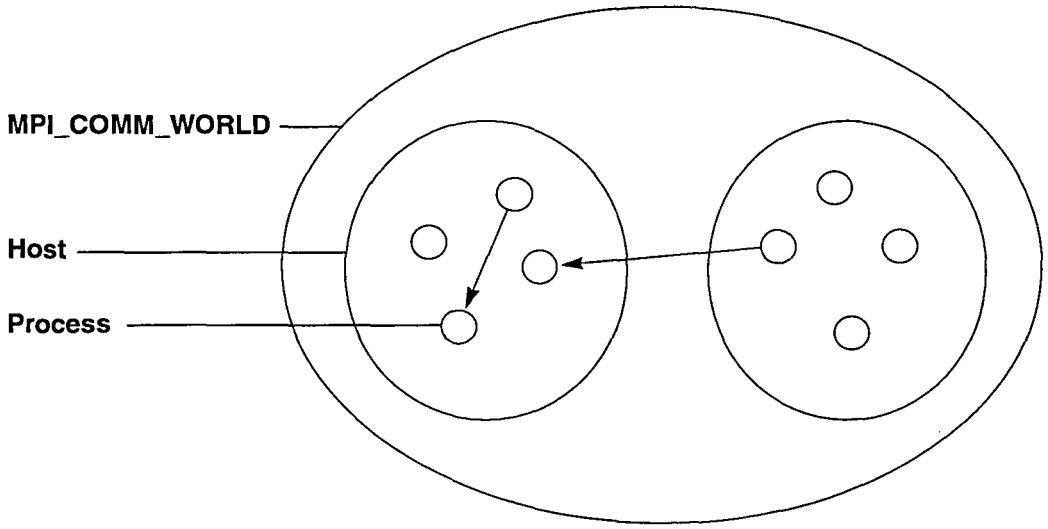


Figure 4 Multiprotocol messaging with a K-Class server

The circles within each host represent processes. The arrows represent message passing. An arrow originates from the sending process and terminates at the receiving process.

Point-to-point and collective protocols on D- and K-Class servers support messaging between:

- Processes on the same host—Data is transferred using global shared memory.
- Processes on different hosts—Data is transferred using TCP/IP sockets.

Run-time environment variables

Environment variables are used to alter the way HP MPI executes an application. The variable settings determine how an application behaves and how an application allocates internal resources at runtime.

Many applications run without setting any environment variables. However, applications that use a large number of nonblocking messaging requests, require debugging support, or need to control process placement may need a more customized configuration.

Environment variables are always local to the system where `mpirun` is running (that is, their values are not propagated to remote hosts).

The environment variables listed below affect the behavior of HP MPI at runtime:

- `MPI_FLAGS`
- `MPI_GLOBMEMSIZE`
- `MPI_TOPOLOGY`
- `MPI_SHMEMCNTL`
- `MPI_TMPDIR`
- `MPI_XMPI`

MPI_FLAGS

`MPI_FLAGS` modifies the general behavior of HP MPI. The `MPI_FLAG` syntax is shown below:

```
MPI_FLAG [d,][ecxdb,][edde,][exdb,][j,][s[a|p]
[#],][v]
```

where

`d`

Displays library debug information.

`ecxdb`

Starts a separate CXdb session for each process. This option is provided for backward compatibility on S- and X-Class servers only. The debugger must be in the command search path. For more information about debugging using CXdb, see CXdb online help or the CXdb man pages.

`edde`

Starts the application under the DDE debugger. The debugger must be in the command search path. This option is supported on D- and K-Class servers only.

exdb

Starts the application under the xdb debugger. The debugger must be in the command search path. This option is supported on D- and K-Class servers only.

j

Prints the HP MPI job identifier.

s[a|p][#]

Selects signal and maximum time-delay for guaranteed message progression. The sa option selects SIGALRM. The sp option selects SIGPROF. The # option is the number of seconds to wait before issuing a signal to trigger message progression. The default value of this flag is sp600, which issues a SIGPROF every ten minutes.

Note

The SIGPROF option is not supported on S- and X-Class servers when your application executable is in Extended Standard Object Module format.

This mechanism is used to guarantee message progression in applications that use nonblocking messaging requests followed by prolonged periods of time in which HP MPI routines are not called.

v

Prints the version number.

MPI_GLOBMEMSIZE

MPI_GLOBMEMSIZE specifies the amount of shared memory allocated for all processes in an HP MPI application. MPI_GLOBMEMSIZE has the following syntax:

```
MPI_GLOBMEMSIZE amount
```

where amount specifies the amount of shared memory in bytes.

By default, each process allocates 2 Mbytes of shared memory. Be sure that the value specified for MPI_GLOBMEMSIZE is less than the amount of global shared memory allocated for the subcomplex when working with X-Class servers.

MPI_TOPOLOGY

MPI_TOPOLOGY controls application process placement within a subcomplex on SPP1600 multinode, S-, and X-Class servers. MPI_TOPOLOGY has the following syntax:

```
MPI_TOPOLOGY [[sc]/[hypernode]:][topology]
```

where

sc

Identifies the name of a subcomplex.

hypernode

Specifies the logical hypernode within the subcomplex on which to start the first process. By default, the initial hypernode is chosen by the operating system.

topology

Is a comma-separated list that specifies the number of processes to start on each logical hypernode in the subcomplex, beginning with logical hypernode 0.

HP MPI uses logical hypernode numbering. The operating system handles the mapping from physical to logical hypernodes. This mapping follows the lowest-to-highest sorted order of physical hypernode numbers. For example, in a 2-node subcomplex using physical hypernodes 3 and 4, physical hypernode 3 would map to logical hypernode 0, and physical hypernode 4 would map to logical hypernode 1.

An MPI_TOPOLOGY value of System/3:4,0,4,4 specifies that logical hypernodes zero, two, and three of the subcomplex System each run four processes. The first application process is started on logical hypernode three.

When running a multinode application where some processes run different executables, MPI_TOPOLOGY settings in the appfile override any settings you might have specified by setting MPI_TOPOLOGY from the command line. See "Creating an appfile" on page 31 for more information.

The number of processes specified using MPI_TOPOLOGY must match the number of processes specified in mpirun. For example, if you set MPI_TOPOLOGY to 2,3 and invoke mpirun with -np 6, the system will generate an error message and terminate your job.

Also, be sure that the number of hypernodes specified in MPI_TOPOLOGY matches the number of available hypernodes on the subcomplex you want to use. For example, if you set MPI_TOPOLOGY to 6,2,3 and System only contains hypernodes 0 and 1, the system will generate an error message and terminate your job. To prevent this problem, use the scm utility to determine the configuration of system subcomplexes before invoking mpirun.

Note

The default subcomplex on all systems is called System. Use the mpa utility to set the default to another subcomplex.

MPI_SHMEMCNTL

MPI_SHMEMCNTL controls the subdivision of each process's shared memory for the purposes of point-to-point and collective communications. MPI_SHMEMCNTL syntax is shown below:

```
MPI_SHMEMCNTL [nenv,][frag,][small,][large]
```

where

`nenv`

Specifies the number of envelopes per process pair.

`frag`

Denotes the size in bytes of the message-passing fragments region.

`small`

Specifies the size in bytes of the small, generic-shared memory region.

`large`

Denotes the size in bytes of the large, generic-shared memory region. The generic regions are used by some collective routines.

MPI_TMPDIR

By default, HP MPI uses the /tmp directory to store temporary files needed for its operations. `MPI_TMPDIR` is used to point to a different temporary directory. `MPI_TMPDIR` syntax is shown below:

`MPI_TMPDIR` *directory*

where *directory* specifies an existing directory used to store temporary files.

MPI_XMPI

`MPI_XMPI` specifies options for run-time raw trace generation. These options represent an alternate way to set tracing rather than using the trace options supplied with `mpirun`.

The argument list for `MPI_XMPI` contains the prefix name for the file where each process writes its own raw trace data. Each process creates its own filename by concatenating the prefix, a period, and the process's global rank number.

For example, if a process has rank 0 and the prefix is hello_world, the process's raw trace file would be hello_world.0. If the file prefix name does not begin with a forward slash (/) (for example, /tmp/test), the raw trace file is stored in the directory in which the process is executing `MPI_Init`.

`MPI_XMPI` syntax is shown below:

`MPI_XMPI` *prefix* `[:bs###] [:nc] [:off] [:s]`

where

prefix

Specifies the file prefix name.

`bs###`

Denotes the buffering size in kbytes for dumping raw trace data. Actual buffering size may be rounded up by the system. The default buffering size is 4096 kbytes. Specifying a large buffering size reduces the need to flush raw trace data to a file when process buffers reach capacity. Flushing too frequently can cause communication routines to run slower. If this problem occurs, increase the buffering size.

`nc`

Specifies no clobber, which means that an HP MPI application aborts if a file with the name specified in *prefix* already exists.

`off`

Denotes that trace generation is initially turned off and only begins after all processes collectively call `MPIHP_Trace_on`.

`s`

Specifies a simpler tracing mode by omitting `MPI_Test`, `MPI_Testall`, `MPI_Testany`, and `MPI_Testsome` calls that do not complete a request. This option may reduce the size of trace data so that `xmpi` runs faster.

Note

Even though you can specify tracing options through the MPI_XMPI environment variable, the recommended approach is to use the mpirun command instead. Using mpirun to specify tracing options guarantees that multihost applications perform tracing in a consistent manner. See "mpirun" on page 30 for more information.

Run-time utility commands

HP MPI provides a set of utility commands to supplement the MPI library routines. These commands include:

- `mpirun`
- `mpiclean`
- `mpijob`
- `xmpi`
- `mpitrget`

mpirun

`mpirun` starts an HP MPI application.

`mpirun` syntax has two forms:

- `mpirun [-np #] [-help] [-version] [-djpvW] [-t` *spec*`] [-h` *host*`] [-l` *user*`] [-e` *var*`[=`*val*`] [...]] [-sp` *paths*`]` *program* `[`*args*`]`
- `mpirun [-help] [-version] [-djpvW] [-t` *spec*`] [-f` *appfile*`]`

where

`-np #`

Starts # copies of the program.

`-help`

Prints usage information for the utility.

`-version`

Prints the version information.

`-d`

Turns on debug mode to debug the executable.

`-j`

Prints the HP MPI job ID.

`-p`

Turns on pretend mode. That is, go through the motions of starting an HP MPI application but do not create any processes. This is useful for debugging and checking whether the appfile (if used) is setup correctly.

`-v`

Turns on verbose mode.

`-W`

Does not wait for the application to terminate before returning.

`-t` *spec*

Enables run-time raw trace generation for all processes. *spec* specifies options used when tracing. See "MPI_XMPI" on page 28 for more information about tracing options.

`-h` *host*

Starts the processes on *host* (default is localhost).

`-l` *user*

Specifies the user name on the target host (default is local username).

args

Specifies command line arguments to the program.

`-e` *var*`[=`*val*`]`

Sets the environment variable *var* for the program and gives it the value *val* if provided. Environment variable substitutions (for example, $FOO) are supported in the *val* argument.

`-sp` *paths*

Sets the target shell PATH environment variable to *paths*. Search paths are separated by the colon (:) character.

program

Specifies the name of the executable to run.

`-f` *appfile*

Starts the application described in *appfile*.

The first syntax is used for applications where all processes execute the same program on the same host. For example,

```
% mpirun -j -np 3 send_receive
```

runs the send_receive application with three processes and prints out the job ID.

The second syntax must be used for applications that consist of multiple programs or that run on multiple hosts or subcomplexes. In this case, each program called by the application is listed in a file called an appfile.

Creating an appfile

The format of entries in an appfile is line oriented. Lines that end with the special \ character are continued on the next line, forming a single logical line. A logical line starting with the pound (#) character is treated as a comment. Each program, along with its arguments, is listed on a separate logical line.

You can specify the `-h`, `-l`, `-np`, `-e`, and `-sp` options (from the `mpirun` command) in an appfile. Options following a program name are treated as the program's command line arguments and are not processed by `mpirun`.

The ranks of the processes in MPI_COMM_WORLD are guaranteed to be ordered according to their sequential order in an appfile.

The general form of an appfile entry is:

`[-h` *remote_host*`] [-e` *var*`[=`*val*`] [...]]`
`[-l` *user*`] [-sp` *paths*`] [-np #]` *program* `[`*args*`]`

where

`-h` *remote_host*

Specifies the remote host where a remote executable is stored (defaults to local host). *remote_host* is either a host name or an IP address.

`-np #`

Starts # copies of the program (defaults to one).

`-e` *var=val*

Sets the environment variable *var* for the program and gives it the value *val* if provided (defaults to not setting environment variables).

`-l` *user*

Specifies the user name on the target host (default is current user name).

`-sp` *paths*

Sets the target shell PATH environment variable to *paths*. Search paths are separated by the colon (:) character (default is do not override the path).

executable_name

Specifies the name of the executable to run.

args

Specifies command line arguments to the program.

One way to schedule processes on remote SPP1600, S-, and X-Class servers is to set the `-e` option in the appfile:

`-h` *remote_host* `-e MPI_TOPOLOGY=`*val* `[-np #]` *program* `[`*args*`]`

mpijob

`mpijob` lists the HP MPI jobs running on the system. The `mpijob` syntax is shown below:

`mpijob [-help] [-a] [-u] [-j` *id* `[...]]`

where

`-help`
Prints usage information for the utility.

`-a`
Lists jobs for all users.

`-u`
Sorts jobs by user name.

`-j` *id*
Provides process status for job *id*.

When invoked, `mpijob` reports the following information for each job:

JOB
HP MPI job identifier.

USER
User name of the owner.

NPROCS
Number of processes.

PROGNAME
Program names used in the HP MPI application.

By default, your jobs are listed by job ID in increasing order. However, you can specify the `-a` and `-u` options to change the default behavior.

If you specify the `-j` option, `mpijob` reports the following information for each job:

RANK
Rank for each process in the job.

HOST
Host where the job is running.

PID
Process identifier for each process in the job.

LIVE
Flag that indicates whether the process is running (an x is used) or has been terminated.

PROGNAME

Program names used in the HP MPI application.

A `mpijob` output using the `-a` and `-u` options is shown below. The output lists jobs for all users and sorts them by user name.

```
JOB      USER      NPROCS    PROGNAME
22617    estep     100       /home/estep/cool_dude
22573    guo       14        /home/guo/soccer_head
22677    orrick    4         /home/orrick/aggie_land
22623    raja      12        /home/raja/mpi_geek
22504    romero    10        /home/romero/diplomat_extraordinaire
22547    takao     8         /home/takao/xmpi_blues
```

mpiclean

`mpiclean` kills lingering processes in a running HP MPI application. `mpiclean` syntax has two forms:

- `mpiclean [-help] [-v] -j` *id* `[...]`
- `mpiclean [-help] [-v] [-sc` *name* `| -scid` *id*`] prog [...]`

where

`-help`

Prints usage information for the utility.

`-v`

Turns on verbose mode.

`-j` *id*

Kills the processes of job number *id*. You can specify multiple job IDs.

`-sc` *name*

Restricts the operation to the named subcomplex. This option is mutually exclusive with the `-scid` option.

`-scid` *id*

Restricts the operation to subcomplex number *id*. This option is mutually exclusive with the `-sc` option.

`prog`

Specifies the binary filename to kill. You can specify multiple filenames.

The first syntax is used for all servers. The second syntax is provided for backward compatibility on SPP1600, S-, and X-Class servers.

The MPI library checks for the abnormal termination of processes while your application is running. In some cases, application bugs can cause processes to deadlock and linger in the system. When this occurs, you can use `mpijob` to identify hung jobs and `mpiclean` to kill all processes in the hung application.

There are two ways to kill an HP MPI application.The preferred way is to provide `mpiclean` with the application's job ID (obtained by using the `-j` option when invoking `mpirun`). However, you can kill only jobs that you own. The second way is considered obsolete. It is provided on S- and X-Class servers only for backward compatibility. In this approach, you provide `mpiclean` with a list of binary filenames you own. `mpiclean` locates the matching processes and kills them.

You can restrict the second cleanup method to a single subcomplex by using the `-sc` or `-scid` options. This is helpful in cases where the same code is running independently on several subcomplexes and only one of these applications needs to be killed.

xmpi

`xmpi` invokes the XMPI utility. The `xmpi` syntax is shown below:

`xmpi [-h][-bg` *arg*`][-bd` *arg*`][-bw` *arg*`][-display` *arg*`]`
`[-fg` *arg*`][-geometry` *arg*`][-iconic][-title` *arg*`]`

where

`-h`
Prints usage information for the utility.

`-bg` *arg*
Specifies the background color.

`-bd` *arg*
Denotes the border color.

`-bw` *arg*
Specifies the width of the border in pixels.

`-display` *arg*
Designates the X-window display server to use.

`-fg` *arg*
Specifies the foreground color.

`-geometry` *arg*
Specifies size and position.

`-iconic`
Designates that the application start as an icon.

`-title` *arg*

Specifies the title of the application.

For more information, see "Using XMPI" on page 38.

mpitrget

`mpitrget` combines raw trace files (a raw trace file contains a history of the computation and communication processing for a single process) into a consolidated output file with a .tr suffix. This output file is then loaded and run on XMPI.

Note

You must make sure that mpitrget can access the trace files for all processes running locally and remotely.

`mpitrget` should only be used if the `-t` option was specified when invoking `mpirun` or the MPI_XMPI environment variable was set before invoking `mpirun`. The `mpitrget` syntax is shown below:

`mpitrget [-o` *filename*`] [-f] [-rm] [-v] [-help] -p` *prefix*

where

`-o` *filename*

Specifies the filename for output. If this option is not set, *prefix*.tr is used for the output filename.

`-f`

Removes the existing file before writing the output.

`-rm`

Removes the raw trace files after writing the output.

`-v`

Turns on verbose mode.

`-help`

Prints usage information for the utility.

`-p` *prefix*

Specifies the prefix used to name the input file. If *prefix* does not begin with a forward slash (/), the file is searched for in the directory in which `mpitrget` is executing.

For example,

```
% mpitrget -rm -p example
```

specifies that the output filename is example.tr and that the raw trace files are removed after example.tr is created.

Analyzing HP MPI applications

4

This chapter provides information about utilities used to analyze HP MPI applications. The topics covered are:

- Using the profiling interface
- Using XMPI
- Using CXpa, CXdb, and CXtrace
- Using HP-UX debuggers

Using the profiling interface

HP MPI provides a profiling interface for collecting statistics and measuring performance. The profiling interface allows you to intercept calls to the MPI library at link time and perform some action. For example, you may want to measure the time spent in each call to a certain library routine or create a logfile.

All routines in the MPI library begin with the `MPI` prefix. Based on the MPI standard, these routines are also callable using the `PMPI` prefix (for example, `PMPI_Send`).

To use the profiling interface, you write wrapper versions of the MPI library routines you want the linker to intercept. These wrapper routines collect data for some statistic or perform some other action. The wrapper then calls its corresponding routine in the MPI library using its `PMPI` prefix.

For example, suppose you want to measure the elapsed time for each call to `MPI_Send`. In this case, you create a wrapper called `MPI_Send` that uses `MPI_Wtime` to measure the elapsed time for each call. After `MPI_Wtime` completes, your wrapper then calls `PMPI_Send` from the MPI library to actually send the message.

Using XMPI

XMPI is an X/Motif graphical user interface for running applications, monitoring processes and messages, and viewing trace files. XMPI provides a graphical display of the state of processes within an HP MPI application.

XMPI is useful when analyzing programs at the application level (for example, examining HP MPI data types and communicators). Unlike other profilers and debuggers, you can run XMPI without having to recompile or relink your application.

XMPI runs in one of two modes: postmortem mode or interactive mode. In postmortem mode, you can view trace information for each process in your application. In interactive mode, you can monitor process communications and take snapshots while your application is actually running.

The default X resource settings that determine how XMPI displays on your workstation are stored in /opt/mpi/lib/X11/app-defaults/XMPI. See "XMPI resource file" on page 109 for a list of these settings.

Working with postmortem mode

In order to use XMPI's postmortem mode, you must first create a trace file. Then, you can load this file into XMPI to view state information for each process in your application.

Creating a trace file

Use these instructions to create a trace file:

Step 1 Enter

% **mpirun -t** *spec* **-np #** *program*

where

`-t` *spec*

Enables run-time raw trace generation for all processes. *spec* specifies options used when tracing. See "MPI_XMPI" on page 28 for more information about tracing options.

`-np #`

Specifies the number of processes to run.

program

Specifies the name of the executable to run.

A raw trace dump, *prefix.n*, is created for each application process where *n* ranges from 0 to (# - 1).

Step 2 Enter

% **mpitrget -p** *prefix*

where *prefix* specifies the prefix attached to the trace output file.

`mpitrget` consolidates all the individual trace dump files from step 1 into a single file called *prefix*.tr. See "mpitrget" on page 36 for information about other options you can specify.

Using the MPIHP_Trace_on and MPIHP_Trace_off routines

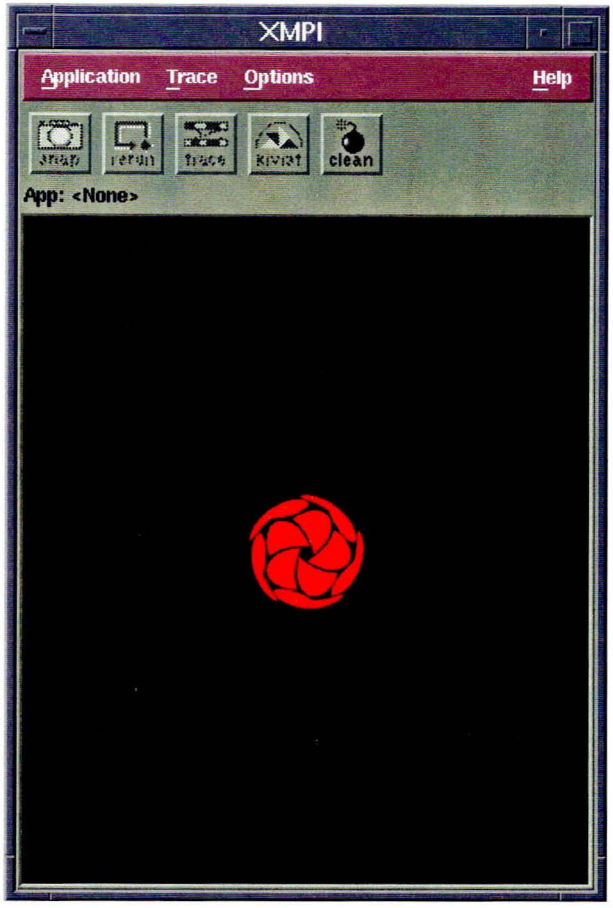
HP MPI provides the `MPIHP_Trace_on` and `MPIHP_Trace_off` routines to help troubleshoot application problems.

You insert the `MPIHP_Trace_on` and `MPIHP_Trace_off` pair around suspect code in your application. Then, you build the application and invoke `mpirun` with the `-t` option to enable application tracing. The trace information collected is only for the code between `MPIHP_Trace_on` and `MPIHP_Trace_off`. You can then create a consolidated trace file using `mpitrget` and run the trace file in XMPI to help identify problems during application execution.

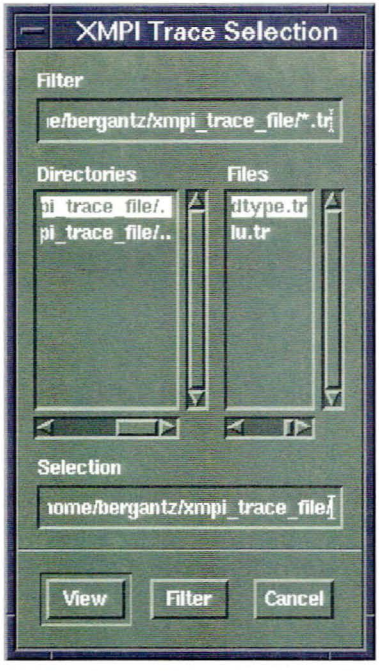
Viewing a trace file

Use these instructions to view a trace file:

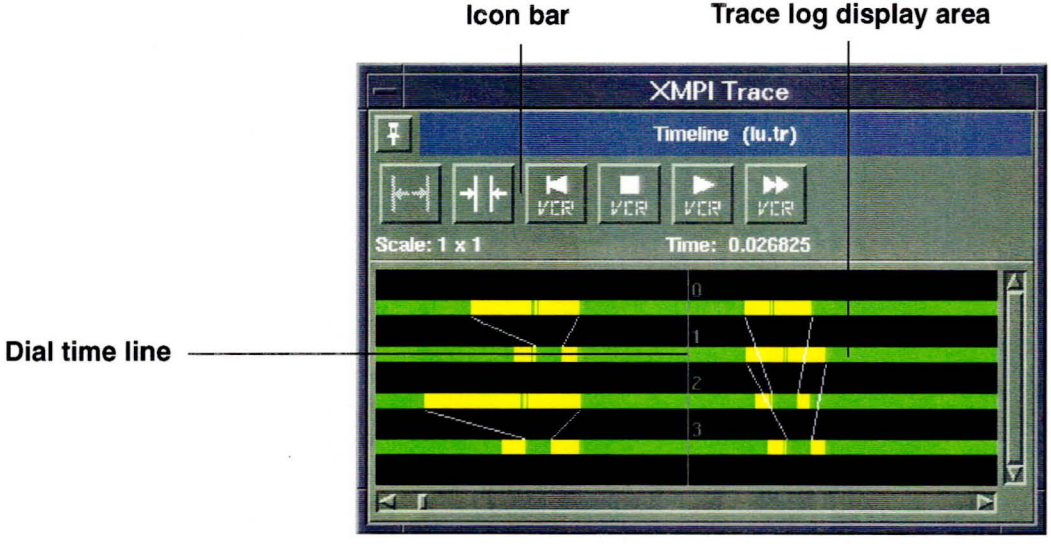
Step 1 Enter `xmpi` to open the XMPI main window (see "xmpi" on page 35 for information about other options you can specify).



Step 2 Select View from the Trace menu to open the XMPI Trace Selection dialog.



Step 3 Type the full path name of the appropriate trace file in the Selection field and choose View to open the XMPI Trace dialog.



The XPMI Trace dialog consists of an icon bar across the top, the current magnification and dial time just below, and a trace log display area below that.

The icon bar contains icons that (from left to right):

- Increase the magnification of the trace log.
- Decrease the magnification of the trace log.
- Rewind the trace log to the beginning. The dial time is also reset to the beginning.
- Stop playing the trace log.
- Play the trace log.
- Fast forward the trace log.

To set the magnification for viewing a trace file, select the Increase or Decrease icon on the icon bar.

The dial time indicates how long the application has been running in seconds.

The trace log display area shows a separate trace for each process in the application. The dial time is represented as a vertical line. The rank for each process is shown where the dial time line intersects a process trace.

The state of a process at any time is indicated by one of three colors:

Green

Signifies that a process is running outside MPI.

Red

Denotes that a process is blocked, waiting for communication to finish before the process resumes execution.

Yellow

Represents a process's overhead time inside MPI (for example, time spent doing message packing).

Blocking point-to-point communications are represented by a trace for each process showing the time spent in system overhead and time spent blocked waiting for communication. A line is drawn connecting the appropriate send and receive trace segments. The line starts at the beginning of the send segment and ends at the end of the receive segment.

For nonblocking point-to-point communications, a system overhead segment is drawn when a send and receive are initiated. When the communication is completed using a wait or a test, segments are drawn showing system overhead and blocking time. Lines are also drawn between matching sends and receives, except in this case, the line is drawn from the segment where the send was initiated to the segment where the corresponding receive completed.

Collective communications are also represented by a trace for each process showing the time spent in system overhead and time spent blocked waiting for communication.

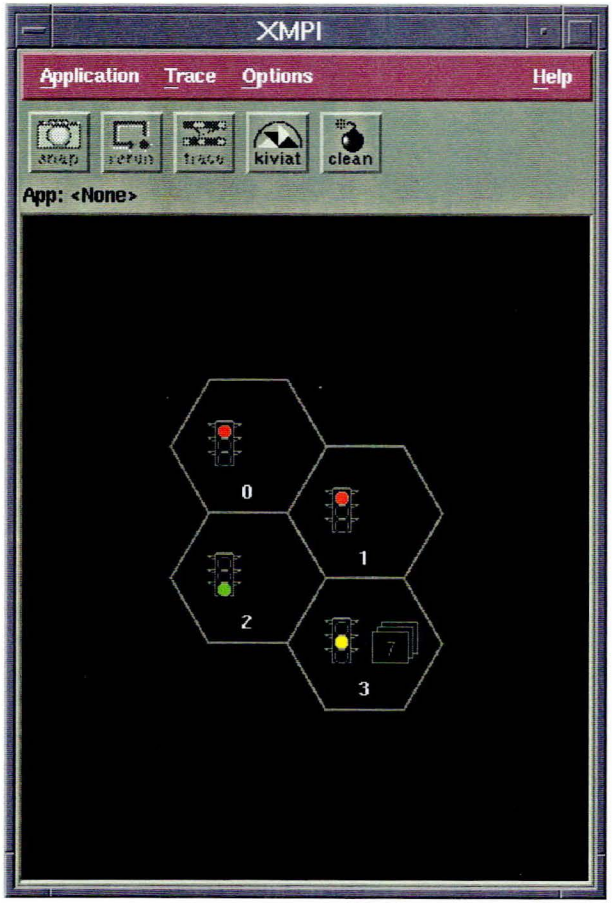
Owing to the use of partial tracing, some send and receive segments may not have a matching segment. In this case, a stub line is drawn out of the send segment or into the receive segment.

To play the trace file, select the Play or Fast forward icons on the icon bar. For any given dial time, the state of the trace file is reflected in the main window, the Focus dialog, the Datatype dialog, and the Kiviat dialog.

Viewing process information from a trace

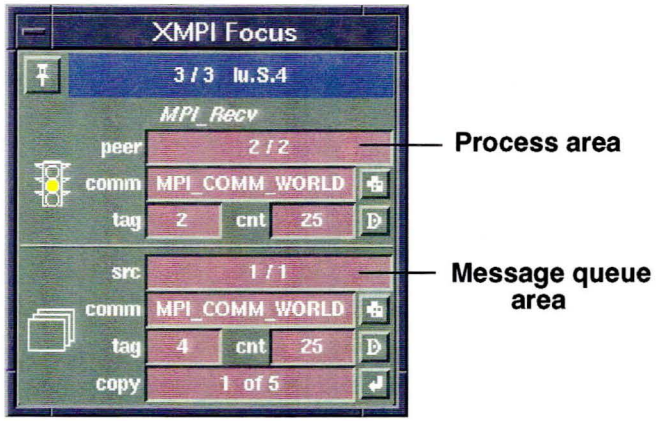
Use these instructions to view process information from a trace.

Step 1 Start XMPI and open a trace for viewing. The XMPI main window fills with a group of tiled hexagons, each representing the current state of a process and labelled by the process's rank within MPI_COMM_WORLD.



The current state of a process is indicated by the color of the signal light (either green, red, or yellow) in the hexagon. This color corresponds to the elapsed run time (current dial time) of the trace file in the XMPI Trace dialog. As the trace file is played, the color changes as processes communicate with each other.

Step 2 Select the hexagon representing the process you want more information about to open the XMPI Focus dialog.



The XMPI Focus dialog consists of a process area and a message queue area.

The values in the process area and message queue area fields correspond to the current dial time of the trace file in the XMPI Trace dialog. As the trace file is played, the values in the fields change as processes communicate with each other.

The process area describes the current state of a process together with the name and arguments for the HP MPI function currently being executed. The fields include:

peer

Displays the rank of a process. A process is identified by its rank in MPI_COMM_WORLD, a slash (/), and the rank of the process within the current communicator.

comm

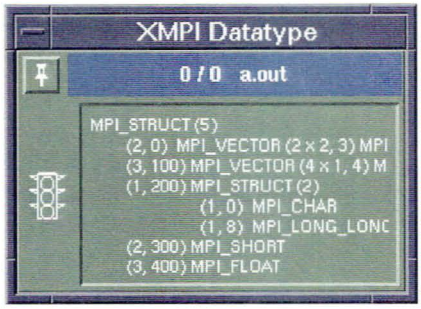
Shows the communicator being used by the HP MPI function. If you select the icon to the right of the comm field, the hexagons for processes that belong to the communicator are highlighted in the XMPI main window.

tag

Displays the value of the tag argument associated with the message.

cnt

Shows the count of the message data elements associated with the message when it was sent. Select the icon to the right of the cnt field to open the XMPI Datatype dialog.



The XMPI Datatype dialog displays the type map of the datatype associated with the message when it was sent. This datatype can be any one of the default datatypes (this varies depending upon the language used) or a user-defined datatype.

The datatype shown corresponds to the current dial time of the trace file in the XMPI Trace dialog. As the trace file is played, the datatype changes as processes communicate with each other.

The message queue area describes the current state of the queue of messages sent to the process but not yet received. The fields include:

src

Displays the rank of the process sending the message. A process is identified by its rank in MPI_COMM_WORLD, a slash (/), and the rank of the process within the current communicator.

comm

Shows the communicator being used by the HP MPI function. If you select the icon to the right of the comm field, the hexagons for processes that belong to the communicator are highlighted in the XMPI main window.

tag

Displays the value of the tag argument associated with the message when it was sent.

cnt

Shows the count of the message data elements associated with the message when it was sent. If you select the icon to the right of the cnt field, the XMPI Datatype dialog displays. The XMPI Datatype dialog displays the type map of the datatype associated with the message when it was sent.

copy

Displays the number of copies of the message that was sent. For example, if a process sends 10 messages to another where six of the messages have one type of message envelope and the remaining four have another, the copy field toggles between 6 of 10 and 4 of 10. In this case, a message envelope consists of the sender, the communicator, the tag, the count, and the datatype.

This behavior results from treating the six messages that all have the same envelope as one copy and the remaining four messages as a different copy. That way, if a communication involves a hundred messages all having the same envelope, you can work with a single copy rather than a hundred copies.

Step 3 Select Quit from the Application menu to close XMPI.

Viewing kiviat information from a trace file

Kiviat graphs are used to display performance data. Use these instructions to view kiviat information from a trace file.

Step 1 Start XMPI and open a trace for viewing.

Step 2 Select Kiviat from the Trace menu to open the XMPI Kiviat dialog.

The XMPI Kiviat window shows, in a segmented pie-chart format, the cumulative time up to the current dial time spent by each process in the running, overhead, and blocked states.

The cumulative time for each process corresponds to the current dial time of the trace file in the XMPI Trace dialog. As the trace file is played, the cumulative time changes as processes communicate with each other.

You can use the kiviat view to determine whether processes are load balanced and applications are synchronized. If an application is load balanced, the amount of time processes spend in each state should be equal. If an application is synchronized, the segments representing each of the three states should be concentric.

Step 3 Select Quit from the Application menu to close XMPI.

Working with interactive mode

In order to use XMPI's interactive mode, you should first set up your viewing options. Then, you can load an existing appfile into XMPI to view state information for each process in your application.

Setting up your viewing options

Use these instructions to set up your viewing options:

Step 1 Enter `xmpi` to open the XMPI main window (see "xmpi" on page 35 for information about other options you can specify).

Step 2 Select Monitoring from the Options menu to open the XMPI Monitoring dialog.

The fields include:

Automatic snapshot

Enables the automatic snapshot function. If automatic snapshot is enabled, XMPI takes snapshots of the application you are running and displays state information for each process.

If automatic snapshot is disabled, XMPI displays information for each process when the application begins, but this information is not updated. Disabling automatic snapshot may lead to buffer overflow problems because the contents of each process buffer are unloaded every time a snapshot is taken. For communication-intensive applications, process buffers can quickly fill and overflow.

You can enable or disable automatic snapshot while your application is running. This could be useful during troubleshooting when the application has run to a certain point and you want to disable automatic snapshot to study process state information.

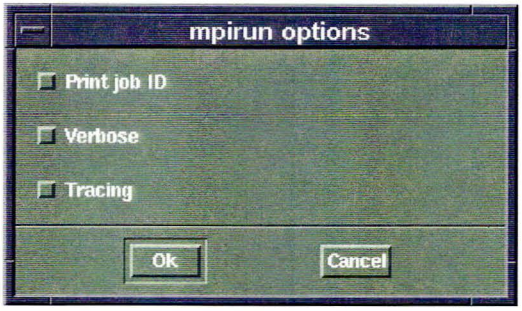
Monitor interval in second

Determines how often XMPI takes a snapshot when automatic snapshot is enabled.

Step 3 Select Buffers from the Options menu to open the XMPI Buffers dialog.

The single field, Enter message status table size in kilobytes, specifies the size of each process buffer. When you run an application, state information for each process is stored in a separate buffer. You may need to increase buffer size if overflow problems occur.

Step 4 Select mpirun from the Options menu to open the mpirun options dialog.



The fields include:

Print job ID

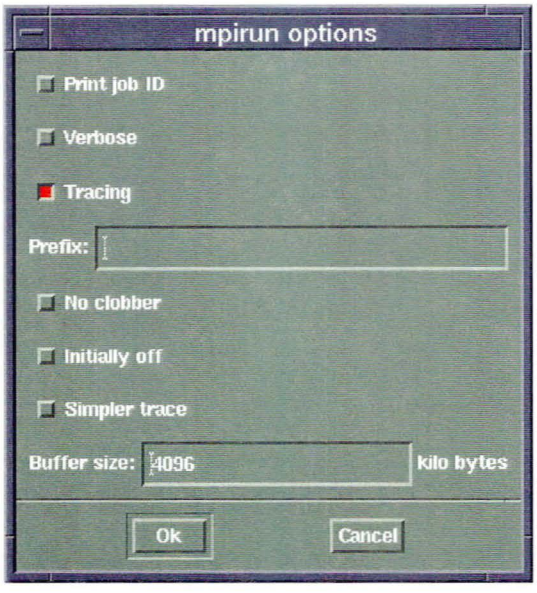
Enables printing of the HP MPI job ID.

Verbose

Enables verbose mode.

Tracing

Enables run-time raw trace generation for all application processes. If you select the Tracing button, the mpirun options trace dialog is opened.



The fields include:

Prefix

Specifies the prefix name for the file where each process writes its own raw trace data. Each process creates its own filename by concatenating the prefix, a period, and the process's global rank number.

No clobber

Specifies no clobber, which means that an HP MPI application aborts if a file with the name specified in the Prefix field already exists.

Initially off

Specifies that trace generation is initially turned off.

Simpler trace

Specifies a simpler tracing mode by omitting `MPI_Test`, `MPI_Testall`, `MPI_Testany`, and `MPI_Testsome` calls in the application that do not complete a request.

Buffer size

Denotes the buffering size in kbytes for dumping raw trace data. Actual buffering size may be rounded up by the system. The default buffering size is 4096 kbytes. Specifying a large buffering size reduces the need to flush raw trace data to a file when process buffers reach capacity. Flushing too frequently can cause communication routines to run slower. If this problem occurs, increase the buffering size.

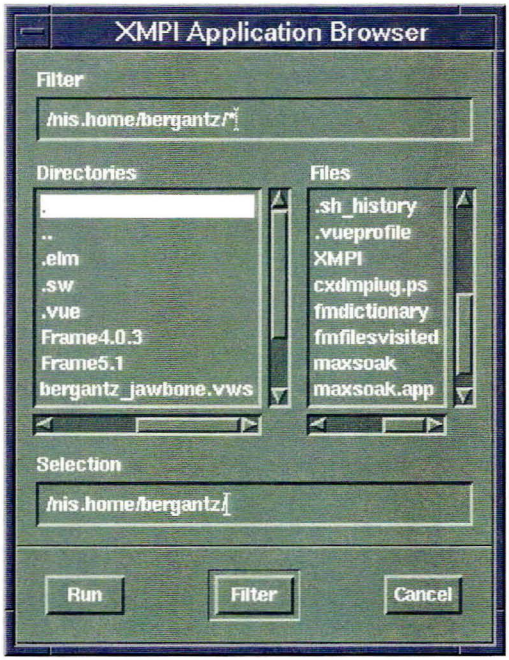
Step 5 Select Quit from the Application menu to close XMPI.

Running an appfile

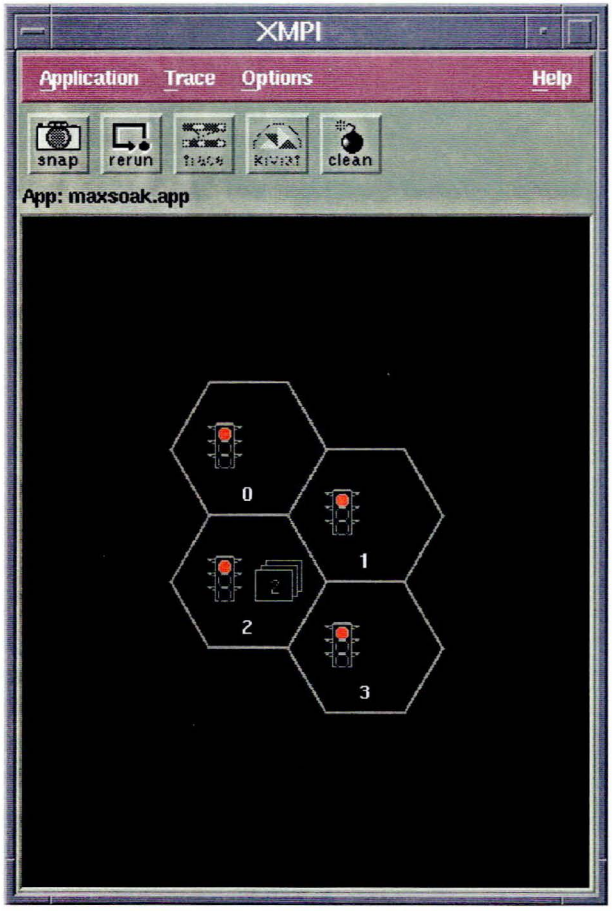
Use these instructions to view an appfile:

Step 1 Enter `xmpi` to open the XMPI main window (see "xmpi" on page 35 for information about other options you can specify).

Step 2 Select Browse&Run from the Application menu to open the XMPI Application Browser dialog.



Step 3 Type the full path name of an existing appfile in the Selection field and choose Run. The XMPI main window fills with a group of tiled hexagons, each representing the current state of a process and labelled by the process's rank within MPI_COMM_WORLD.



The current state of a process is indicated by the color of the signal light (either green, red, or yellow) in the hexagon. These process hexagons disappear when the application has run to completion.

If automatic snapshot is enabled, XMPI takes snapshots of the application and displays state information for each process on the XMPI main window, the XMPI Focus dialog, and the XMPI Datatype dialog. Refer to "Viewing process information from a trace" on page 43 for more information about accessing the XMPI Focus and XMPI Datatype dialogs.

If automatic snapshot is disabled, XMPI displays information for each process when the application begins, but this information is not updated.

Regardless of whether automatic snapshot is enabled, you can take application snapshots manually by selecting Snapshot from the Application menu. In this case, XMPI displays information for each process, but this information is not updated until you select Snapshot from the Application menu again.

Step 4 Select Clean from the Application menu at any time to kill the application and close any associated XMPI Focus and XMPI Datatype dialogs. You can then run another application by selecting an appfile from the XMPI Application Browser dialog.

Step 5 Select Quit from the Application menu to close XMPI.

Using CXpa, CXdb, and CXtrace

CX tools are useful when analyzing applications at the process level. These tools are available for applications running only on SPP1600, S-, and X-Class servers.

CX tools rely heavily on the compiler and the linker. All applications must be recompiled before using CXdb and CXtrace, and all applications must be relinked before using CXpa. Presently, the C, C++, and Fortran 77 compilers are supported.

CXpa

CXpa allows you to profile each process in an HP MPI application. The profile information is stored in a separate performance data file. During analysis, you merge the data from these separate files into a single performance data file for the application.

With CXpa, you can analyze data using one or more of the following metrics:

- Wall clock time
- CPU time
- Execution counts
- Cache miss counts
- Latency time
- Dynamic call graph

You can display the data as a 3D profile, a 2D profile, a report, or a dynamic call graph.

For more information about profiling HP MPI applications using CXpa, see *CXpa Reference* (B5639-90002).

CXdb

CXdb provides a single, scalable interface for debugging multiple processes in HP MPI applications. You can apply debugging commands and view information for individual processes and threads, groups of processes and threads, or all processes and threads.

CXdb provides the standard set of debugging functions including:

- Setting breakpoints, tracepoints, and watchpoints
- Printing and modifying variables
- Viewing source and assembly code
- Viewing the contents of application registers

To debug symbolically with CXdb, you must have compiled and linked your application with the `-g` option. If you do not specify the `-g` option, you can still perform assembly-language debugging.

For more information about debugging HP MPI applications using CXdb, see CXdb online help or the CXdb man pages.

CXtrace

CXtrace is a performance analysis tool. Its instrumentation and monitoring systems help improve parallel applications by capturing and visualizing execution trace information.

CXtrace instruments at the source-code level. If you change the instrumentation, however, you must recompile your application before using CXtrace.

CXtrace works well on small- and medium-sized trace files (medium-sized files average about 200 kbytes). CXtrace also provides information about message traffic patterns over time and source code-to-message correlation.

For more information about analyzing HP MPI applications using CXtrace, see *CXtrace User's Guide* (B5639-90003).

Using HP-UX debuggers

If you are running your application on a D- or K-Class server, you can use the DDE and xdb debuggers to troubleshoot your applications. Refer to "MPI_FLAGS" on page 25 for more information about setting the `MPI_FLAGS` environment variable to access these debuggers.

Tuning application performance

5

This chapter provides information about tuning applications to improve performance. The topics covered are:

- General tuning
- SPP1600, S-, and X-Class tuning

The general tuning information applies to applications running on all Exemplar servers. The SPP1600, S-, and X-Class tuning information applies only to applications running on these servers.

General tuning

When you are developing HP MPI applications, several factors can affect performance. These factors include:

- Message latency and bandwidth
- CPU subscription
- MPI routine selection

Message latency and bandwidth

Latency is the time between the initiation of the data transfer in the sending process and the arrival of the first byte in the receiving process.

Latency is often dependent upon the length of messages being sent. An application's messaging behavior can vary greatly based upon whether a large number of small messages or a few large messages are sent.

Message bandwidth is the reciprocal of the time needed to transfer a byte. Bandwidth is normally expressed in megabytes per second. Bandwidth becomes important when message sizes are large.

To improve latency and bandwidth:

- Reduce the number of process communications by designing coarse-grained applications.
- Use contiguous data to avoid having to pack and unpack data in sending and receiving buffers.
- Use collective operations whenever possible. This eliminates the overhead of using `MPI_Send` and `MPI_Recv` each time when one process communicates with others.

CPU subscription

Subscription refers to the match of CPUs and active processes on a host or subcomplex. Table 9 lists the possible subscription types.

Table 9 Subscription types

Subscription type	Description
Under subscribed	More CPUs than active processes
Fully subscribed	Equal number of CPUs and active processes
Over subscribed	More active processes than CPUs

When a host or subcomplex is over subscribed, application performance decreases because of increased context switching.

Context switching can degrade application performance by slowing the computation phase, increasing message latency, and lowering message bandwidth. Simulations that use timing-sensitive algorithms can produce unexpected or erroneous results when run on an over subscribed system.

So, to get peak performance when running your application, minimize context switching by reducing over subscription on any hosts and subcomplexes you use.

MPI routine selection

To achieve the lowest message latencies and highest message bandwidths for point-to-point communications, use the MPI blocking routines `MPI_Send()` and `MPI_Recv()` whenever possible.

For tasks that require collective operations, use the appropriate MPI collective routine. HP MPI takes advantage of the Exemplar's shared memory to perform efficient data movement and maximize your application's communication performance.

SPP1600, S-, and X-Class tuning

When working with HP MPI applications on SPP1600, S-, and X-Class servers, several factors can affect application performance. These factors include:

- Multilevel parallelism
- Process placement

Multilevel parallelism

There are several ways to improve the performance of applications that use multilevel parallelism:

- Use the MPI library to provide coarse-grained parallelism and a parallelizing compiler to provide fine-grained (that is, thread-based) parallelism. An appropriate mix of coarse- and fine-grained parallelism provides better overall performance.
- Assign only one multithreaded process per hypernode when placing application processes. This ensures that enough CPUs are available as different process threads become active.
- Restrict MPI calls (for example, send and receive) to only a single thread.

Process placement

Because messaging bandwidth and latency are better within a hypernode than between hypernodes, you can improve performance by placing HP MPI processes that communicate heavily on the same hypernode. One way to do this is to use the `MPI_TOPOLOGY` environment variable to tell an application the number of processes to run on each available hypernode.

For example, suppose you want to run an application on a SPP1600 server using a subcomplex called System. This subcomplex spans four hypernodes and contains the 20 CPUs listed below:

- Hypernode 0: 5 CPUs
- Hypernode 1: 2 CPUs
- Hypernode 2: 5 CPUs
- Hypernode 3: 8 CPUs

Suppose the application you want to run contains the 16 processes listed below:

- Set 1: Ranks 1-4
- Set 2: Ranks 5-8
- Set 3: Ranks 9-12
- Set 4: Ranks 13-16

Ideally, you should use a process placement that allows each set of processes to run on a single hypernode to maximize message-passing performance.

By default, HP MPI places processes by fully subscribing each hypernode before moving on to the next. If the processes in your application are placed using this approach, you get the placement shown in Figure 5.

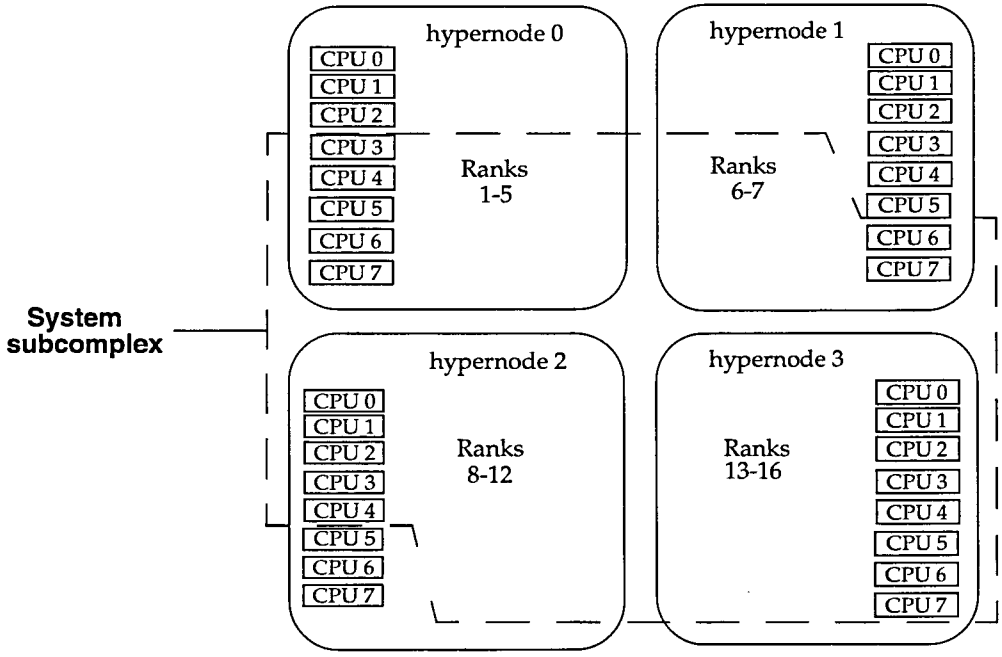


Figure 5 Default process placement

While this distribution prevents CPU oversubscription, it does not provide optimum message-passing performance because the processes from sets two and three are split across hypernodes. Communications within these process groups may become a bottleneck when running the application.

You can solve this problem by specifying the number of processes you want to run on each hypernode as shown below:

- Hypernode 0 --> Ranks 1-4
- Hypernode 1 --> unused
- Hypernode 2 --> Ranks 5-8
- Hypernode 3 --> Ranks 9-16

This distribution results in a placement shown in Figure 6.

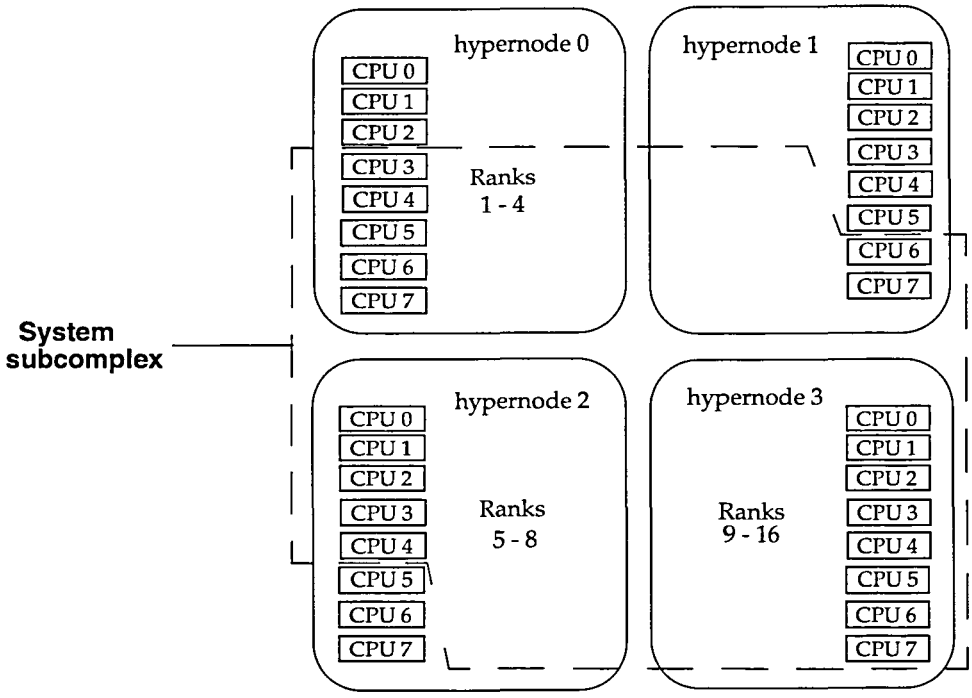


Figure 6 Optimal process placement

To specify this process placement, set `MPI_TOPOLOGY` by entering:

```
% setenv MPI_TOPOLOGY 4,0,4,8
```

For more information, see "MPI_TOPOLOGY" on page 26.

Troubleshooting applications 6

This chapter describes hints and limitations when working with HP MPI applications. You should check this information first when troubleshooting problems. The topics covered are organized by development task and include problems in areas such as:

- Building
- Starting
- Running
- Completing

Building

You can solve most build-time problems by referring to the documentation for the compiler you are using.

If you decide to use your own build script, be sure to specify all necessary input libraries. To determine what libraries are needed, check the contents of the compilation utilities stored in the HP MPI /opt/mpi/bin subdirectory.

When compiling Fortran 77 applications, avoid using the `+autodblpad` option if you only want to double the size of floating-point constants and variables. Instead, create a compiler directive file with the statement `$autodbl dbl`. Then, specify `+Q` *filename* when invoking the compiler where *filename* is the name of your directive file.

Starting

When starting multihost applications, make sure that:

- All remote hosts are listed in your .rhosts file
- Application binaries are available on the necessary remote hosts and are executable
- The `-sp` option is passed to `mpirun` via an appfile if necessary

Running

Run-time problems originate from many sources. These sources include interoperability, message buffering, input and output, certain Fortran 90 programming features, and UNIX open file descriptors among others.

SPP-UX and HP-UX interoperability

Depending upon what server resources are available, applications may run on heterogeneous systems such that certain portions run on SPP-UX servers (SPP1600, S-, and X-Class) and other portions run on HP-UX servers (D- and K-Class servers).

For example, suppose you create a MPMD application that calculates the average acceleration of particles in a simulated cyclotron. The application consists of a four-process program called sum_accelerations and an eight-process program called calculate_average.

Because you have access to a K-Class server called hpux_server and an X-Class server called sppux_server, you create the following appfile:

```
-h hpux_server -np 4 sum_accelerations
-h sppux_server -np 8 calculate_average
```

Then, you invoke `mpirun` passing it the name of the appfile you created. In this case, even though the two application programs run on different platforms, all processes can communicate with each other resulting in twelve-way parallelism. The four processes belonging to the sum_accelerations application are ranked 0 through 3, and the eight processes belonging to the calculate_average application are ranked 4 through 11.

Message buffering

According to the MPI 1.1 standard, message buffering may or may not occur when processes communicate with each other. You should, therefore, be careful when coding communications that depend upon buffering to work correctly.

For example, when two processes use `MPI_Send` to simultaneously send a message to each other and use `MPI_Recv` to receive the messages, the results are unpredictable. If the messages are buffered, communication works correctly. If the messages are not buffered, however, each process hangs in `MPI_Send` waiting for `MPI_Recv` to take the message.

External input and output

Each process in HP MPI applications can read and write data. In some applications, however, having one process handle all input and output (and communicate with other processes using collective operations) is more efficient.

You can use stdin and stdout in your applications to read and write data. Stdout is supported regardless of whether your application runs locally or remotely. Stdin, however, may or may not be supported depending upon how you run your application, whether the application is run locally or remotely, and whether the `-W` option is used when invoking `mpirun`. The run invocations under which stdin is supported are shown in Table 10 for the hello_world application. All multihost invocations use an appfile called hello_world.

Table 10 Run invocations that support stdin

Run invocation	Is stdin supported?
`hello_world -np #`	Yes
`mpirun -np # hello_world`	Yes
`mpirun -W -np # hello_world`	No
`mpirun -W -np # -f hello_world` (multihost local and remote)	No
`mpirun -np # -f hello_world` (multihost local)	Yes
`mpirun -np # -f hello_world` (multihost remote)	No

Fortran 90 programming

The MPI 1.1 standard defines bindings for Fortran 77 but not Fortran 90.

Although most Fortran 90 MPI applications work using the Fortran 77 MPI bindings, some Fortran 90 features can cause unexpected behavior when used with HP MPI.

In Fortran 90, an array is not always stored in contiguous memory. When noncontiguous array data are passed to an HP MPI subroutine, Fortran 90 copies the data into temporary storage, passes it to the HP MPI subroutine, and copies it back when the subroutine returns. As a result, HP MPI is given the address of the copy but not the original data.

In some cases, this copy-in and copy-out operation can cause a problem. For a nonblocking HP MPI call, the subroutine returns immediately and the temporary storage is deallocated. When HP MPI tries to access the already invalid memory, the behavior is unknown. Moreover, HP MPI operates close to the system level and needs to know the address of the original data (although even if the address is known, HP MPI does not know if the data are contiguous or not).

UNIX open file descriptors

UNIX imposes a limit to the number of file descriptors that application processes can have open at one time. When running a multihost application, each local process opens a socket to each remote process. An HP MPI application with a large amount of off-host processes can quickly reach the file descriptor limit. Ask your system administrator to increase the limit if your applications frequently exceed the maximum.

Completing

In HP MPI, `MPI_Finalize` is a barrier-like collective routine that waits until all application processes have called it before returning. If your application exits without calling `MPI_Finalize`, pending requests may not complete.

When running an application, `mpirun` normally waits until all processes have called `MPI_Finalize`.

You can also use `mpijob` and `mpiclean` to detect and clean up suspect applications that may have terminated abnormally.

Each HP MPI application is identified by a job ID, unique on the server where `mpirun` is invoked. If you use the `-j` option, `mpirun` prints the job ID of the application that it runs. Then, you can invoke `mpijob` with the job ID to display the status of your application.

If your application hangs or terminates abnormally, you can use `mpiclean` to kill any lingering processes. In this context, you use the job ID from `mpirun` to specify the application to terminate.

MPI library routines and extensions

A

This appendix describes the 128 MPI library routines and three HP MPI library extensions. The library routines and extensions are listed alphabetically.

Table 11 describes the C version of the routines. Table 12 describes the Fortran 77 version of the routines. Table 13 describes the C version of the library extensions. Table 14 describes the Fortran 77 version of the library extensions.

C version of MPI routines

Table 11 C version of MPI routines

Routine	Description
int **MPI_Abort**(MPI_Comm comm, int errorcode)	Aborts all tasks in a communicator.
int **MPI_Address**(void* location, MPI_Aint *address)	Returns the address of a location.
int **MPI_Allgather**(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcounts, MPI_Datatype recvtype, MPI_Comm comm)	Sends the contents of each process's send buffer to all other processes.
int **MPI_Allgatherv**(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, MPI_Comm comm)	Sends a varying count of data from each process's send buffer to all other processes.
int **MPI_Allreduce**(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)	Combines the elements in the input buffer of each process and returns the combined value to the receive buffer of each process.

Table 11 C version of MPI routines **(continued)**

Routine	Description
int **MPI_Alltoall**(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)	Sends distinct data from each process to all other processes.
int **MPI_Alltoallv**(void* sendbuf, int *sendcounts, int *sdispls, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)	Sends distinct data from each process to all other processes. The location of data for the send and the location of the placement of the data on the receive side are specified in the call.
int **MPI_Attr_delete**(MPI_Comm comm, int keyval)	Deletes attributes from the cache based upon a key.
int **MPI_Attr_get**(MPI_Comm comm, int keyval, void *attribute_val, int *flag)	Receives attribute values based upon a key.
int **MPI_Attr_put**(MPI_Comm comm, int keyval, void* attribute_val)	Stores an attribute value.
int **MPI_Barrier**(MPI_Comm comm)	Blocks the calling process until all other processes in the group have called the routine.
int **MPI_Bcast**(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)	Broadcasts a message from the root process to all other processes in the group including itself.
int **MPI_Bsend**(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)	Sends a message in buffered mode.
int **MPI_Bsend_init**(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_request *request)	Creates a persistent communication request for a buffered mode send.
int **MPI_Buffer_attach**(void* buffer, int size)	Provides MPI with a buffer in the user's memory that can be used for buffering outgoing messages.
int **MPI_Buffer_detach**(void* buffer, int* size)	Detaches the buffer currently associated with MPI.
int **MPI_Cancel**(MPI_Request *request)	Marks a pending, nonblocking send or receive call for cancellation.
int **MPI_Cart_coords**(MPI_Comm comm, int rank, int maxdims, int *coords)	Translates process ranks to logical process coordinates as the ranks are used in point-to-point communications.
int **MPI_Cart_create**(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)	Returns a handle to a new communicator to which cartesian topology information is attached.

Table 11 C version of MPI routines **(continued)**

Routine	Description
int **MPI_Cart_get**(MPI_Comm comm, int maxdims, int *dims, int *periods, int *coords)	Returns the cartesian topology information associated with a communicator.
int **MPI_Cart_map**(MPI_Comm comm, int ndims, int *dims, int *periods, int *newrank)	Computes an optimal placement for the calling process on the physical machine.
int **MPI_Cart_rank**(MPI_Comm comm, int *coords, int *rank)	Translates logical process coordinates to process ranks as the coordinates are used in point-to-point communications.
int **MPI_Cart_shift**(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)	Returns the shifted source and destination ranks given a shift amount and direction.
int **MPI_Cart_sub**(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)	Partitions the communicator group into subgroups that form lower-dimensional cartesian subgrids.
int **MPI_Cartdim_get**(MPI_Comm comm, int *ndims)	Returns the cartesian topology information associated with a communicator.
int **MPI_Comm_compare**(MPI_Comm comm1, MPI_Comm comm2, int *result)	Compares two communicators.
int **MPI_Comm_create**(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)	Creates a new communicator and context with a communication group.
int **MPI_Comm_dup**(MPI_Comm comm, MPI_Comm *newcomm)	Duplicates the existing communicator with associated key values.
int **MPI_Comm_free**(MPI_Comm *comm)	Marks the communication object for deallocation.
int **MPI_Comm_group**(MPI_Comm comm, MPI_Group *group)	Returns a handle to the group in the communicator.
int **MPI_Comm_rank**(MPI_Comm comm, int *rank)	Returns the rank of a process in the communicator's group.
int **MPI_Comm_remote_group**(MPI_Comm comm, MPI_Group *group)	Returns the remote group in the intercommunicator.
int **MPI_Comm_remote_size**(MPI_Comm comm, int *size)	Returns the size of the remote group in the intercommunicator.
int **MPI_Comm_size**(MPI_Comm comm, int *size)	Returns the number of processes involved in a communicator.
int **MPI_Comm_split**(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)	Partitions the group associated with the communicator into disjoint subgroups.

Table 11 C version of MPI routines **(continued)**

Routine	Description
int **MPI_Comm_test_inter**(MPI_Comm comm, int *flag)	Determines whether a communicator is an intercommunicator or not.
int **MPI_Dims_create**(int nnodes, int ndims, int *dims)	Helps select a balanced distribution of processes per coordinate direction.
int **MPI_Errhandler_create**(MPI_Handler_function *function, MPI_Errhandler *errhandler)	Registers a user-specified routine for use as an exception handler.
int **MPI_Errhandler_free**(MPI_Errhandler *errhandler)	Marks an error handler for deallocation.
int **MPI_Errhandler_get**(MPI_Comm comm, MPI_Errhandler *errhandler)	Returns a handle to the error handler that is currently associated with the communicator.
int **MPI_Errhandler_set**(MPI_Comm comm, MPI_Errhandler errhandler)	Associates a new error handler with the communicator at the calling process.
int **MPI_Error_class**(int errorcode, int *errorclass)	Maps each standard error code onto itself.
int **MPI_Error_string**(int errorcode, char *string, int *resultlen)	Returns the error string associated with an error code.
int **MPI_Finalize**(void)	Cleans up all MPI states.
int **MPI_Gather**(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)	Sends the contents of each process's send buffer to the root process.
int **MPI_Gatherv**(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm)	Sends a varying count of data from each process's send buffer to the root process.
int **MPI_Get_count**(MPI_Status *status, MPI_Datatype datatype, int *count)	Returns the number of entries received in the receive buffer.
int **MPI_Get_elements**(MPI_Status *status, MPI_Datatype datatype, int *elements)	Returns the number of elements in a datatype.
int **MPI_Get_processor_name**(char *name, int len)	Returns the name of the processor on which it was called at the moment of the call.
int **MPI_Graph_create**(MPI_Comm comm_old, int nnodes, int *index, int *edges, int reorder, MPI_Comm *comm_graph)	Returns a handle to a new communicator to which the graph topology information is attached.
int **MPI_Graph_get**(MPI_Comm comm, int maxindex, int maxedges, int *index, int *edges)	Retrieves graph topology information associated with a communicator.

Table 11 C version of MPI routines **(continued)**

Routine	Description
int **MPI_Graph_map**(MPI_Comm comm, int nnodes, int *index, int *edges, int *newrank)	Maps process to graph topology information.
int **MPI_Graph_neighbors**(MPI_Comm comm, int rank, int maxneighbors, int *neighbors)	Returns the neighbors of a node associated with a graph topology.
int **MPI_Graph_neighbors_count**(MPI_Comm comm, int rank, int *nneighbors)	Returns the number of neighbors of a node associated with a graph topology.
int **MPI_Graphdims_get**(MPI_Comm comm, int *nnodes, int *nedges)	Retrieves graph topology information associated with a communicator.
int **MPI_Group_compare**(MPI_Group group1, MPI_Group group2, int *result)	Returns the relationship between two groups.
int **MPI_Group_difference**(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)	Creates a group by computing the difference between two existing groups.
int **MPI_Group_excl**(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)	Creates a group by reordering an existing group and only taking unlisted members.
int **MPI_Group_free**(MPI_Group *group)	Marks a group object for deallocation.
int **MPI_Group_incl**(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)	Creates a group by reordering an existing group and only taking listed members.
int **MPI_Group_intersection**(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)	Creates a group by computing the intersection of two existing groups.
int **MPI_Group_range_excl**(MPI_Group group, int n, int ranges [] [3], MPI_Group *newgroup)	Creates a group by excluding ranges of processes from an existing group.
int **MPI_Group_range_incl**(MPI_Group group, int n, int ranges [] [3], MPI_Group *newgroup)	Creates a group from ranges of ranks in an existing group.
int **MPI_Group_rank**(MPI_Group group, int *rank)	Returns the rank of this process in a group.
int **MPI_Group_size**(MPI_Group group, int *size)	Returns the size or number of processes in the group.
int **MPI_Group_translate_ranks**(MPI_Group group1, int n, int *ranks1, MPI_Group group2, int *ranks2)	Determines the relative numbering of the same processes in two different groups.
int **MPI_Group_union**(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)	Creates a group by computing the union of two existing groups.
int **MPI_Ibsend**(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)	Starts a buffered mode, nonblocking send.
int **MPI_Init**(int *argc, char ***argv)	Initializes the MPI execution environment.

Table 11 C version of MPI routines **(continued)**

Routine	Description
int **MPI_Initialized**(int *flag)	Determines whether MPI_Init has been called.
int **MPI_Intercomm_create**(MPI_Comm local_comm, int local_leader, MPI_Comm peer_comm, int remote_leader, int tag, MPI_Comm *newintercomm)	Creates an intercommunicator.
int **MPI_Intercomm_merge**(MPI_Comm intercomm, int high, MPI_Comm *newintracomm)	Creates an intracommunicator from the union of two groups.
int **MPI_Iprobe**(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)	Returns flag = true if there is a message that can be received and that matches the pattern specified by the arguments.
int **MPI_Irecv**(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)	Starts a nonblocking receive.
int **MPI_Irsend**(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)	Starts a ready mode, nonblocking send.
int **MPI_Isend**(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)	Starts a standard mode, nonblocking send.
int **MPI_Issend**(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)	Starts a synchronous mode, nonblocking send.
int **MPI_Keyval_create**(MPI_Copy_function *copy_fn, MPI_Delete_function *delete_fn, int *keyval, void* extra_state)	Generates a new attribute key.
int **MPI_Keyval_free**(int *keyval)	Frees an existing attribute key.
int **MPI_Op_create**(MPI_User_function function, int commute, MPI_Op *op)	Binds a user-defined global operation to a handle.
int **MPI_Op_free**(MPI_Op *op)	Marks a user-defined reduction operation for deallocation.
int **MPI_Pack**(void* inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outsize, int *position, MPI_Comm comm)	Packs the message in the send buffer into the specified buffer space.
int **MPI_Pack_size**(int incount, MPI_Datatype datatype, MPI_Comm comm, int *size)	Returns the maximum number of buffer bytes required to hold the data.
int **MPI_Pcontrol**(const int level, ...)	Calls the specified profiling package.

Table 11 C version of MPI routines **(continued)**

Routine	Description
int **MPI_Probe**(int source, int tag, MPI_Comm comm, MPI_Status *status)	Returns only after a message is found that matches the pattern specified by the arguments.
int **MPI_Recv**(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)	Starts a blocking receive.
int **MPI_Recv_init**(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)	Creates a persistent communication request for a receive operation.
int **MPI_Reduce**(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)	Combines the elements in the input buffer of each process in the group and returns the combined value in the output buffer of the root process.
int **MPI_Reduce_scatter**(void* sendbuf, void* recvbuf, int *recvcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)	Combines values and scatters the results.
int **MPI_Request_free**(MPI_Request *request)	Marks the request object for deallocation.
int **MPI_Rsend**(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)	Sends a message in ready mode.
int **MPI_Rsend_init**(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)	Creates a persistent communication object for a ready mode send operation.
int **MPI_Scan**(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)	Performs a prefix reduction on data distributed across the group.
int **MPI_Scatter**(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)	Sends the contents of the root process's send buffer to other processes in the group.
int **MPI_Scatterv**(void* sendbuf, int *sendcounts, int *displs, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)	Sends a varying count of data from the root process's send buffer to other processes in the group.
int **MPI_Send**(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)	Starts a standard mode, blocking send.
int **MPI_Send_init**(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)	Creates a persistent communication request for a standard mode send operation and binds it to all the arguments of a send operation.

Table 11 C version of MPI routines **(continued)**

Routine	Description
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)	Executes a blocking send and receive.
int **MPI_Sendrecv_replace**(void* buf, int count, MPI_Datatype datatype, int dest, int sendtag, int source, int revctag, MPI_Comm comm, MPI_Status *status)	Executes a blocking send and receive where both operations use the same buffer.
int **MPI_Ssend**(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)	Sends a message in synchronous mode.
int **MPI_Ssend_init**(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)	Creates a persistent communication object for a synchronous mode send operation.
int **MPI_Start**(MPI_Request *request)	Initiates a communication with a persistent request handle.
int **MPI_Startall**(int count, MPI_Request *array_of_requests)	Initiates a collection of requests.
int **MPI_Test**(MPI_Request *request, int *flag, MPI_Status *status)	Returns flag = true if the request (operation) identified in the call is complete.
int **MPI_Test_cancelled**(MPI_Status *status, int *flag)	Marks a pending, nonblocking communication operation (send or receive) for cancellation.
int **MPI_Testall**(int count, MPI_Request *array_of_requests, int *flag, MPI_Status *array_of_statuses)	Returns flag = true if all communications associated with active handles in the array have completed.
int **MPI_Testany**(int count, MPI_Request *array_of_requests, int *index, int *flag, MPI_Status *status)	Tests for completion of either one or none of the operations associated with active handles.
int **MPI_Testsome**(int incount, MPI_Request *array_of_requests, int *outcount, int *array_of_indices, MPI_Status *array_of_statuses)	Tests for some given communications to complete.
int **MPI_Topo_test**(MPI_Comm comm, int *status)	Returns the type of topology that is assigned to a communicator.
int **MPI_Type_commit**(MPI_Datatype *datatype)	Commits the datatype (that is, the formal description of a communication buffer), not the content of that buffer.
int **MPI_Type_contiguous**(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)	Creates a contiguous datatype.

Table 11 C version of MPI routines **(continued)**

Routine	Description
int **MPI_Type_extent**(MPI_Datatype datatype, MPI_Aint *extent)	Returns the extent of a datatype.
int **MPI_Type_free**(MPI_Datatype *datatype)	Marks the datatype object identified in the call for deallocation.
int **MPI_Type_hindexed**(int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)	Creates an indexed datatype with offsets in bytes.
int **MPI_Type_hvector**(int count, int blocklength, MPI_Aint stride, MPI_Datatype oldtype, MPI_datatype *newtype)	Creates a vector (strided) datatype with offset in bytes.
int **MPI_Type_indexed**(int count, int *array_of_blocklengths, int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)	Creates an indexed datatype.
int **MPI_Type_lb**(MPI_Datatype datatype, MPI_Aint *displacement)	Returns the lower bound of a datatype.
int **MPI_Type_size**(MPI_Datatype datatype, int *size)	Returns the total size (in bytes) of the entries in the datatype identified in the call.
int **MPI_Type_struct**(int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype)	Creates a structured datatype.
int **MPI_Type_ub**(MPI_datatype datatype, MPI_Aint *displacement)	Returns the upper bound of a datatype.
int **MPI_Type_vector**(int count, int blocklength, int stride, MPI_datatype oldtype, MPI_datatype *newtype)	Creates a vector (strided) datatype.
int **MPI_Unpack**(void* inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)	Unpacks a message into the receive buffer.
int **MPI_Wait**(MPI_Request *request, MPI_Status *status)	Waits for an MPI send or receive to complete.
int **MPI_Waitall**(int count, MPI_Request *array_of_requests, MPI_Status *array_of_statuses)	Blocks until all communication operations associated with active handles in the list complete and return their status.
int **MPI_Waitany**(int count, MPI_Request *array_of_requests, int *index, MPI_Status *status)	Blocks until one of the operations associated with the active requests in the array has completed.

Table 11 C version of MPI routines **(continued)**

Routine	Description
int **MPI_Waitsome**(int count, MPI_Request *array_of_requests, int *outcount, int *array_of_indices, MPI_Status *array_of_statuses)	Waits until at least one of the operations associated with active handles in the list have completed.
double **MPI_Wtick**(void)	Returns the resolution of the MPI_Wtime routine in seconds.
double **MPI_Wtime**(void)	Returns a floating-point number of seconds representing wall-clock time.

Fortran 77 version of MPI routines

Table 12 Fortran 77 version of MPI routines

Routine	Description
MPI_Abort (comm, errorcode, ierror) integer comm, errorcode, ierror	Aborts all tasks in a communicator.
MPI_Address(location, address, ierror) *type* location(*) integer address, ierror	Returns the address of a location.
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, ierror) *type* sendbuf(*), recvbuf(*) integer sendcount, sendtype, recvcount, recvtype, comm, ierror	Sends the contents of each process's send buffer to all other processes.
MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm, ierror) *type* sendbuf(*), recvbuf(*) integer sendcount, sendtype, recvcounts(*), displs(*), recvtype, comm, ierror	Sends a varying count of data from each process's send buffer to all other processes.
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror) *type* sendbuf(*), recvbuf(*) integer count, datatype, op, comm, ierror	Combines the elements in the input buffer of each process and returns the combined value to the receive buffer of each process.

Table 12 Fortran 77 version of MPI routines **(continued)**

Routine	Description
MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, ierror) *type* sendbuf(*), recvbuf(*) integer sendcount, sendtype, recvcount, recvtype, comm, ierror	Sends distinct data from each process to all other processes.
MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, ierror) *type* sendbuf(*), recvbuf(*) integer sendcounts(*), sdispls(*), sendtype, recvcounts(*), rdispls(*), recvtype, comm, ierror	Sends distinct data from each process to all other processes. The location of data for the send and the location of the placement of the data on the receive side are specified in the call.
MPI_Attr_delete(comm, keyval, ierror) integer comm, keyval, ierror	Deletes attributes from the cache based upon a key.
MPI_Attr_get(comm, keyval, attribute_val, flag, ierror) integer comm, keyval, attribute_val, ierror logical flag	Receives attribute values based upon a key.
MPI_Attr_put(comm, keyval, attribute_val, ierror) integer comm, keyval, attribute_val, ierror	Stores an attribute value.
MPI_Barrier(comm, ierror) integer comm, ierror	Blocks the calling process until all other processes in the group have called the routine.
MPI_Bcast(buffer, count, datatype, root, comm, ierror) *type* buffer(*) integer count, datatype, root, comm, ierror	Broadcasts a message from the root process to all other processes in the group including itself.
MPI_Bsend(buf, count, datatype, dest, tag, comm, ierror) *type* buf(*) integer counr, datatype, dest, tag, comm, ierror	Sends a message in buffered mode.
MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror) *type* buf(*) integer count, datatype, dest, tag, comm, request, ierror	Creates a persistent communication request for a buffered mode send.
MPI_Buffer_attach(buffer, size, ierror) *type* buffer(*) integer size, ierror	Provides MPI with a buffer in the user's memory that can be used for buffering outgoing messages.

Table 12 Fortran 77 version of MPI routines **(continued)**

Routine	Description
MPI_Buffer_detach(buffer, size, ierror) *type* buffer(*) integer size, ierror	Detaches the buffer currently associated with MPI.
MPI_Cancel(request, ierror) integer request, ierror	Marks a pending, nonblocking send or receive call for cancellation.
MPI_Cart_coords(comm, rank, maxdims, coords, ierror) integer comm, rank, maxdims, coords(*), ierror	Translates process ranks to logical process coordinates as the ranks are used in point-to-point communications.
MPI_Cart_create(comm_old, ndims, dims, periods, reorder, comm_cart, ierror) integer comm_old, ndims, dims(*), comm_cart, ierror logical periods(*), reorder	Returns a handle to a new communicator to which cartesian topology information is attached.
MPI_Cart_get(comm, maxdims, dims, periods, coords, ierror) integer comm, maxdims, dims(*), coords(*), ierror logical periods(*)	Returns the cartesian topology information associated with a communicator.
MPI_Cart_map(comm, ndims, dims, periods, newrank, ierror) integer comm, ndims, dims(*), newrank, ierror logical periods(*)	Computes an optimal placement for the calling process on the physical machine.
MPI_Cart_rank(comm, coords, rank, ierror) integer comm, coords(*), rank, ierror	Translates logical process coordinates to process ranks as the coordinates are used in point-to-point communications.
MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror) integer comm, direction, disp, rank_source, rank_dest, ierror	Returns the shifted source and destination ranks given a shift amount and direction.
MPI_Cart_sub(comm, remain_dims, newcomm, ierror) integer comm, newcomm, ierror logical remain_dims(*)	Partitions the communicator group into subgroups that form lower-dimensional cartesian subgrids.
MPI_Cartdim_get(comm, ndims, ierror) integer comm, ndims, ierror	Returns the cartesian topology information associated with a communicator.
MPI_Comm_compare(comm1, comm2, result, ierror) integer comm1, comm2, result, ierror	Compares two communicators.

Table 12 Fortran 77 version of MPI routines **(continued)**

Routine	Description
MPI_Comm_create(comm, group, newcomm, ierror) integer comm, group, newcomm, ierror	Creates a new communicator and context with a communication group.
MPI_Comm_dup(comm, newcomm, ierror) integer comm, newcomm, ierror	Duplicates the existing communicator with associated key values.
MPI_Comm_free(comm, ierror) integer comm, ierror	Marks the communication object for deallocation.
MPI_Comm_group(comm, group, ierror) integer comm, group, ierror	Returns a handle to the group in the communicator.
MPI_Comm_rank(comm, rank, ierror) integer comm, rank, ierror	Returns the rank of a process in the communicator's group.
MPI_Comm_remote_group(comm, group, ierror) integer comm, group, ierror	Returns the remote group in the intercommunicator.
MPI_Comm_remote_size(comm, size, ierror) integer comm, size, ierror	Returns the size of the remote group in the intercommunicator.
MPI_Comm_size(comm, size, ierror) integer comm, size, ierror	Returns the number of processes involved in a communicator.
MPI_Comm_split(comm, color, key, newcomm, ierror) integer comm, color, key, newcomm, ierror	Partitions the group associated with the communicator into disjoint subgroups.
MPI_Comm_test_inter(comm, flag, ierror) integer comm, ierror logical flag	Determines whether a communicator is an intercommunicator or not.
MPI_Dims_create(nnodes, ndims, dims, ierror) integer nnodes, ndims, dims(*), ierror	Helps select a balanced distribution of processes per coordinate direction.
MPI_Errhandler_create(function, errhandler, ierror) external function integer errhandler, ierror	Registers a user-specified routine for use as an exception handler.
MPI_Errhandler_free(errhandler, ierror) integer errhandler, ierror	Marks an error handler for deallocation.
MPI_Errhandler_get(comm, errhandler, ierror) integer comm, errhandler, ierror	Returns a handle to the error handler that is currently associated with the communicator.

Table 12 Fortran 77 version of MPI routines **(continued)**

Routine	Description
MPI_Errhandler_set(comm, errhandler, ierror) integer comm, errhandler, ierror	Associates a new error handler with the communicator at the calling process.
MPI_Error_class(errorcode, errorclass, ierror) integer errorcode, errorclass, ierror	Maps each standard error code onto itself.
MPI_Error_string(errorcode, string, resultlen, ierror) integer errorcode, resultlen, ierror character*MPI_MAX_ERROR_STRING string	Returns the error string associated with an error code.
MPI_Finalize(ierror) integer ierror	Cleans up all MPI states.
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror) *type* sendbuf(*), recvbuf(*) integer sendcount, sendtype, recvcount, recvtype, root, comm, ierror	Sends the contents of each process's send buffer to the root process.
MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm, ierror) *type* sendbuf(*), recvbuf (*) integer sendcount, sendtype, recvcounts(*), displs(*), recvtype, root, comm, ierror	Sends a varying count of data from each process's send buffer to the root process.
MPI_Get_count(status, datatype, count, ierror) integer status(*), datatype, count, ierror	Returns the number of entries received in the receive buffer.
MPI_Get_elements(status, datatype, elements, ierror) integer status(*), datatype, elements, ierror	Returns the number of elements in a datatype.
MPI_Get_processor_name(name, resultlen, ierror) character*MPI_MAX_PROCESSOR_NAME name integer resultlen, ierror	Returns the name of the processor on which it was called at the moment of the call.
MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph, ierror) integer comm_old, nnodes, index(*), edges(*), comm_graph, ierror logical reorder	Returns a handle to a new communicator to which the graph topology information is attached.

Table 12 Fortran 77 version of MPI routines **(continued)**

Routine	Description
MPI_Graph_get(comm, maxindex, maxedges, index, edges, ierror) integer comm, maxindex, maxedges, index(*), edges(*), ierror	Retrieves graph topology information associated with a communicator.
MPI_Graph_map(comm, nnodes, index, edges, newrank, ierror) integer comm, nnodes, index(*), edges(*), newrank, ierror	Maps process to graph topology information.
MPI_Graph_neighbors(comm, rank, maxneighbors, neighbors, ierror) integer comm, rank, maxneighbors, neighbors(*), ierror	Returns the neighbors of a node associated with a graph topology.
MPI_Graph_neighbors_count(comm, rank, nneighbors, ierror) integer comm, rank, nneighbors, ierror	Returns the number of neighbors of a node associated with a graph topology.
MPI_Graphdims_get(comm, nnodes, nedges, ierror) integer comm, nnodes, nedges, ierror	Retrieves graph topology information associated with a communicator.
MPI_Group_compare(group1,group2, result, ierror) integer group1, group2, result, ierror	Returns the relationship between two groups.
MPI_Group_difference(group1, group2, newgroup, ierror) integer group1, group2, newgroup, ierror	Creates a group by computing the difference between two existing groups.
MPI_Group_excl(group, n, ranks, newgroup, ierror) integer group, n, ranks(*), newgroup, ierror	Creates a group by reordering an existing group and only taking unlisted members.
MPI_Group_free(group, ierror) integer group, ierror	Marks a group object for deallocation.
MPI_Group_incl(group, n, ranks, newgroup, ierror) integer group, n, ranks(*), newgroup, ierror	Creates a group by reordering an existing group and only taking listed members.
MPI_Group_intersection(group1, group2, newgroup, ierror) integer group1, group2, newgroup, ierror	Creates a group by computing the intersection of two existing groups.
MPI_Group_range_excl(group, n, ranges, newgroup, ierror) integer group, n, ranges(3, *), newgroup, ierror	Creates a group by excluding ranges of processes from an existing group.
MPI_Group_range_incl(group, n, ranges, newgroup, ierror) integer group, n, ranges(3, *), newgroup, ierror	Creates a group from ranges of ranks in an existing group.

Table 12 Fortran 77 version of MPI routines **(continued)**

Routine	Description
MPI_Group_rank(group, rank, ierror) integer group, rank, ierror	Returns the rank of this process in a group.
MPI_Group_size(group, size, ierror) integer group, size, ierror	Returns the size or number of processes in the group.
MPI_Group_translate_ranks(group1, n, ranks1, group2, ranks2, ierror) integer group1, n, ranks1(*), group2, ranks2(*), ierror	Determines the relative numbering of the same processes in two different groups.
MPI_Group_union(group1, group2, newgroup, ierror) integer group1, group2, newgroup, ierror	Creates a group by computing the union of two existing groups.
MPI_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror) *type* buf(*) integer count, datatype, dest, tag, comm, request, ierror	Starts a buffered mode, nonblocking send.
MPI_Init(ierror) integer ierror	Initializes the MPI execution environment.
MPI_Initialized(flag, ierror) logical flag integer ierror	Determines whether MPI_Init has been called.
MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader, tag, newintercomm, ierror) integer local_comm, local_leader, peer_comm, remote_leader, tag, newintercomm, ierror	Creates an intercommunicator.
MPI_Intercomm_merge(intercomm, high, newintracomm, ierror) logical high integer intercomm, newintracomm, ierror	Creates an intracommunicator from the union of two groups.
MPI_Iprobe(source, tag, comm, flag, status, ierror) logical flag integer source, tag, comm, status (*), ierror	Returns flag = true if there is a message that can be received and that matches the pattern specified by the arguments.
MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror) *type* buf(*) integer count, datatype, source, tag, comm, request, ierror	Starts a nonblocking receive.

Table 12 Fortran 77 version of MPI routines **(continued)**

Routine	Description
MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror) *type* buf(*) integer count, datatype, dest, tag, comm, request, ierror	Starts a ready mode, nonblocking send.
MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror) *type* buf (*) integer count, datatype, dest, tag, comm, request, ierror	Starts a standard mode, nonblocking send.
MPI_Issend(buf, count, datatype, dest, tag, comm, request, ierror) *type* buf(*) integer count, datatype, dest, tag, comm, request, ierror	Starts a synchronous mode, nonblocking send.
MPI_Keyval_create(copy_fn, delete_fn, keyval, extra_state, ierror) external copy_fn, delete_fn integer keyval, extra_state, ierror	Generates a new attribute key.
MPI_Keyval_free(keyval, ierror) integer keyvak, ierror	Frees an existing attribute key.
MPI_Op_create(function, commute, op, ierror) external function logical commute integer op, ierror	Binds a user-defined global operation to a handle.
MPI_Op_free(op, ierror) integer op, ierror	Marks a user-defined reduction operation for deallocation.
MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror) *type* inbuf(*), outbuf(*) integer incount, datatype, outsize, position, comm, ierror	Packs the message in the send buffer into the specified buffer space.
MPI_Pack_size(incount, datatype, comm, size, ierror) integer incount, datatype, comm, size, ierror	Returns the maximum number of buffer bytes required to hold the data.
MPI_Pcontrol(level) integer level	Calls the specified profiling package.

Table 12 Fortran 77 version of MPI routines **(continued)**

Routine	Description
MPI_Probe(source, tag, comm, status, ierror) integer source, tag, comm, status(*), ierror	Returns only after a message is found that matches the pattern specified by the arguments.
MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror) *type* buf(*) integer count, datatype, source, tag, comm, status(*), ierror	Starts a blocking receive.
MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror) *type* buf(*) integer count, datatype, source, tag, comm, request, ierror	Creates a persistent communication request for a receive operation.
MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror) *type* sendbuf(*), recvbuf(*) integer count, datatype, op, root, comm, ierror	Combines the elements in the input buffer of each process in the group and returns the combined value in the output buffer of the root process.
MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm, ierror) *type* sendbuf(*), recvbuf(*) integer recvcounts(*), datatype, op, comm, ierror	Combines values and scatters the results.
MPI_Request_free(request, ierror) integer request, ierror	Marks the request object for deallocation.
MPI_Rsend(buf, count, datatype, dest, tag, comm, ierror) *type* buf(*) integer count, datatype, dest, tag, comm, ierror	Sends a message in ready mode.
MPI_Rsend_init(buf, count, datatype, dest, tag, comm, request, ierror) *type* buf(*) integer count, datatype, dest, tag, comm, request, ierror	Creates a persistent communication object for a ready mode send operation.
MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror) *type* sendbuf(*), recvbuf(*) integer count, datatype, op, comm, ierror	Performs a prefix reduction on data distributed across the group.

Table 12 Fortran 77 version of MPI routines **(continued)**

Routine	Description
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror) *type* sendbuf(*), recvbuf(*) integer sendcount, sendtype, recvcount, recvtype, root, comm, ierror	Sends the contents of the root process's send buffer to other processes in the group.
MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror) *type* sendbuf(*), recvbuf(*) integer sendcounts(*), displs(*), sendtype, recvcount, recvtype, root, comm, ierror	Sends a varying count of data from the root process's send buffer to other processes in the group.
MPI_Send(buf, count, datatype, dest, tag, comm, ierror) *type* buf(*) integer count, datatype, dest, tag, comm, ierror	Starts a standard mode, blocking send.
MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror) *type* buf(*) integer count, datatype, dest, tag, comm, request, ierror	Creates a persistent communication request for a standard mode send operation and binds it to all the arguments of a send operation.
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status, ierror) *type* sendbuf(*), recvbuf(*) integer sendcount, sendtype, dest, sendtag, recvcount, recvtype, source, recvtag, comm, status(*), ierror	Executes a blocking send and receive.
MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, revctag, comm, status, ierror) *type* buf(*) integer count, datatype, dest, sendtag, source, recvtag, comm, status(*), ierror	Executes a blocking send and receive where both operations use the same buffer.
MPI_Ssend(buf, count, datatype, dest, tag, comm, ierror) *type* buf(*) integer count, datatype, dest, tag, comm, ierror	Sends a message in synchronous mode.
MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror) *type* buf(*) integer count, datatype, dest, tag, comm, request, ierror	Creates a persistent communication object for a synchronous mode send operation.

Table 12 Fortran 77 version of MPI routines **(continued)**

Routine	Description
MPI_Start(request, ierror) integer request, ierror	Initiates a communication with a persistent request handle.
MPI_Startall(count, array_of_requests, ierror) integer count, array_of_requests(*), ierror	Initiates a collection of requests.
MPI_Test(request, flag, status, ierror) logical flag integer request, status(*), ierror	Returns flag = true if the request (operation) identified in the call is complete.
MPI_Test_cancelled(status, flag, ierror) logical flag integer status(*), ierror	Marks a pending, nonblocking communication operation (send or receive) for cancellation.
MPI_Testall(count, array_of_requests, flag, array_of_statuses, ierror) logical flag integer count, array_of_requests(*), array_of_statuses(MPI_STATUS_SIZE, *), ierror	Returns flag = true if all communications associated with active handles in the array have completed.
MPI_Testany(count, array_of_requests, index, flag, status, ierror) logical flag integer count, array_of_requests(*), index, status(*), ierror	Tests for completion of either one or none of the operations associated with active handles.
MPI_Testsome(incount, array_of_requests, outcount, array_of_indices, array_of_statuses, ierror) integer incount, array_of_requests(*), outcount, array_of_indices(*), array_of_statuses(MPI_STATUS_SIZE, *), ierror	Tests for some given communications to complete.
MPI_Topo_test(comm, status, ierror) integer comm, status, ierror	Returns the type of topology that is assigned to a communicator.
MPI_Type_commit(datatype, ierror) integer datatype, ierror	Commits the datatype (that is, the formal description of a communication buffer), not the content of that buffer.
MPI_Type_contiguous(count, oldtype, newtype, ierror) integer count, oldtype, newtype, ierror	Creates a contiguous datatype.
MPI_Type_extent(datatype, extent, ierror) integer datatype, extent, ierror	Returns the extent of a datatype.

Table 12 Fortran 77 version of MPI routines **(continued)**

Routine	Description
MPI_Type_free(datatype, ierror) integer datatype, ierror	Marks the datatype object identified in the call for deallocation.
MPI_Type_hindexed(count, array_of_blocklengths, array_of_displacements, oldtype, newtype, ierror) integer count, array_of_blocklengths(*), array_of_displacements(*), oldtype, newtype, ierror	Creates an indexed datatype with offsets in bytes.
MPI_Type_hvector(count, blocklength, stride, oldtype, newtype, ierror) integer count, blocklength, stride, oldtype, newtype, ierror	Creates a vector (strided) datatype with offset in bytes.
MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements, oldtype, newtype, ierror) integer count, array_of_blocklengths(*), array_of_displacements(*), oldtype, newtype, ierror	Creates an indexed datatype.
MPI_type_lb(datatype, displacement, ierror) integer datatype, displacement, ierror	Returns the lower bound of a datatype.
MPI_Type_size(datatype, size, ierror) integer datatype, size, ierror	Returns the total size (in bytes) of the entries in the datatype identified in the call.
MPI_Type_struct(count, array_of_blocklengths, array_of_displacements, array_of_types, newtype, ierror) integer count, array_of_blocklenghts(*), array_of_displacements(*), array_of_types(*), newtype, ierror	Creates a structured datatype.
MPI_Type_ub(datatype, displacement, ierror) integer datatype, displacement, ierror	Returns the upper bound of a datatype.
MPI_Type_vector(count, blocklength, stride, oldtype, newtype, ierror) integer count, blocklength, stride, oldtype, newtype, ierror	Creates a vector (strided) datatype.
MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm, ierror) *type* inbuf(*), outbuf(*) integer insize, position, outcount, datatype, comm, ierror	Unpacks a message into the receive buffer.
MPI_Wait(request, status, ierror) integer request, status(*), ierror	Waits for an MPI send or receive to complete.

Table 12 Fortran 77 version of MPI routines **(continued)**

Routine	Description
MPI_Waitall(count, array_of_requests, array_of_statuses, ierror) integer count, array_of_requests(*), array_of_statuses(MPI_STATUS_SIZE,*), ierror	Blocks until all communication operations associated with active handles in the list complete and return their status.
MPI_Waitany(count, array_of_requests,index, status, ierror) integer count, array_of_requests(*), index, status(*), ierror	Blocks until one of the operations associated with the active requests in the array has completed.
MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices, array_of_statuses, ierror) integer incount, array_of_requests(*), outcount, array_of_indices(*), array_of_statuses(MPI_STATUS_SIZE,*), ierror	Waits until at least one of the operations associated with active handles in the list have completed.
double precision **MPI_Wtick**()	Returns the resolution of the MPI_Wtime routine in seconds.
double precision **MPI_Wtime**()	Returns a floating-point number of seconds representing wall-clock time since some time in the past.

C version of HP MPI library extensions

Table 13 C version of HP MPI library extensions

Routine	Description
int **MPIHP_Comm_id**(MPI_Comm comm, int *id)	Returns the communicator context ID
int **MPIHP_Trace_off**(void)	Stops HP MPI application tracing
int **MPIHP_Trace_on**(void)	Starts HP MPI application tracing

Fortran 77 version of HP MPI library extensions

Table 14 Fortran 77 version of HP MPI library extensions

Routine	Description
subroutine **MPIHP_Comm_id**(comm, id, ierr) integer comm, id, ierr	Returns the communicator context ID
subroutine **MPIHP_Trace_off**(ierr) integer ierr	Stops HP MPI application tracing
subroutine **MPIHP_Trace_on**(ierr) integer ierr	Starts HP MPI application tracing

Example applications

B

This appendix provides example applications that supplement the information in "MPI concepts" on page 2. The examples included are shown in Table 15.

Table 15 Example applications shipped with HP MPI

Name	Language	Description	-np argument
send_receive.f	Fortran 77	Illustrates a simple send and receive operation.	-np >= 2
ping_pong.c	C	Measures the time it takes to send and receive data between two processes.	-np = 2
compute_pi.f	Fortran 77	Computes pi by integrating $f(x)=4/(1+x^2)$.	-np >= 1
master_worker.f90	Fortran 90	Distributes sections of an array and performs computation on all sections in parallel.	-np >= 2
sort.C, entry.C, entry.H, sort_input0, sort_input1	C++	Sorts elements from two data files in ascending order.	-np = 2
communicator.c	C	Copies the default communicator MPI_COMM_WORLD.	-np = 2
multi_par.c	C	Illustrates how to use multiple threads with the standard hello_world application. This example can only be run on S- and X-Class servers.	-np >= 2

These examples and their make file are located in the /opt/mpi/help subdirectory. The examples are presented for illustration purposes only. They may not necessarily represent the most efficient way to solve a given problem.

To build and run the examples (except for multi_par.c):

Step 1 Edit the .cshrc file and add the /opt/mpi/bin subdirectory to your path statement.

Step 2 Change to a writable directory.

Step 3 Enter
% `cp /opt/mpi/help/* .`

Step 4 Enter `make` to build all the examples or `make` *example_name* to build a specific application example.

Step 5 Enter
% `mpirun -j -w -np #` *program*

where *program* specifies the path to the executable created in step 4.

For the multi_par.c example, follow steps 1 through 3 and step 5. To build multi_par.c as in step 4, enter

% `mpicc +O3 +Oparallel -o multi_par multi_par.c`

send_receive.f

In this Fortran 77 example, process 0 sends an array to other processes in the default communicator MPI_COMM_WORLD.

```
program main

include 'mpif.h'

integer rank, size, to, from, tag, count, i, ierr
integer src, dest
integer st_source, st_tag, st_count
integer status(MPI_STATUS_SIZE)
double precision data(100)

call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)

if (size .eq. 1) then
        print *, 'must have at least 2 processes'
        call MPI_Finalize(ierr)
        stop
endif

print *, 'Process ', rank, ' of ', size, ' is alive'
```

```
 dest = size - 1
 src = 0

 if (rank .eq. src) then
         to = dest
         count = 10
         tag = 2001

         do i=1, 10
                 data(i) = 1
         enddo

         call MPI_Send(data, count, MPI_DOUBLE_PRECISION,
+                      to, tag, MPI_COMM_WORLD, ierr)
 endif

 if (rank .eq. dest) then
         tag = MPI_ANY_TAG
         count = 10
         from = MPI_ANY_SOURCE
         call MPI_Recv(data, count, MPI_DOUBLE_PRECISION,
+                      from, tag, MPI_COMM_WORLD, status, ierr)
         call MPI_Get_Count(status, MPI_DOUBLE_PRECISION,
+                           st_count, ierr)
         st_source = status(MPI_SOURCE)
         st_tag = status(MPI_TAG)

         print *, 'Status info: source = ', st_source,
+                ' tag = ', st_tag, ' count = ', st_count
         print *, rank, ' received', (data(i),i=1,10)
 endif

 call MPI_Finalize(ierr)

 stop
 end
```

send_receive output

The output from running the send_receive executable is shown below. The application was run with -np = 10.

```
Process  0 of  10 is alive
Process  1 of  10 is alive
Process  3 of  10 is alive
Process  5 of  10 is alive
Process  9 of  10 is alive
```

```
Process  2 of  10 is alive
Status info: source =  0 tag =  2001 count =  10
9 received 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
Process  4 of  10 is alive
Process  7 of  10 is alive
Process  8 of  10 is alive
Process  6 of  10 is alive
```

ping_pong.c

This C example is used as a performance benchmark to measure the amount of time it takes to send and receive data between two processes.

Define the CHECK macro to check data integrity and the SPPUX macro for better timing on SPP-UX.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

#ifdef SPPUX
#include <sys/cnx_ail.h>
#define MPI_Wtime()      ((double) cit_read())
#define unit_per_sec()   ((double) cit_per_sec_read())
#else
#define unit_per_sec()   ((double) 1)
#endif

#define NLOOPS           1000
#define ALIGN            4096

/*
 * local functions
 */
static int               compare();
/*
 * static variables
 */
static double            vtmp[NLOOPS];
static unsigned int      max_ui = ~0;

main(argc, argv)

int                      argc;
char                     *argv[];

{
        int              i, j;
```

```
        double          start, stop;
        double          ovrhd, min, med, max, unit_per_usec;
        int             nbytes = 0;
        int             rank, size;
        MPI_Status      status;
        char            *buf;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        if (size != 2) {
                if ( ! rank) printf("ping_pong: must have two processes\n");
                MPI_Finalize();
                exit(0);
        }

        nbytes = (argc > 1) ? atoi(argv[1]) : 0;
        if (nbytes < 0) nbytes = 0;
/*
 * Page-align buffers and displace them in the cache to avoid collisions.
 */
        buf = (char *) malloc(nbytes + 524288 + (ALIGN - 1));
        if (buf == 0) {
                MPI_Abort(MPI_COMM_WORLD, MPI_ERR_BUFFER);
                exit(1);
        }

        buf = (char *) ((((unsigned long) buf) + (ALIGN - 1)) & ~(ALIGN - 1));
        if (rank == 1) buf += 524288;
        memset(buf, 0, nbytes);
/*
 * Ping-pong.
 */
        if (rank == 0) {
                printf("ping-pong %d bytes ...\n", nbytes);

                for (i = 0; i < NLOOPS; i++) vtmp[i] = 0;

                ovrhd = max_ui;
                unit_per_usec = unit_per_sec() / 1000000.;

                for (i = 0; i < NLOOPS; i++) {
                        start = MPI_Wtime();
                        stop = MPI_Wtime();

                        if (stop > start) {
                                stop = stop - start;
                        } else {
                                stop = max_ui + stop - start + 1;
                        }
```

```
                        if (stop < ovrhd) ovrhd = stop;
                }
/*
 * warm-up loop
 */
                for (i = 0; i < 5; i++) {
                        MPI_Send(buf, nbytes, MPI_CHAR, 1, 1, MPI_COMM_WORLD);
                        MPI_Recv(buf, nbytes, MPI_CHAR,
                                        1, 1, MPI_COMM_WORLD, &status);
                }
/*
 * timing loop
 */
                for (i = 0; i < NLOOPS; i++) {
#ifdef CHECK
                        for (j = 0; j < nbytes; j++) buf[j] = (char) (j + i);
#endif
                        start = MPI_Wtime();

                        MPI_Send(buf, nbytes, MPI_CHAR,
                                        1, 1000 + i, MPI_COMM_WORLD);
#ifdef CHECK
                        memset(buf, 0, nbytes);
#endif
                        MPI_Recv(buf, nbytes, MPI_CHAR,
                                        1, 2000 + i, MPI_COMM_WORLD, &status);

                        stop = MPI_Wtime();
#ifdef CHECK
                        for (j = 0; j < nbytes; j++) {
                                if (buf[j] != (char) (j + i)) {
                                        printf("error: buf[%d] = %d, not %d\n",
                                                j, buf[j], j = i);
                                        break;
                                }
                        }
#endif

                        vtmp[i] = (stop - start - ovrhd) / 2 / unit_per_usec;
                }

                qsort(vtmp, NLOOPS, sizeof(double), compare);

                min = vtmp[0];
                med = vtmp[NLOOPS / 2];
                max = vtmp[NLOOPS - 1];

                printf("%d bytes: %.2f %.2f %.2f usec/msg\n",
                        nbytes, min, med, max);
                if (nbytes > 0) {
                        printf("%d bytes: %.2f %.2f %.2f MB/sec\n", nbytes,
                                nbytes / min, nbytes / med, nbytes / max);
```

```
                }
        }
        else {
/*
 * warm-up loop
 */
                for (i = 0; i < 5; i++) {
                        MPI_Recv(buf, nbytes, MPI_CHAR,
                                        0, 1, MPI_COMM_WORLD, &status);
                        MPI_Send(buf, nbytes, MPI_CHAR, 0, 1, MPI_COMM_WORLD);
                }

                for (i = 0; i < NLOOPS; i++) {
                        MPI_Recv(buf, nbytes, MPI_CHAR,
                                        0, 1000 + i, MPI_COMM_WORLD, &status);
                        MPI_Send(buf, nbytes, MPI_CHAR,
                                        0, 2000 + i, MPI_COMM_WORLD);
                }
        }

        MPI_Finalize();
        exit(0);
}

/*
 *      compare
 *
 *      Function:       - compare two doubles
 *      Accepts:        - ptr to two doubles
 *      Returns:        - -1/0/1
 */
static int
compare(p1, p2)
double                  *p1, *p2;

{
        return( (*p1 > *p2) ? 1 : ( (*p1 < *p2) ? -1 : 0) );
}
```

ping_pong output

The output from running the ping_pong executable is shown below. The application was run with -np = 2.

```
ping-pong 0 bytes ...
0 bytes: 2.98 3.99 34.99 usec/msg
```

compute_pi.f

This Fortran 77 example computes pi by integrating $f(x) = 4/(1 + x^2)$. Each process:

- Receives the number of intervals used in the approximation
- Calculates the areas of its rectangles
- Synchronizes for a global summation

Process 0 prints the result and the time it took to complete the calculation.

```
      program main

      include 'mpif.h'

      double precision PI25DT
      parameter(PI25DT = 3.14159265358979323846264338327d0)

      double precision  mypi, pi, h, sum, x, f, a
      integer n, myid, numprocs, i, ierr
C
C Function to integrate
C
      f(a) = 4.d0 / (1.d0 + a*a)

      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
      print *, "Process ", myid, " of ", numprocs, " is alive"

      sizetype = 1
      sumtype = 2

      if (myid .eq. 0) then
              n = 100
      endif

      call MPI_BCAST(n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
C
C Calculate the interval size.
C
      h = 1.0d0 / n
      sum  = 0.0d0

      do 20 i = myid + 1, n, numprocs
              x = h * (dble(i) - 0.5d0)
              sum = sum + f(x)
 20   continue

      mypi = h * sum
C
C Collect all the partial sums.
C
```

```
      call MPI_REDUCE(mypi, pi, 1, MPI_DOUBLE_PRECISION,
     +                MPI_SUM, 0, MPI_COMM_WORLD, ierr)
C
C Process 0 prints the result.
C
      if (myid .eq. 0) then
             write(6, 97) pi, abs(pi - PI25DT)
 97          format('  pi is approximately: ', F18.16,
     +              '  Error is: ', F18.16)
      endif

      call MPI_FINALIZE(ierr)

      stop
      end
```

compute_pi output

The output from running the compute_pi executable is shown below. The application was run with -np = 10.

```
Process  0 of  10 is alive
Process  1 of  10 is alive
Process  3 of  10 is alive
Process  9 of  10 is alive
Process  7 of  10 is alive
Process  5 of  10 is alive
Process  6 of  10 is alive
Process  2 of  10 is alive
Process  4 of  10 is alive
Process  8 of  10 is alive
pi is approximately: 3.1416009869231250
Error is:  .0000083333333318
```

master_worker.f90

In this Fortran 90 example, a master task initiates (numtasks - 1) number of worker tasks. The master distributes an equal portion of an array to each worker task. Each worker task receives its portion of the array and sets the value of each element to (the element's index + 1). Each worker task then sends its portion of the modified array back to the master.

```
program array_manipulation
  include 'mpif.h'

  integer (kind=4) :: status(MPI_STATUS_SIZE)
  integer (kind=4), parameter :: ARRAYSIZE = 10000, MASTER = 0
  integer (kind=4) :: numtasks, numworkers, taskid, dest, index, i
  integer (kind=4) :: arraymsg, indexmsg, source, chunksize, int4, real4
  real (kind=4) :: data(ARRAYSIZE), result(ARRAYSIZE)
  integer (kind=4) :: numfail

  call MPI_Init(ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD, taskid, ierr)
  call MPI_Comm_size(MPI_COMM_WORLD, numtasks, ierr)
  numworkers = numtasks - 1
  chunksize = (ARRAYSIZE / numworkers)
  arraymsg = 1
  indexmsg = 2
  int4 = 4
  real4 = 4
  numfail = 0

! ****************************** Master task ******************************
  if (taskid .eq. MASTER) then
     data = 0.0
     index = 1
     do dest = 1, numworkers
        call MPI_Send(index, 1, MPI_INTEGER, dest, 0, MPI_COMM_WORLD, ierr)
        call MPI_Send(data(index), chunksize, MPI_REAL, dest, 0, &
             MPI_COMM_WORLD, ierr)
        index = index + chunksize
     end do

     do i = 1, numworkers
        source = i
        call MPI_Recv(index, 1, MPI_INTEGER, source, 1, MPI_COMM_WORLD, &
             status, ierr)
        call MPI_Recv(result(index), chunksize, MPI_REAL, source, 1, &
             MPI_COMM_WORLD, ststus, ierr)
     end do

     do i = 1, numworkers*chunksize
        if (result(i) .ne. (i+1)) then
           print *, 'element ', i, ' expecting ', (i+1), ' actual is ', result(i)
           numfail = numfail + 1
        endif
```

```
      enddo

      if (numfail .ne. 0) then
         print *, 'out of ', ARRAYSIZE, ' elements, ', numfail, ' wrong answers'
      else
         print *, 'correct results!'
      endif
   end if

! ****************************** Worker task ********************************
   if (taskid .gt. MASTER) then
      call MPI_Recv(index, 1, MPI_INTEGER, MASTER, 0, MPI_COMM_WORLD, &
           status, ierr)
      call MPI_Recv(result(index), chunksize, MPI_REAL, MASTER, 0, &
           MPI_COMM_WORLD, status, ierr)

      do i = index, index + chunksize
         result(i) = i + 1
      end do

      call MPI_Send(index, 1, MPI_INTEGER, MASTER, 1, MPI_COMM_WORLD, ierr)
      call MPI_Send(result(index), chunksize, MPI_REAL, MASTER, 1, &
           MPI_COMM_WORLD, ierr)
   end if
   call MPI_Finalize(ierr)

end program array_manipulation
```

master_worker output

The output from running the master_worker executable is shown below. The application was run with `-np = 2`.

```
correct results!
```

sort.C

In this C++ example, there are two processes. The processes do an ascending sort on the data elements from two input files. After completing the sort, the processes synchronize using `MPI_Barrier`. The sorted output is then partitioned equally and stored in sort_output_0 and sort_output_1 in ascending order.

Each process reads in data from its own input file. Process 0 reads from sort_input0 and process 1 reads from sort_input1.

For the sort application to work correctly, each input file must have an even number of data elements (ten elements each for this example).

The sort application contains the entry.H, entry.C, and sort.C files. entry.H contains the C++ class definitions. entry.C contains the class attributes and methods. sort.C contains logic to perform the sort computation.

entry.H source file

```
#include <iostream.h>
#include <fstream.h>

class Entry {
private:
        int value;
public:
        Entry() { value = 0; }
        Entry(int x) { value = x; }
        Entry(const Entry &e) { value = e.getValue(); }
        Entry& operator= (const Entry &e)
                { value = e.getValue(); return (*this); }
        int getValue() const { return value; }
        int operator> (const Entry &e) const { return (value > e.getValue()); }
        friend ostream& operator<< (ostream &os, const Entry &e)
                { return (os << e.value); }
        friend istream& operator>> (istream &is, Entry &e)
                { return (is >> e.value); }
};

class BlockOfEntries {
private:
        Entry **entries;
        int numOfEntries;

        ifstream inFile;
        ofstream outFile;
public:
        BlockOfEntries(char *infname, char *outfname, int *numOfEntries_p);
        ~BlockOfEntries();

        int getnumOfEntries() { return numOfEntries; }
```

```
        void setLeftShadow(const Entry &e) { *(entries[0]) = e; }
        void setRightShadow(const Entry &e)
                { *(entries[numOfEntries - 1]) = e; }

        const Entry& getLeftEnd() { return *(entries[1]); }
        const Entry& getRightEnd() { return *(entries[numOfEntries - 2]); }

        void singleStepOddEntries();
        void singleStepEvenEntries();
};
```

entry.C source file

```
#include <limits.h>
#include "entry.H"

const Entry MAXENTRY(INT_MAX);
const Entry MINENTRY(INT_MIN);

BlockOfEntries::BlockOfEntries(char *infname, char *outfname,
                               int *numOfEntries_p) :
  inFile(infname, ios::in), outFile(outfname, ios::out)

{
        inFile >> numOfEntries;                 // read # entries
        *numOfEntries_p = numOfEntries;
        numOfEntries += 2;                      // add left/right shadows
        entries = new Entry *[numOfEntries];    // allocate space for entries

// read all the entries
        for(int i = 1; i < numOfEntries - 1; i++) {
                entries[i] = new Entry;         // use default constructor
                inFile >> *(entries[i]);        // use overloaded global
                                                // operator>>
        }

// initialize shadow entries
        entries[0] = new Entry(MINENTRY);
        entries[numOfEntries - 1] = new Entry(MAXENTRY);
}

BlockOfEntries::~BlockOfEntries()

{
// output number of entries and the sorted entries; delete them too
        outFile << numOfEntries-2 << endl;
        for(int i = 1; i < numOfEntries - 1; i++) {
                outFile << *(entries[i]) << endl; // use overloaded global
                                                  // operator<<
                delete entries[i];
```

```
        }

// delete shadows and array
        delete entries[0];
        delete entries[numOfEntries-1];
        delete [] entries;
}

void BlockOfEntries::singleStepOddEntries()

{
        for(int i = 0; i < numOfEntries - 1; i += 2) {
// use Entry::operator>
                if (*(entries[i]) > *(entries[i + 1]) ) {
                        Entry *temp = entries[i + 1];
                        entries[i + 1] = entries[i];
                        entries[i] = temp;
                }
        }
}

void BlockOfEntries::singleStepEvenEntries()

{
        for(int i = 1; i < numOfEntries - 2; i += 2) {
// use Entry::operator>
                if (*(entries[i]) > *(entries[i + 1]) ) {
                        Entry *temp = entries[i + 1];
                        entries[i + 1] = entries[i];
                        entries[i] = temp;
                }
        }
}
```

sort.C source file

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include "entry.H"
#include <mpi.h>

main(int argc, char **argv)

{
        int             iMyRank, iNumProcesses;

        MPI_Init(&argc, &argv);

        if (argc != 2) {
```

```
                cout << "Usage: "
                        << argv[0]
                        << " <base filename> "
                        << endl;
                MPI_Finalize();
                exit(1);
        }

        MPI_Comm_rank(MPI_COMM_WORLD, &iMyRank);
        MPI_Comm_size(MPI_COMM_WORLD, &iNumProcesses);

        char infname[256], outfname[256];

        sprintf(infname, "%s%d", argv[1], iMyRank);
        sprintf(outfname, "sort_output_%d", argv[1], iMyRank);

// read in the block of entries
        int numEntries;

        BlockOfEntries *aBlock = new
                        BlockOfEntries(infname, outfname, &numEntries);
        numEntries *= iNumProcesses;            // Get total # entries.

        for(int j = 0; j < numEntries / 2; j++) {
// synchronize before updating shadow entries
                MPI_Barrier(MPI_COMM_WORLD);
// update shadow entries
                int iRecvVal, iSendVal;
                MPI_Request RequestSortRequest;
                MPI_Status StatusStatus;
// everyone except iNumProcesses-1 posts a receive for
// rightShadow from right
                if (iMyRank != (iNumProcesses - 1)) {
                        MPI_Irecv(&iRecvVal, 1, MPI_INT, iMyRank+1,
                                        MPI_ANY_TAG, MPI_COMM_WORLD,
                                        &RequestSortRequest);
                }

// everyone except 0 sends leftEnd to left
                if (iMyRank != 0) {
                        iSendVal = aBlock->getLeftEnd().getValue();
                        MPI_Send(&iSendVal, 1, MPI_INT,
                                        iMyRank - 1, 1, MPI_COMM_WORLD);
                }

                if (iMyRank != (iNumProcesses - 1)) {
// Wait for receive to complete.
                        MPI_Wait(&RequestSortRequest, &StatusStatus);
                        aBlock->setRightShadow(Entry(iRecvVal));
                }

// everyone except 0 posts a receive for leftShadow from left
```

```
                if (iMyRank != 0) {
                        MPI_Irecv(&iRecvVal, 1, MPI_INT, iMyRank-1,
                                        MPI_ANY_TAG, MPI_COMM_WORLD,
                                        &RequestSortRequest);
                }

// everyone except iNumProcesses-1 sends rightEnd to right
                if (iMyRank != (iNumProcesses - 1)) {
                        iSendVal = aBlock->getRightEnd().getValue();
                        MPI_Send(&iSendVal, 1, MPI_INT,
                                        iMyRank + 1, 1, MPI_COMM_WORLD);
                }

                if (iMyRank != 0) {
// Wait for receive to complete.
                        MPI_Wait(&RequestSortRequest, &StatusStatus);
                        aBlock->setLeftShadow(Entry(iRecvVal));
                }

                aBlock->singleStepOddEntries();
                aBlock->singleStepEvenEntries();
        }

        delete aBlock;          // output sorted entries and delete the block

// Wait for everyone to complete.
        MPI_Barrier(MPI_COMM_WORLD);

        MPI_Finalize();
        exit(0);
}
```

sort output

The output from the sort executable is stored in sort_output_0 and sort_output_1 in ascending order and is shown in Table 16. The application was run with `-np = 2`.

Table 16 Output from running the sort executable

Output file	Contents
sort_output_0	-6524, -134, -64, -6, -1, 0, 3, 6, 34, 86
sort_output_1	234, 345, 635, 653, 765, 2345, 2345, 2345, 5634, 8763

communicator.c

This C example shows how to make a copy of the default communicator MPI_COMM_WORLD.

```
#include <stdio.h>
#include <mpi.h>

main(argc, argv)

int                     argc;
char                    *argv[];

{
        int             rank, size, data;
        MPI_Status      status;
        MPI_Comm        libcomm;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        MPI_Comm_dup(MPI_COMM_WORLD, &libcomm);

        if (rank == 0) {
                data = 12345;
                MPI_Send(&data, 1, MPI_INT, 1, 5, MPI_COMM_WORLD);
                data = 6789;
                MPI_Send(&data, 1, MPI_INT, 1, 5, libcomm);
        } else {
                MPI_Recv(&data, 1, MPI_INT, 0, 5, libcomm, &status);
                printf("received libcomm data = %d\n", data);
                MPI_Recv(&data, 1, MPI_INT, 0, 5, MPI_COMM_WORLD, &status);
                printf("received data = %d\n", data);
        }

        MPI_Comm_free(&libcomm);
        MPI_Finalize();
        exit(0);
}
```

communicator output

The output from running the communicator executable is shown below. The application was run with `-np = 2`.

```
received libcomm data = 6789
received data = 12345
```

multi_par.c

This C example shows how to use multiple threads with the standard hello_world application. When the multi_par executable is run, two processes fork and create two threads each. The example can only be run on S- and X-Class servers.

```
#include <mpi.h>
#include <sys/cnx_ail.h>


/*
 * constants and macros
 */

#define SIZE    20
#define TIMES   2

main ( argc, argv )

int             argc;
char            **argv;

{
        int     rank, size;
        int     array[SIZE];
        int     i,j;
        int     fl_et = 1;

        MPI_Init(&argc,&argv);

        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        printf("rank:%d is running on node:%d\n",
                        rank, current_node());


        for (j = 1; j < TIMES; ++j ) {
/*
 * the pragma
 * each MPI process goes two way parallel
 */
#pragma _CNX loop_parallel(ivar = i , max_threads=2)
                for ( i = 0; i < SIZE; ++i ) {
                        printf("hello, I'm rank %d out of %d: thread %d\n",
                                                rank, size, cps_stid());
                        array[i] = i%j;
                }
        }

        MPI_Finalize();
}
```

multi_par output

The output from running the multi_par executable is shown below. The application was run with -np = 2.

```
rank:0 is running on node:0
rank:1 is running on node:0
hello, I'm rank 0 out of 2: thread 0
hello, I'm rank 0 out of 2: thread 1
hello, I'm rank 0 out of 2: thread 0
hello, I'm rank 0 out of 2: thread 1
hello, I'm rank 0 out of 2: thread 0
hello, I'm rank 0 out of 2: thread 1
hello, I'm rank 1 out of 2: thread 0
hello, I'm rank 1 out of 2: thread 1
hello, I'm rank 0 out of 2: thread 0
hello, I'm rank 0 out of 2: thread 1
hello, I'm rank 1 out of 2: thread 0
hello, I'm rank 1 out of 2: thread 1
hello, I'm rank 0 out of 2: thread 0
hello, I'm rank 0 out of 2: thread 1
hello, I'm rank 1 out of 2: thread 0
hello, I'm rank 1 out of 2: thread 1
hello, I'm rank 0 out of 2: thread 0
hello, I'm rank 0 out of 2: thread 1
hello, I'm rank 1 out of 2: thread 0
hello, I'm rank 1 out of 2: thread 1
hello, I'm rank 0 out of 2: thread 0
hello, I'm rank 0 out of 2: thread 1
hello, I'm rank 1 out of 2: thread 0
hello, I'm rank 1 out of 2: thread 1
hello, I'm rank 0 out of 2: thread 0
hello, I'm rank 0 out of 2: thread 1
hello, I'm rank 1 out of 2: thread 0
hello, I'm rank 1 out of 2: thread 1
hello, I'm rank 0 out of 2: thread 0
```

```
hello, I'm rank 0 out of 2: thread 1
hello, I'm rank 1 out of 2: thread 0
hello, I'm rank 1 out of 2: thread 1
hello, I'm rank 0 out of 2: thread 0
hello, I'm rank 0 out of 2: thread 1
hello, I'm rank 1 out of 2: thread 0
hello, I'm rank 1 out of 2: thread 1
hello, I'm rank 1 out of 2: thread 0
hello, I'm rank 1 out of 2: thread 1
hello, I'm rank 1 out of 2: thread 0
hello, I'm rank 1 out of 2: thread 1
```

XMPI resource file

C

This appendix displays the contents of the XMPI X resource file stored in /opt/mpi/lib/X11/app-defaults/XMPI.

You should make your own copy of the resource file (you can copy it to the .Xdefaults file in your home directory) and tailor it accordingly.

To save your changes and rebuild the X resource database from scratch, enter:

% **xrdb** *filename*

To save your changes and merge them into the existing X resource database, enter:

% **xrdb -merge** *filename*

In both cases, *filename* represents the name of your tailored resource file.

```
XMPI*Title:XMPI

XMPI*IconName:XMPI

XMPI*multiClickTime:500

XMPI*background:lightgray

XMPI*fontList:
-*-helvetica-bold-r-normal--*-120-*-*-*-*-*-*

XMPI*rankFont:
-*-helvetica-bold-r-normal--*-120-*-*-*-*-*-*

XMPI*msgFont:
-*-helvetica-medium-r-normal--*-120-*-*-*-*-*-*

XMPI*fo_func.fontList:
-*-helvetica-bold-o-normal--*-120-*-*-*-*-*-*

XMPI*dt_dtype.fontList:
-*-helvetica-medium-r-normal--*-100-*-*-*-*-*-*
```

```
XMPI*ctl_bar.topShadowColor:lightslateblue
XMPI*ctl_bar.bottomShadowColor:darkslateblue
XMPI*ctl_bar.background:slateblue
XMPI*ctl_bar.foreground:white
XMPI*banner.background:slateblue
XMPI*banner.foreground:white
XMPI*view_draw.background:black
XMPI*view_draw.foreground:gray
XMPI*trace_draw.background:gray
XMPI*trace_draw.foreground:black
XMPI*kiviat_draw.background:gray
XMPI*kiviat_draw.foreground:black
XMPI*app_list.visibleItemCount:8
XMPI*aschema_list.visibleItemCount:20
XMPI*aschema_text.columns:24
XMPI*prog_mgr*columns:16
XMPI*comCol:cyan
XMPI*rcomCol:plum
XMPI*label_frame.XmLabel.background:#D3B5B5
XMPI*XmToggleButtonGadget.selectColor:red
XMPI*XmToggleButton.selectColor:red
```

Glossary

This glossary lists terms and definitions often used when working with MPI applications.

A

asynchronous
Communication in which sending and receiving processes place no constraints on each other in terms of completion. The communication operation between the two processes may also overlap with computation.

B

bandwidth
Reciprocal of the time needed to transfer a byte. Bandwidth is normally expressed in megabytes per second.

barrier
Collective operation used to synchronize the execution of processes. `MPI_Barrier` blocks the calling process until all receiving processes have called it. This is a useful approach for separating two stages of a computation so messages from each stage are not overlapped.

blocking receive
Communication in which the receiving process does not return until its data buffer contains the data transferred by the sending process.

blocking send
Communication in which the sending process does not return until its associated data buffer is available for reuse. The data transferred can be copied directly into the matching receive buffer or a temporary system buffer.

broadcast
One-to-many collective operation where the root process sends a message to all other processes in the communicator including itself.

buffered send mode
Form of blocking send where the sending process returns when the message is buffered in application-supplied space or when the message is received.

buffering
Amount or act of copying that a system uses to avoid deadlocks. A large amount of buffering can adversely affect performance and make MPI applications less portable and predictable.

C

cluster
Group of computers linked together with an interconnect and software that functions collectively as a parallel machine.

collective communication
Communication that involves sending or receiving messages among a group of processes at the same time. The communication can be one-to-many, many-to-one, or many-to-many. The main collective routines are `MPI_Bcast`, `MPI_Gather`, and `MPI_Scatter`.

communicator
Global object that groups application processes together. Processes in a communicator can communicate with each other or with processes in another group. Conceptually, communicators define a communication context and a static group of processes within that context.

context
Internal abstraction used to define a safe communication space for processes. Within a communicator, context separates point-to-point and collective communications.

D

data-parallel model
Design model where data is partitioned and distributed to each process in an application. Operations are performed on each set of data in parallel and intermediate results are exchanged between processes until a problem is solved.

derived datatypes
User-defined structures that specify a sequence of basic datatypes and integer displacements for noncontiguous data. You create derived datatypes through the use of type-constructor functions that describe the layout of sets of primitive types in memory. Derived types may contain arrays as well as combinations of other primitive datatypes.

domain decomposition
Breaking down an MPI application's computational space into regular data structures such that all computation on these structures is identical and performed in parallel.

E

explicit parallelism
Programming style that requires you to specify parallel constructs directly. Using the MPI library is an example of explicit parallelism.

F

functional decomposition
Breaking down an MPI application's computational space into separate tasks such that all computation on these tasks is performed in parallel.

G

gather
Many-to-one collective operation where each process (including the root) sends the contents of its send buffer to the root.

granularity

Measure of the work done between synchronization points. Fine-grained applications focus on execution at the instruction level of a program. Such applications are load balanced but suffer from a low computation/communication ratio. Coarse-grained applications focus on execution at the program level where multiple programs may be executed in parallel.

group
Set of tasks that can be used to organize MPI applications. Multiple groups are useful for solving problems in linear algebra and domain decomposition.

H

hypernode
Building block of an Exemplar-scalable system. Each hypernode consists of a number of CPUs, I/O, and memory connected by a crossbar and joined to other hypernodes by a Coherent Toroidal Interconnect link.

I

implicit parallelism
Programming style where parallelism is achieved by software layering (that is, parallel constructs are generated through the software). High performance Fortran is an example of implicit parallelism.

intercommunicators
Communicators that allow only processes within the same group or in two different groups to exchange data. These communicators support only point-to-point communication.

intracommunicators
Communicators that allow processes within the same group to exchange data. These communicators support both point-to-point and collective communication.

L

latency
Time between the initiation of the data transfer in the sending process and the arrival of the first byte in the receiving process.

load balancing
Measure of how evenly the work load is distributed among an application's processes. When an application is perfectly balanced, all processes share the total work load and complete at the same time.

locality
Degree to which computations performed by a processor depend only upon local data. Locality is measured in several ways including the ratio of local to nonlocal data accesses.

M

message-passing model
Model in which processes communicate with each other by sending and receiving messages. Applications based on message passing are nondeterministic by default. However, when one process sends two or more messages to another, the transfer is deterministic in the sense that the messages are always received in the order sent.

MIMD
Multiple instruction multiple data. Category of applications in which many instruction streams are applied concurrently to multiple data sets.

MPI
Message passing interface. Set of library routines used to design scalable parallel applications. These routines provide a wide range of operations that include computation, communication, and synchronization. MPI 1.1 is the current standard supported by major vendors.

MPI-2
New standard that adds additional functionality to MPI 1.1.

MPMD
Multiple data multiple program. Implementations of HP MPI that use two or more separate executables to construct an application. This design style can be used to simplify the application source and reduce the size of spawned processes. Each process may run a different executable.

multilevel parallelism
Refers to multithreaded processes that call MPI routines to perform computations. This approach is beneficial for problems that can be decomposed into logical parts for parallel execution (for example, a looping construct that spawns multiple threads to perform a computation and then joins after the computation is complete).

N

nonblocking receive
Communication in which the receiving process returns before a message is stored in the receive buffer. Nonblocking receives are useful when communication and computation can be effectively overlapped in an MPI application. Use of nonblocking receives may also avoid system buffering and memory-to-memory copying.

nonblocking send
Communication in which the sending process returns before a message is stored in the send buffer. Nonblocking sends are useful when communication and computation can be effectively overlapped in an MPI application.

NUMA
Nonuniform memory access architecture. Amount of time for processes to access memory across hypernodes is nonuniform depending upon where data is stored.

P

parallel efficiency
Speed up in the execution of a parallel application.

point-to-point communication
Communication where data transfer involves sending and receiving messages between two processes. This is the simplest form of data transfer in a message-passing model.

polling
Mechanism to handle asynchronous events by actively checking to determine if an event has occurred.

process
Address space together with a program counter, a set of registers, and a stack. Processes can be single threaded or multithreaded. Single-threaded processes can only perform one task at a time. Multithreaded processes can perform multiple tasks concurrently as when overlapping computation and communication.

R

race condition
Situation in which multiple processes vie for the same resource and receive it in an unpredictable manner. Race conditions can lead to cases where applications do not run correctly from one invocation to the next.

rank
Integer between zero and (number of processes - 1) that defines the order of a process in a communicator. Determining the rank of a process is important when solving problems where a master process partitions and distributes work to slave processes. The slaves perform some computation and return the result to the master as the solution.

ready send mode
Form of blocking send where the sending process cannot start until a matching receive is posted. The sending process returns immediately.

reduction
Binary operations (such as summation, multiplication, and boolean) applied globally to all processes in a communicator. These operations are only valid on numeric data and are always associative but may or may not be commutative.

S

scalable
Ability to deliver an increase in application performance proportional to an increase in hardware resources (normally, adding more CPUs).

scatter
One-to-many operation where the root's send buffer is partitioned into *n* segments and distributed to all processes such the *i*th process receives the *i*th segment. *n* represents the total number of processes in the communicator.

send modes

Point-to-point communication in which messages are passed using one of four different types of blocking sends. The four send modes include standard mode (`MPI_Send`), buffered mode (`MPI_Bsend`), synchronous mode (`MPI_Ssend`), and ready mode (`MPI_Rsend`). The modes are all invoked in a similar manner and all pass the same arguments.

shared memory model

Model in which each process can access a shared address space. Concurrent accesses to shared memory are controlled by synchronization primitives.

SIMD

Single instruction multiple data. Category of applications in which homogeneous processes execute the same instructions on their own data.

SPMD

Single program multiple data. Implementations of HP MPI where an application is completely contained in a single executable. SPMD applications begin with the invocation of a single process called the master. The master then spawns some number of identical child processes. The master and the children all run the same executable.

standard send mode

Form of blocking send where the sending process returns when the system can buffer the message or when the message is received.

stride

Constant amount of memory space between data elements where the elements are stored noncontiguously. Strided data are sent and received using derived data types.

subcomplex

Group of CPUs and their associated memory that may span multiple hypernodes on the same host. Hosts are partitioned into subcomplex configurations to achieve the best mix of hardware and software resources.

synchronization

Bringing multiple processes to the same point in their execution before any can continue. For example, `MPI_Barrier` is a collective routine that blocks the calling process until all receiving processes have called it. This is a useful approach for separating two stages of a computation so messages from each stage are not overlapped.

synchronous send mode

Form of blocking send where the sending process returns only if a matching receive is posted and the receiving process has started to receive the message.

T

tag
Integer label assigned to a message when it is sent. Message tags are one of the synchronization variables used to ensure that a message is delivered to the correct receiving process.

task
Uniquely addressable thread of execution.

thread
Smallest notion of execution in a process. All MPI processes have one or more threads. Multithreaded processes have one address space but each process thread contains its own counter, registers, and stack. This allows rapid context switching because threads require little or no memory management.

topologies
Process configurations that determine which processes to run on specific hypernodes in a given subcomplex. You can use the `MPI_TOPOLOGY` environment variable in HP MPI or one of the MPI library routines (for example, `MPI_GRAPH_CREATE` or `MPI_CART_CREATE`) to define an application topology.

trace
Information collected during program execution that you can use to analyze your application. You can collect trace information and store it in a file for later use or analyze it directly when running your application interactively (for example, when you run an application in the XMPI utility).

Index

Symbols

A

B

C

D

E

F

G

H

K

L

M

N

O

P

R

S

T

U

X

B6011-90001