HP MPI User's Guide

Third Edition

B6011-96008 (M)
B6011-90001 (C)
June 1998

Edition: Third

B6011-90001
Remarks: Released with HP MPI V1.4, June, 1998.

Edition: Second

B6011-90001
Remarks: Released with HP MPI V1.3, October, 1997.

Edition: First

B6011-90001
Remarks: Released with HP MPI V1.1, January, 1997.

Notice

Contents

Figures

Tables

Preface

This guide describes the HP MPI 1.4 implementation of the Message Passing Interface (MPI) standard. The guide helps you use HP MPI to develop and run 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 the MPI 1.2 and 2.0 standards. *MPI: The Complete Reference* (http://www.netlib.org/utk/papers/mpi-book) describes the MPI 1.1 standard. You can access the HTML version of the MPI 2.0 standard at http://www.mpi-forum.org.

An HTML version of this guide is provided with HP MPI. See “Directory structure” on page 34 for more information.

Some sections in this book contain command-line examples used to demonstrate HP MPI concepts. These examples use the /bin/csh syntax for illustration purposes.

HP MPI features

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

- Compliance with the MPI 1.2 standard. Support for a subset of the MPI 2.0 standard including:
  - Language interoperability—Allows you to write mixed-language applications or applications that call library routines written in another language.
  - MPI I/O—Supports I/O operations for large files (>2 Gbytes).
  - One-sided communication—Separates the transfer of data from process synchronization. This is best for applications with dynamically changing data access patterns where data distribution is fixed or slowly changing.
- Thread compliance—Allows concurrently running threads in a process to make MPI calls as if the calls are executed in some kind of order.
- 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.
- Profiling—Supports process tracing and monitoring using XMPI and counter instrumentation for collecting cumulative application statistics. A new utility, mpiview, allows you to view counter instrumentation data in graphical format.
- Multiprotocol support—Supports different 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.
- Data mover—Accelerates messaging on scalable servers running under SPP-UX.
- Compliance with the UNIX 95 standard.

- Derived datatypes—Supports optimized collection and communication of derived datatypes.
- Integration with Platform Computing's Load-Sharing Facility product.
- Daemon—Improves application scalability by optionally using daemons for all off-host communication.
- Gang scheduling—Schedules processes as a gang if gang scheduling is enabled.
- Support for 32- and 64-bit libraries.
- Optimization of MPI topology creation functions.
- Diagnostics library—Provides advanced run-time error checking and analysis.

System platforms

HP MPI runs under HP-UX 10.20 and 11.0. It also runs under SPP-UX 5.*x*.

The HP-UX operating system is used on:

- Workstations: s700 series B-, C-, and J-Class
- Midrange servers: s800 series D- and K-Class
- High-end servers: V-Class.

The SPP-UX operating system is used on:

- SPP1600 servers (single- and multi-hypernode)
- S-Class servers
- X-Class servers

Notational conventions

This section describes notational conventions used in this book.

`bold monospace`	In command examples, **`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*]
Brackets ([ ])	In command examples, square brackets designate optional entries.

NOTE A note highlights important supplemental information.

Associated Documents

Associated documents include:

- *MPI: The Complete Reference*, published by MIT Press
- *Parallel Programming Guide for HP-UX Systems*
- *CXperf User's Guide*
- *CXperf Command Reference*

The table below shows World Wide Web sites that contain additional MPI information.

Access	To learn more about
http://www.hp.com/go/mpi	Hewlett-Packard's HP MPI web page
http://www.mpi-forum.org	Official site of the MPI forum
http://www.mcs.anl.gov/Projects/mpi/index.html	Argonne National Laboratory's MPICH implementation of MPI
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
http://www.tc.cornell.edu/Edu/Tutor/MPI/	Cornell Theory Center's MPI tutorial and lab exercises
http://www.mcs.anl.gov/home/thakur/romio	Argonne National Laboratory's implementation of MPI I/O

Credits

HP MPI is based upon MPICH from Argonne National Laboratory and Mississippi State University and LAM from Ohio Supercomputing Center.

The XMPI utility is based upon LAM's version, available at http://www.osc.edu/lam.html.

HP MPI includes ROMIO, a portable implementation of MPI I/O developed at the Argonne National Laboratory.

1 Introduction

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 underlying hardware. 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) 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 local process accesses the memory of a remote process without aid from the remote process. The local process accesses this memory explicitly, not the way it accesses its local memory.
- Threads—In a multithreaded process, the values of application variables are shared by all the threads.

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, the sending and receiving processes must work to transfer the data from the local memory of one to the local memory of the other.

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

- Portability—Message passing 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 on shared and distributed multiprocessors.
- Simplicity—Model supports explicit control of memory references for easier debugging.

However, creating message-passing applications may require more effort than letting a parallelizing compiler produce parallel applications.

MPI concepts

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

Although several message-passing libraries exist on different systems, MPI is popular 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 routines provide a set of functions that support point-to-point communications, collective operations, process groups and communication contexts, process topologies, and datatype manipulation.

Although the MPI library contains a large number of routines to choose from, you can design a number of applications by using the six routines listed in Table 1.

Table 1 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

NOTE You must call `MPI_Finalize` in your application to conform to the MPI Standard. HP MPI issues a warning when a process exits without calling `MPI_Finalize`.

There should be no code before `MPI_Init` and after `MPI_Finalize`. Applications that violate this rule are nonportable and may give incorrect results.

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. Doing this is much more efficient in terms of performance than using `MPI_Send` and `MPI_Recv` 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*)

where

latency	Specifies the time between the initiation of the data transfer in the sending process and the arrival of the first byte in the receiving process.
message_size	Specifies the size of the message in Mbytes.
bandwidth	Denotes the reciprocal of the time needed to transfer a byte. Bandwidth is normally expressed in Mbytes per second.

Obviously, low latencies and high bandwidths lead to better performance.

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.

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 the number of processes 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 *rank* is an integer between zero and (*size* - 1).

Refer to example “communicator.c” on page 134 for more information about using communicators.

Sending and receiving messages

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

In blocking communications, the sending process does not return until the send buffer is available for reuse.

In nonblocking communications, the sending process returns immediately, but the send buffer is not safe for reuse. In this case:

1. The sending routine begins the message transfer and returns immediately.
2. The application does 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 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 and the buffer is ready for reuse.
- Buffered mode (`MPI_Bsend`)—The sending process returns when the message is buffered in the application-supplied buffer.

  Avoid using the `MPI_Bsend` mode. This mode forces an additional copy operation.
- 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 invoked in a similar manner and pass the same arguments. The only difference is in the routine name used to send the message (that is, `MPI_Send` versus `MPI_Ssend`).

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 datatype 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	Designates the 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 datatype 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	Designates the 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	Returns information about the received message. Status information is useful when wildcards are used or the received message is smaller than expected. Status may also contain error codes.

Refer to examples “send_receive.f” on page 121, “ping_pong.c” on page 123, and “master_worker.f90” on page 128 for more information about using blocking communications.

Nonblocking communication

MPI provides nonblocking versions of the blocking send and receive calls. Table 2 lists these calls.

Table 2 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

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_Com` *comm*`, MPI_Request *`*req*`);`

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

To complete nonblocking sends and receives, you can use `MPI_Wait` or `MPI_Test`. The completion of a send indicates that the sending process is free to access the send buffer. The completion of a receive indicates that the receive buffer contains the message, the receiving process is free to access it, and the status object is set.

Collective operations

Applications may require coordinated operations among multiple processes. For example, all processes need to cooperate to 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

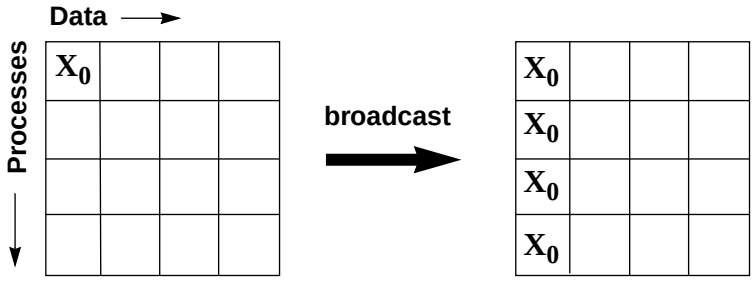
Collective communication involves the exchange of data among all processes in a group. The communication can be one-to-many, many-to-one, or many-to-many.

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:

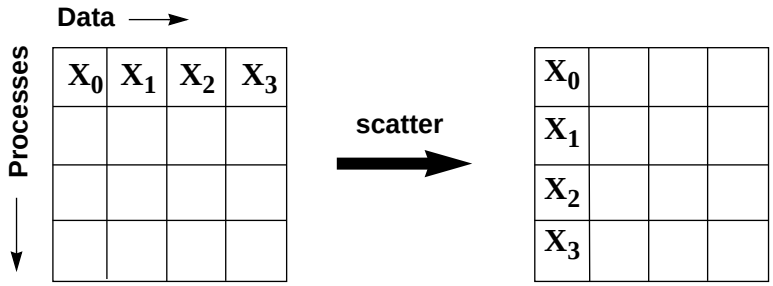
Broadcast — A one-to-many operation where the root sends its data to all other processes in the communicator, including itself. Figure 1 shows the broadcast operation for a data locations in one process.

Figure 1 MPI broadcast operation



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 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 datatype of the buffer entries.
root	Specifies the rank of the root.
comm	Designates the 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 datatype 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 datatype of the receive buffer elements.
root	Denotes the rank of the sending process.
comm	Designates the 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 datatype of the send and receive buffers.
op	Specifies the reduction operation.
root	Indicates the rank of the root process.
comm	Designates the 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 126 and “cart.C” on page 130 for more information about using collective operation.

MPI datatypes

You can use predefined datatypes (for example, MPI_INT in C) to transfer 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 or Fortran application).

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

A derived datatype specifies a sequence of basic datatypes and integer displacements describing the data layout in memory. Derived datatypes are more efficient than using `MPI_Pack` and `MPI_Unpack`, but they cannot handle the case where the data layout varies and is unknown by the receiver beforehand (for example, messages that embed their own layout description).

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

The types of constructor functions include:

- Contiguous—Allows replication of a datatype into contiguous locations.
- Vector—Allows replication of a datatype into locations that consist of equally spaced blocks.
- Indexed—Allows replication of a datatype 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 datatype into a sequence of blocks such that each block consists of replications of different datatypes, copies, and displacements.

HP MPI optimizes collection and communication of derived datatypes.

To create a vector datatype, 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 datatype.
newtype	Specifies the new datatype.

You must now commit the derived datatype 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, 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 datatype 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	Designates the communication context that identifies a group of processes.

Multilevel parallelism

By default, processes in an MPI application can only do 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, a set of registers, and a stack.

A multithreaded process has one address space, but each process thread contains its own counter, registers, and stack.

Multilevel parallelism refers to MPI processes that have multiple threads. Processes become multithreaded through calls to multithreaded libraries, parallel directives and pragmas, and auto-compiler parallelism.

Multilevel parallelism is beneficial for problems you can decompose into logical parts for parallel execution (for example, a looping construct that spawns multiple threads to do a computation and joins after the computation is complete).

See “multi_par.f” on page 135 for an example of multilevel parallelism.

Advanced topics

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

- Error handling
- Process topologies
- User-defined datatypes
- Process grouping
- Attribute caching

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

2 Getting started

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
```

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. For example:

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

Building and running your first application

To quickly gain experience with HP MPI, start 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 -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 HP-UX or SPP-UX or hello_world.c is built on HP-UX so the same binary can run on both hosts):

Step 1. Edit the .rhosts file on jawbone and wizard. Add an entry for wizard in the .rhosts file on jawbone and an entry for jawbone in the .rhosts file on wizard.

Step 2. Change to a writable directory.

Step 3. Enter

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

This step builds the hello_world executable.

Step 4. Copy the hello_world executable from jawbone to your home directory on wizard.

Step 5. 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 the host name or IP address where the specified processes must be run.

Step 6. 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
```

3 Understanding HP MPI

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

- MPI 2.0 Features
- Directory structure
- Compiling applications
- Running applications

MPI 2.0 Features

HP MPI supports the following features from the MPI 2.0 standard:

- MPI I/O (chapter 9)
- Language interoperability (section 4.12)
- Thread safety (section 8.7)
- One-sided communication (chapter 6)
- Miscellaneous features (sections 4.6 through 4.10 and section 8.3)

Each of these features is briefly described below. For more information, refer to the HTML version of the MPI 2.0 standard at http://www.mpi-forum.org.

MPI I/O

Unix I/O functions provide a model of a portable file system. However, the portability and optimization needed for parallel I/O cannot be achieved using this model.

The MPI 2.0 standard defines an interface for parallel I/O that supports partitioning of file data among processes. The standard also supports a collective interface for transferring global data structures between process memories and files.

HP MPI I/O supports a subset of the MPI 2.0 standard using ROMIO, a portable implementation developed at Argonne National Laboratory. This subset includes:

- File manipulation (section 9.2)
- File views (section 9.3)
- Data access (section 9.4 except sections 9.4.4 and 9.4.5)
- Consistency and semantics (section 9.6)

HP MPI I/O has the following limitations:

- All nonblocking I/O requests use a MPIO_Request object instead of MPI_Request. The `MPIO_Test` and `MPIO_Wait` routines are provided to test and wait on MPIO_Request objects. `MPIO_Test` and `MPIO_Wait` have the same semantics as `MPI_Test` and `MPI_Wait` respectively.
- The status argument is not returned in any MPI I/O operation.
- All calls that involve MPI I/O file offsets must use an 8-byte integer. Because HP-UX Fortran 77 only supports 4-byte integers, all Fortran 77 source files that involve file offsets must be compiled using HP-UX Fortran 90. In this case, the Fortran 90 offset is defined by KIND = integer(MPI_OFFSET_KIND).
- `MPI_File_set_atomicity(MPI_File` *fh*`, int` *flag*`)` returns MPI_ERR_UNKNOWN if *fh* is a file that resides on systems other than NFS and VxFS.
- Some I/O routines (for example, `MPI_File_open`, `MPI_File_delete`, and `MPI_File_set_info`) take an input argument called *info*. Supported keys for this argument include:
  - cb_buffer_size—Buffer size for collective I/O.
  - cb_nodes—Number of processes that actually perform I/O in collective I/O.
  - ind_rd_buffer_size—Buffer size for data sieving in independent reads.
  - ind_wr_buffer_size—Buffer size for data sieving in independent writes.

  If a given key is not supported or if the value is invalid, *info* is ignored.

Refer to example “io.c” on page 145 for more information about MPI I/O.

Language interoperability

Language interoperability allows you to write mixed-language applications or applications that call library routines written in another language. For example, you can write applications in Fortran or C that call MPI library routines written in C or Fortran respectively.

MPI provides a special set of conversion routines for converting objects between languages. The types of objects that you can convert include MPI communicators, data types, groups, requests, reduction operations, and status. The routines are shown in Table 3.

Table 3 Language interoperability conversion routines

Routine	Description
`MPI_Fint MPI_Comm_c2f(MPI_Comm);`	Converts a C communicator handle into a Fortran handle.
`MPI_Comm MPI_Comm_f2c(MPI_Fint);`	Converts a Fortran communicator handle into a C handle.
`MPI_Fint MPI_Type_c2f(MPI_Datatype);`	Converts a C data type into a Fortran data type.
`MPI_Datatype MPI_Type_f2c(MPI_Fint);`	Converts a Fortran data type into a C data type.
`MPI_Fint MPI_Group_c2f(MPI_Group);`	Converts a C group into a Fortran group.
`MPI_Group MPI_Group_f2c(MPI_Fint);`	Converts a Fortran group into a C group.
`MPI_Fint MPI_Op_c2f(MPI_Op);`	Converts a C reduction operation into a Fortran reduction operation.
`MPI_Op MPI_Op_f2c(MPI_Fint);`	Converts a Fortran reduction operation into a C reduction operation.
`MPI_Fint MPI_Request_c2f(MPI_Request);`	Converts a C request into a Fortran request.

Routine	Description
`MPI_Request MPI_Request_f2c(MPI_Fint);`	Converts a Fortran request into a C request.
`int MPI_Status_c2f(MPI_Status *, MPI_Fint *);`	Converts a C status into a Fortran status.
`int MPI_Status_f2c(MPI_Fint *, MPI_Status *);`	Converts a Fortran status into a C status.

Thread-compliant library

HP MPI provides a thread-compliant library (libmtmpi) for applications running under HP-UX 11.0 (32-and 64-bits) and applications running under SPP-UX 5.3 that use the pthread library.

By default, the nonthread-compliant library (libmpi) is used when running MPI jobs. Table 4 shows which library to use for a given HP MPI application type.

Table 4 HP MPI library usage

Application type	Libmtmpi	Libmpi	Comments
Non-threaded MPI application		Yes	Most MPI applications.
Non-threaded MPI application with mostly nonblocking communication	Yes	Yes	Potential performance improvement if run with libmtmpi.
-lveclib, MLIB is not parallel		Yes	Should not link with CPSlib. MLIB calls cannot go thread parallel.
-lveclib, MLIB is thread parallel	Yes		Must link with CPSlib. MLIB calls may go thread parallel.
+O3 +Oparallel	Yes		Must use libmtmpi.
Using pthreads or CPSlib calls	Yes		Must use libmtmpi.

To link with libmtmpi, use the `-lmtmpi` option when compiling your application. If you use the `-lmtmpi` option when compiling your application under SPP-UX, you must first run `mpa` on your executable. For example:

```
% mpa -parallel -n -m program
```

To create a communication thread for each process in your job (for example, to overlap computation and communication), specify the `ct` option in the `MPI_MT_FLAGS` environment variable. See "MPI_MT_FLAGS" on page 45 for more information.

To set the level of thread support for your job, you can specify the appropriate run-time option in `MPI_MT_FLAGS` or modify your application to use `MPI_Init_thread` instead of `MPI_Init`.

To modify your application, replace the call to `MPI_Init` with

`MPI_Init_thread(int *argc, char *((*argv) []), int` *required*`, int *`*provided*`));`

where

required Specifies the desired level of thread support.

provided Specifies the provided level of thread support.

Table 5 shows the possible thread-initialization values for *required* and the values returned by *provided* for libmpi and libmtmpi.

Table 5 Thread-initialization values

MPI library	**Value for** *required*	**Value returned by** *provided*
libmpi	MPI_THREAD_SINGLE	MPI_THREAD_SINGLE
libmpi	MPI_THREAD_FUNNELED	MPI_THREAD_SINGLE
libmpi	MPI_THREAD_SERIALIZED	MPI_THREAD_SINGLE
libmpi	MPI_THREAD_MULTIPLE	MPI_THREAD_SINGLE
libmtmpi	MPI_THREAD_SINGLE	MPI_THREAD_SINGLE
libmtmpi	MPI_THREAD_FUNNELED	MPI_THREAD_FUNNELED
libmtmpi	MPI_THREAD_SERIALIZED	MPI_THREAD_SERIALIZED
libmtmpi	MPI_THREAD_MULTIPLE	MPI_THREAD_MULTIPLE

Table 6 shows the relationship between the possible thread-support levels in `MPI_Init_thread` and the corresponding options in `MPI_MT_FLAGS`

Table 6 Thread-support levels

`MPI_Init_thread`	`MPI_MT_FLAGS`	Behavior
MPI_THREAD_SINGLE	single	Only one thread will execute.
MPI_THREAD_FUNNELED	fun	The process may be multithreaded, but only the main thread will make MPI calls.
MPI_THREAD_SERIALIZED	serial	The process may be multithreaded, and multiple threads can make MPI calls, but only one call can be made at a time.
MPI_THREAD_MULTIPLE	mult	Multiple threads may call MPI at any time with no restrictions. This option is the default.

Refer to example “thread_safe.c” on page 147 for more information about thread compliance.

To prevent application deadlock, do not call libmtmpi from a signal handler or cancel a thread that is executing inside an MPI routine.

Libmtmpi does not support counter instrumentation (“Using counter instrumentation” on page 66) and trace file analysis (“Using XMPI” on page 71).

Libmtmpi supports calls to `MPI_Init_thread`, `MPI_Is_thread_main`, and `MPI_Query_thread` in the MPI 2.0 standard. No other calls are supported.

One-sided communication

Message-passing communication involves transferring data from the sending process to the receiving process. It also requires synchronization between the sender and receiver.

One-sided communication separates the transfer of data from process synchronization. This mode of communication is best for applications with dynamically changing data access patterns where data distribution is fixed or slowly changing. Processes in such applications, however, may not know which data in their memory needs to be accessible by remote processes or even the identity of these remote processes.

In one-sided communication, the transfer parameters are all available on one side, and a single process specifies both the source and data buffers. In this case, applications can open windows in their memory space that are accessible by remote processes.

HP MPI supports a subset of the MPI 2.0 one-sided communication functionality. Data transfer is handled using `MPI_Put`and `MPI_Get`, which places data in and retrieves data from a remote window segment respectively. Synchronization is handled using `MPI_Win_fence` and `MPI_Win_lock/MPI_Win_unlock`, which synchronizes all data transfers on a window and locks/unlocks a single window respectively.

Restrictions for the HP MPI implementation of one-sided communication include:

- Only single-host operations are supported.
- MPI window segments must be allocated using `MPI_Alloc_mem`; they cannot be placed in COMMON blocks, the stack, or the heap.

Miscellaneous features

Miscellaneous features that are supported include:

- Committing a committed datatype—Allows `MPI_Type_commit` to accept committed datatypes. In this case, the action is the same as a no-op.
- Allowing user functions at process termination—Defines what actions take place when a process terminates. These actions are specified by attaching an attribute to `MPI_Comm_self` with a callback function.

- Determining whether MPI has finished—Allows layered libraries to determine whether MPI is still active by using `MPI_Finalized`.
- Using the info object—Sets (key, value) pairs for an information object as a way to provide system-dependent hints. Info object routines include:
  - `MPI_Info_create`—Creates a new info object.
  - `MPI_Info_set`—Adds the (key,value) pair to info and overrides the value if a value for the same key was previously set.
  - `MPI_Info_delete`—Deletes a (key,value) pair from info.
  - `MPI_Info_get`—Retrieves the value associated with key in a previous call to `MPI_Info_set`.
  - `MPI_Info_get_valuelen`—Retrieves thew length of the value associated with key.
  - `MPI_Info_get_nkeys`—Returns the number of keys currently defined in info.
  - `MPI_Info_get_nthkey`—Returns the nth defined key in info.
  - `MPI_Info_dup`—Duplicates an existing info object, creating a new object with the same (key,value) pairs and ordering of keys.
  - `MPI_Info_free`—Frees info.
- Associating information with status—Sets the number of elements to associate with the status for requests. In addition, sets the status to associate with the cancel flag to indicate whether a request was cancelled. Status routines include:
  - `MPI_Status_set_elements`—Modifies the opaque part of status.
  - `MPI_Status_set_cancelled`—Indicates whether a status request is cancelled.

Directory structure

All HP MPI files are stored in the /opt/mpi directory. The directory structure is organized as shown in Table 7.

Table 7 Organization of the /opt/mpi directory

Subdirectory	Contents
bin	Command files for the HP MPI utilities
doc/html	HTML version of the *HP MPI User's Guide*
help	Source files for the example programs
include	Header files
lib/X11/app-defaults	Application default settings for the XMPI trace utility and the mpiview profiling tool
lib/pa1.1	MPI 32-bit libraries
lib/pa20_64	MPI 64-bit libraries
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

The man pages located in the /opt/mpi/share/man/man1.Z subdirectory are grouped into three categories: compilation, general, and run time. The compilation and run-time categories correspond to available types of HP MPI utilities. All three categories are described in Table 8.

Table 8 Man page categories

Category	Description
Compilation	Describes the available compilation utilities. Refer to “Compiling applications” on page 36 for more information.
General	Describes the general features of HP MPI. The man page is called MPI.1.
Run time	Describes the available run-time utilities. Refer to “Run-time utility commands” on page 54 for more information.

Compiling applications

The compiler you use 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 9.

Table 9 Compilation utilities

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

If aCC is not available, `mpiCC` uses CC as the default C++ compiler.

Even though the `mpiCC` and `mpif90` compilation utilities are shipped with HP MPI, all C++ and Fortran 90 applications 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 10.

Table 10 Compilation environment variables

Utility	Environment variable
`mpicc`	`MPI_CC`
`mpiCC`	`MPI_CXX`
`mpif77`	`MPI_F77`
`mpif90`	`MPI_F90`

To set a compilation environment variable, enter:

`% setenv` *compilation_environment_variable path*

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

64-bit support

HP-UX 11.0 is available as a 32- and 64-bit operating system. You must run 64-bit executables on the 64-bit system (though you can build 64-bit executables on the 32-bit system).

HP MPI supports a 64-bit version of the MPI library on platforms running HP-UX 11.0. Both 32- and 64-bit versions of the library are shipped with HP-UX 11.0 (only a 32-bit version is shipped with HP-UX 10.20). For HP-UX 11.0, you cannot mix 32-bit and 64-bit executables in the same application.

The `mpicc` and `mpiCC` compilation commands link the 64-bit version of the library if you compile with the `+DA2.0W` or `+DD64` options. The `mpif90` compilation command links the 64-bit version of the library if you compile with the `+DA2.0W` option. Otherwise, the 32-bit version is used.

Running applications

Most HP MPI applications are run using the `mpirun` command. You should invoke the `mpirun` command with the `-j` option, which 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` *#* syntax to start your application. For example, to start an executable named hello_world with four processes, enter:

```
% hello_world -j -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` *#* syntax, although `mpirun` is still recommended.

Types of applications

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

Running SPMD applications

A single program multiple data (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 do.

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

```
% mpirun -np # program
```

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

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

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

Running MPMD applications

A multiple program multiple data (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 -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 -f appfile
```

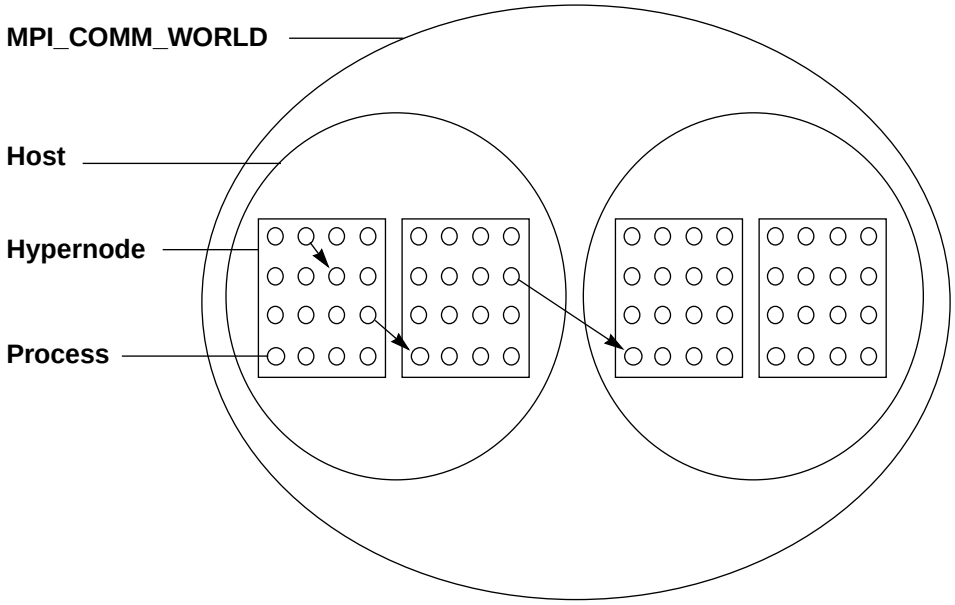
See “Creating an appfile” on page 58 for more information about using appfiles.

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.

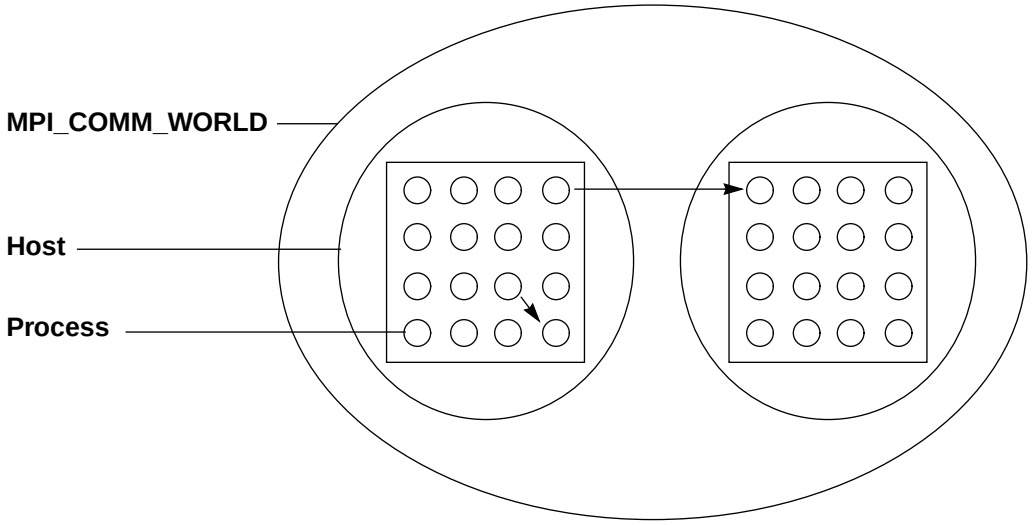
Point-to-point and collective protocols on an X-Class server support messaging between:

- Processes on the same host (on the same or different hypernodes)—Data is transferred using optimized byte-copy and global shared memory.
- Processes on different hosts—Data is transferred using TCP/IP.

The communication speed of protocols for servers running under SPP-UX 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 V2200 server is shown in Figure 4.

Figure 4 Multiprotocol messaging with a V2200 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 servers running under HP-UX support messaging between:

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

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 run time.

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. To propagate environment variables to remote hosts, you must specify each variable in an appfile using the `-e` option. See “Creating an appfile” on page 58 for more information.

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

- `MPI_FLAGS`
- `MPI_DLIB_FLAGS`
- `MPI_MT_FLAGS`
- `MPI_GLOBMEMSIZE`
- `MPI_TOPOLOGY`
- `MPI_SHMEMCNTL`
- `MPI_TMPDIR`
- `MPI_XMPI`
- `MPI_WORKDIR`
- `MPI_CHECKPOINT`
- `MPI_INSTR`
- `MPI_COMMD`
- `MPI_LOCALIP`
- `MP_GANG`

MPI_FLAGS

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

```
[ecxdb,][edde,][exdb,][egdb,][j,][l,][s[a|p][#],][v,]
[y[#],][o,][+E2]
```

where

ecxdb	Starts a separate CXdb session for each process. The debugger must be in the command search path. This option is only provided for backward compatibility on servers running under SPP-UX. See “Debugging HP MPI applications” on page 108 for more information.
edde	Starts the application under the DDE debugger. The debugger must be in the command search path. This option is only supported on servers running under HP-UX. See “Debugging HP MPI applications” on page 108 for more information.
exdb	Starts the application under the xdb debugger. The debugger must be in the command search path. This option is only supported on servers running under HP-UX. See “Debugging HP MPI applications” on page 108 for more information.
egdb	Starts the application under the gdb debugger. The debugger must be in the command search path. This option is only supported on servers running under HP-UX. See “Debugging HP MPI applications” on page 108 for more information.
j	Prints the HP MPI job identifier.
l	Reports memory leaks caused by not freeing memory allocated when an HP MPI job is run. For example, if you create a new communicator or user-defined datatype after you call MPI_Init, you must free the memory allocated to these objects before you call MPI_Finalize. In C, this is analogous to making calls to malloc() and free() for each object created during program execution. Setting the l option may decrease application performance.

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 option is sp604800, which issues a SIGPROF once a week. 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. The SIGPROF option is not supported on servers running under SPP-UX when your application executable is in Extended Standard Object Module format.
v	Prints the version number.
y[#]	Enables spin/yield logic. The spin value # is any integer between zero and 10,000 and is measured in milliseconds. To spin without yielding, specify y without a spin value. The system treats a nonzero spin value as a recommendation only. It does not guarantee that the value you specify will be used.
o	Writes an optimization report to stdout. MPI_Cart_create and MPI_Graph_create optimize the mapping of processes onto the virtual topology if rank reordering is enabled. See “Topology optimization” on page 105 for more information.
+E2	Sets -1 as the value of .TRUE. and 0 as the value for FALSE. when returning logical values from HP MPI routines called within Fortran 77 applications.

MPI_DLIB_FLAGS

MPI_DLIB_FLAGS controls miscellaneous run-time options when using the diagnostics library. The MPI_DLIB_FLAGS syntax is shown below:

[ns,][strict,][nmsg,][dump:*prefix*,][dumpf:*prefix*]

where

ns	Disables message signature analysis.
strict	Enables MPI object-space corruption detection. Setting this option for applications that make calls to routines in the MPI 2.0 standard may produce false error messages.
nmsg	Disables detection of multiple buffer writes during receive operations.
dump:*prefix*	Dumps (unformatted) all sent and received messages to *prefix*.msgs.*rank* where *rank* is the rank of a specific process.
dumpf:*prefix*	Dumps (formatted) all sent and received messages to *prefix*.msgs.*rank* where *rank* is the rank of a specific process.

See “Using the Diagnostics Library” on page 109 for more information.

MPI_MT_FLAGS

MPI_MT_FLAGS controls run-time options when using the thread-compliant version of HP MPI. The MPI_MT_FLAGS syntax is shown below:

[ct,][single,][fun,][serial,][mult]

where

ct	Creates a hidden communication thread for each rank in the job. When enabling this option, be careful not to oversubscribe your system. For example, if you enable ct for a 16-process application running on a 16-way machine, the result will be a 32-way job.
single	Asserts that only one thread will execute.
fun	Asserts that a process may be multithreaded, but only the main thread will make MPI calls (that is, all calls are funneled to the main thread).

`serial`	Asserts that a process may be multithreaded, and multiple threads may make MPI calls, but only one call will be made at a time (that is, all calls are serialized).
`mult`	Asserts that multiple threads may call MPI at any time with no restrictions.

The `single`, `fun`, `serial`, and `mult` options are mutually exclusive. For example, if you specify the `serial` and `mult` options in MPI_MT_FLAGS, only the last option is processed (in this case, the `mult` option).

If no run-time option is specified, the default is `mult`.

For more information about using `MPI_MT_FLAGS` with the thread-compliant library, see “Thread-compliant library” on page 29.

MPI_GLOBMEMSIZE

`MPI_GLOBMEMSIZE` specifies the amount of shared memory allocated for all processes in an HP MPI application. The `MPI_GLOBMEMSIZE` syntax is shown below:

amount

where *amount* specifies the total amount of shared memory in bytes for all processes. The default is 2 Mbytes for up to 64-way applications and 4 Mbytes for larger applications.

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. Otherwise, swapping overhead will degrade application performance.

MPI_TOPOLOGY

MPI_TOPOLOGY controls application process placement within a subcomplex on servers running under SPP-UX (the value is ignored on HP-UX systems). The MPI_TOPOLOGY syntax is shown below:

[[*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 logical 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 maps to logical hypernode 0, and physical hypernode 4 maps 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 58 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 generates an error message and terminates 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, use the `scm` utility to determine the configuration of system subcomplexes before invoking `mpirun`.

The default subcomplex on all systems is called System. Use the `mpa` utility to change 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. The `MPI_SHMEMCNTL` syntax is shown below:

nenv, *frag*, *generic*

where

nenv	Specifies the number of envelopes per process pair. The default is 8.
frag	Denotes the size in bytes of the message-passing fragments region. The default is 87.5 percent of shared memory.
generic	Specifies the size in bytes of the generic-shared memory region. The default is 12.5 percent of shared memory.

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. The `MPI_TMPDIR` syntax is shown below:

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.

The MPI_XMPI syntax is shown below:

prefix[:*bs###*][:nc][:off][:s][:nf][:k]

where

prefix	Specifies the tracing output file prefix. This is a required parameter.
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 tracing for 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.

`nf`	Denotes that a consolidated trace file is not generated. In addition, raw trace files are not deleted. You may want to use this option if your application contains a large number of processes, and you do not want to wait for `MPI_Finalize` to consolidate the raw trace files before your application terminates.
`k`	Specifies that raw trace files are kept.

Even though you can specify tracing options through the `MPI_XMPI` environment variable, the recommended approach is to use the `mpirun` command with the `-t` option instead. In this case, the specifications you provide with the `-t` option take precedence over any specifications you may have set with `MPI_XMPI`. Using `mpirun` to specify tracing options guarantees that multihost applications do tracing in a consistent manner. See “mpirun” on page 55 for more information.

Trace-file generation (in conjunction with XMPI) and counter instrumentation are mutually exclusive profiling techniques. Trace-file generation is not supported for appfiles where all the hosts are remote.

MPI_WORKDIR

By default, HP MPI applications execute in the directory where they are started. `MPI_WORKDIR` changes the execution directory. The `MPI_WORKDIR` syntax is shown below:

directory

where *directory* specifies an existing directory where you want the application to execute.

MPI_CHECKPOINT

You can checkpoint and restart HP MPI applications running under SPP-UX on a single subcomplex by setting `MPI_CHECKPOINT`. In this case, you cannot start your application using `mpirun`. `MPI_CHECKPOINT` does not require specific arguments. For example, to checkpoint and restart the hello_world application:

```
% setenv MPI_CHECKPOINT
% hello_world -np 4
```

When you use MPI_CHECKPOINT, the following limitations apply:

- Your HP MPI job is not assigned a job ID. You cannot monitor or terminate the job using the mpijob or mpiclean utilities respectively. Also, no job ID is printed when the j option is set in MPI_FLAGS. If your application crashes or hangs, you must do manual cleanup using the kill or ipcrm UNIX utilities.
- MPI_Abort does not kill peer processes in the communicator. In this case, only the calling process terminates.
- Direct process-to-process byte-copy is disabled. This results in a bandwidth reduction for large message transfers.

MPI_INSTR

MPI_INSTR enables counter instrumentation for profiling HP MPI applications. The MPI_INSTR syntax is shown below:

prefix[:*b#1*,*#2*][:nd][:nc][:off][:nl][:np][:nm][:c]

where

prefix	Specifies the instrumentation output file prefix. The rank zero process writes the application's measurement data in human-readable format to *prefix*.instr and in internal format to *prefix*.mpiview. If the prefix does not represent an absolute pathname, the instrumentation output file is opened in the working directory of the rank zero process when MPI_Init is called.
b#1,#2	Redefines the instrumentation message bins to include a bin having byte range *#1* and *#2* inclusive. The high bound of the range can be infinity, representing the largest possible message size.
nd	Disables rank-by-peer density information when running counter instrumentation.
nc	Specifies no clobber. If the instrumentation output file exists, MPI_Init aborts.
off	Denotes that counter instrumentation is initially turned off and only begins after all processes collectively call MPIHP_Trace_on.

`nl`	Specifies not to dump a long breakdown of the measurement data to the instrumentation output file (in this case, do not dump minimum, maximum, and average time data).
`np`	Denotes not to dump a per-process breakdown of the measurement data to the instrumentation output file.
`nm`	Specifies not to dump message-size measurement data to the instrumentation output file.
`c`	Specifies not to dump time measurement data to the instrumentation output file.

See “Using counter instrumentation” on page 66 for more information.

Even though you can specify profiling options through the `MPI_INSTR` environment variable, the recommended approach is to use the `mpirun` command with the `-i` option instead. Using `mpirun` to specify profiling options guarantees that multihost applications do profiling in a consistent manner. See “mpirun” on page 55 for more information.

Counter instrumentation and trace-file generation (used in conjunction with XMPI) are mutually exclusive profiling techniques. Counter instrumentation is not supported for appfiles where all the hosts are remote.

MPI_COMMD

`MPI_COMMD` routes all off-host communication through daemons rather than between processes. The `MPI_COMMD` syntax is shown below:

out_frags`,`*in_frags*

where

out_frags	Specifies the number of 16Kbyte fragments available in shared memory for outbound messages. Outbound messages are sent from processes on a given host to processes on other hosts using the communication daemon. The default value for *out_frags* is 64. Increasing the number of fragments for applications with a large number of processes improves system throughput.

in_frags Specifies the number of 16Kbyte fragments available in shared memory for inbound messages. Inbound messages are sent from processes on one or more hosts to processes on a given host using the communication daemon.

The default value for *in_frags* is 64. Increasing the number of fragments for applications with a large number of processes improves system throughput.

See “Communicating using daemons” on page 62 for more information.

MPI_LOCALIP

`MPI_LOCALIP` specifies the host IP address that is assigned throughout a session. Ordinarily, `mpirun` and `xmpi` determine the IP address of the host they are running on by calling `gethostbyaddr`. However, when a host uses a SLIP or PPP protocol, the host's IP address is dynamically assigned only when the network connection is established. In this case, `gethostbyaddr` may not return the correct IP address.

The `MPI_LOCALIP` syntax is shown below:

xxx.xxx.xxx.xxx

where *xxx.xxx.xxx.xxx* specifies the host IP address.

MP_GANG

MP_GANG enables gang scheduling. Gang scheduling can improve application performance in loaded timeshare environments that are over subscribed. It also supports low-latency interactions among processes in shared-memory applications.

The `MP_GANG` syntax is shown below:

`[ON|OFF]`

where

`ON` Enables gang scheduling.

`OFF` Disables gang scheduling.

For multihost configurations, you need to set `MP_GANG` for each appfile entry.

Gang scheduling improves the latency for synchronization by ensuring that all runable processes in a gang are scheduled simultaneously. Processes waiting at a barrier, for example, do not have to wait for processes that are not currently scheduled. This proves most beneficial for applications with frequent synchronization operations. Applications with infrequent synchronization, however, may perform better if gang scheduling is disabled.

Process priorities for gangs are managed identically to timeshare policies. The timeshare priority scheduler determines when to schedule a gang for execution. While it is likely that scheduling a gang will preempt one or more higher priority timeshare processes, the gang-scheduler policy is fair overall. In addition, gangs are scheduled for a single time slice, which is the same for all processes in the system.

HP MPI supports gang scheduling for applications running under HP-UX 11.0 Extension Pack, June 1998 (XR39/IPR9806). MPI processes are allocated statically at the beginning of execution. As an MPI process creates new threads, they are all added to the same gang if `MP_GANG` is enabled.

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`
- `mpiview`

mpirun

`mpirun` starts an HP MPI application.

`mpirun` syntax has four forms:

- `mpirun [-np #] [-help] [-version] [-jpvW] [-t spec] [-i spec] [-h host] [-l user] [-e var[=val] [...]] [-sp paths] [-commd] program [args]`
- `mpirun [-help][-version] [-jpvW] [-t spec] [-i spec] [-commd] -f appfile`
- `bsub [lsf_options] pam -mpi mpirun [-np #][-help] [-version] [-jpvW] [-t spec] [-i spec] [-h host] [-l user] [-e var[=val] [...]] [-sp paths] [-commd] program [args]`
- `bsub [lsf_options] pam -mpi mpirun [-help][-version] [-jpvW] [-t spec] [-i spec] [-commd] -f appfile`

where

`-np #`	Specifies the number of processes to run.
`-help`	Prints usage information for the utility.
`-version`	Prints the version information.
`-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 set up 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 49 for the list of options you can use.

`-i` *spec*	Enables run-time instrumentation profiling for all processes. *spec* specifies options used when profiling. See “MPI_INSTR” on page 51 for the list of options you can use.
`-h` *host*	Starts the processes on *host* (default is *local_host*).
`-l` *user*	Specifies the user name on the target host (default is local username).
`-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.
args	Specifies command-line arguments to the program.
lsf_options	Specifies bsub options that the load-sharing facility (LSF) applies to the entire job (that is, every host). Refer to the bsub(1) man page for a list of options you can use. Note that LSF must be installed for *lsf_options* to work correctly.
`-commd`	Routes all off-host communication through daemons rather than between processes. See “Communicating using daemons” on page 62 for more information.
`-f` *appfile*	Starts the application described in *appfile*.

Use the first syntax 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.

Use the second syntax 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. For example:

```
% mpirun -t my_trace:k -f my_appfile
```

enables tracing, sets the prefix of the tracing output file to my_trace, specifies that the raw trace files are kept, and runs an appfile named my_appfile.

Use the third syntax to invoke LSF for applications where all processes execute the same program on the same host. In this case, LSF assigns a host to the MPI job. For example:

```
% bsub pam -mpi mpirun -np 4 compute_pi
```

requests a host assignment from LSF and runs the compute_pi application with four processes. See “Assigning hosts using LSF” on page 64 for more information.

Use the fourth syntax to invoke LSF for applications that run on multiple hosts. Each host specified in the appfile is treated as a symbolic name, referring to the host that LSF assigns to the MPI job. For example:

```
% bsub pam -mpi mpirun -f my_appfile
```

runs an appfile named my_appfile and requests host assignments for all remote and local hosts specified in my_appfile. If my_appfile contains

```
-h voyager -np 10 send_receive
-h enterprise -np 8 compute_pi
```

host assignments are returned for the two symbolic links voyager and enterprise. See “Assigning hosts using LSF” on page 64 for more information.

When requesting a host from LSF, you must ensure that your executable’s path is accessible by all machines in the resource pool.

Creating an appfile

The format of entries in an appfile is line oriented. Lines that end with the blackslash (\) 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.
`-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).
`-np` *#*	Specifies the number of processes to run (defaults to one).
program	Specifies the name of the executable to run. The executable is searched for in $PATH.
args	Specifies command line arguments to the program.

One way to set environment variables on remote hosts is to use 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	Option that indicates whether the process is running (an x is used) or has been terminated.
PROGNAME	Program names used in the HP MPI application.

An `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
22623    charlie   12       /home/watts
22573    keith     14       /home/richards
22617    mick      100      /home/jagger
22677    ron       4        /home/wood
```

NOTE Invoke `mpijob` on the host on which you initiated `mpirun`.

mpiclean

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

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

where

`-help`	Prints usage information for the utility.
`-v`	Turns on verbose mode.
`-m`	Cleans up your shared-memory segments.
`-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 servers running under SPP-UX. The third syntax is used when an application aborts during `MPI_Init`, and the termination of processes does not destroy the allocated shared-memory segments.

The MPI library checks for 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 only kill jobs that you own.

The second way is only provided on servers running under SPP-UX for backward compatibility. In this approach, you specify `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.

NOTE Invoke `mpiclean` on the host on which you initiated `mpirun`.

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 71.

mpiview

`mpiview` invokes the mpiview utility. The mpiview utility displays counter instrumentation data in graphical form. The `mpiview` syntax is shown below:

`mpiview [`*`prefix`*`.mpiview]`

where *prefix*.`mpiview` specifies the name of the instrumentation profile (in internal format) created by using the `mpirun` command with the `-i` option. For more information, see “Creating an instrumentation profile” on page 66 and the mpiview.1 man page.

Communicating using daemons

By default, off-host communication between processes is implemented using direct socket connections between process pairs. For example, if process A on host1 communicates with processes D and E on host2, then process A sends messages using a separate socket for each process D and E.

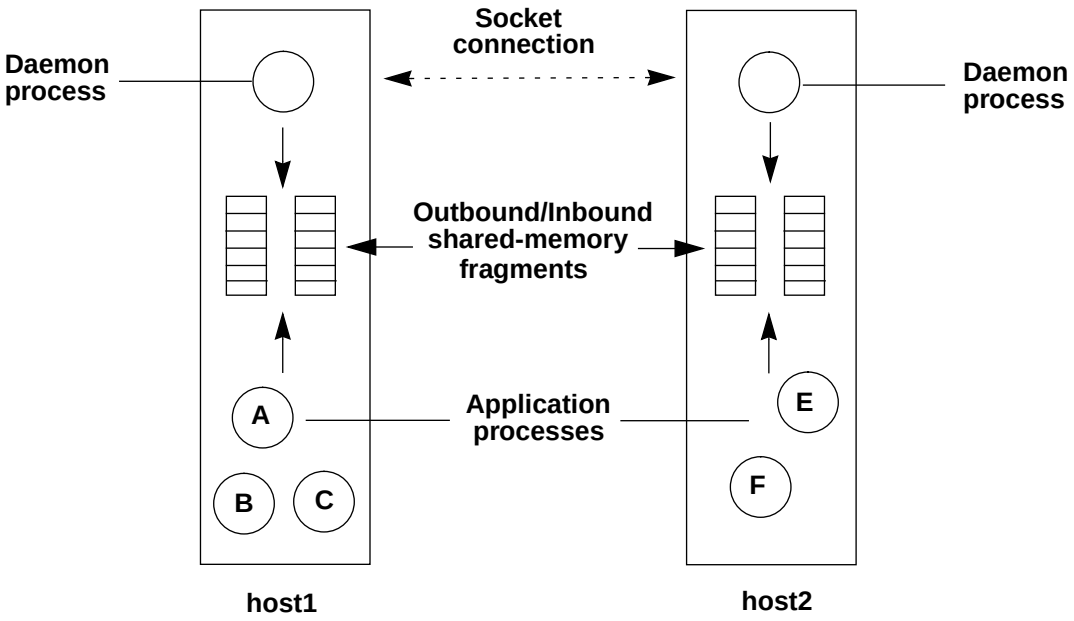
This is referred to as the n-squared or direct approach because to run an *n*-process application, n^2 sockets are required to allow processes on one host to communicate with processes on other hosts. If you use the direct approach, you should be careful that the total number of open sockets does not exceed the system limit.

You can also use an indirect approach and specify that all off-host communication occur between daemons. In this case, the processes on a host use shared memory to send messages to and receive messages from the daemon. The daemon, in turn, uses a socket connection to communicate with daemons on other hosts.

In an appfile (multihost applications), the first process created for each appfile entry becomes the communication daemon for that application. Use the `MPI_TOPOLOGY` environment variable to decide where the first process should run.

Figure 5 shows the structure for daemon communication.

Figure 5 Daemon communication



To use daemon communication, specify the `-commd` option in the `mpirun` command. See “mpirun” on page 55 for more information. Once you have set the `-commd` option, you can use the `MPI_COMMD` environment variable to specify the number of shared-memory fragments used for inbound and outbound messages. See “MPI_COMMD” on page 52 for more information.

Daemon communication may result in lower application performance. Therefore, use it only when scaling an application to a large number of hosts.

Assigning hosts using LSF

The load-sharing facility (LSF) allocates one or more hosts to run an MPI job. In general, LSF improves resource utilization for MPI jobs that run in multihost environments.

By default, `mpirun` maps host names to IP addresses for single-host and multihost jobs. You can use LSF to perform this mapping by specifying a variant of `mpirun` to execute your job. Using LSF improves resource utilization

For example, to execute a single-host job on a local host, enter:

```
% bsub [lsf_options] pam -mpi mpirun [-np #] [-help]
[-version] [-jpvW] [-t spec] [-i spec] [-h host]
[-l user] [-e var[=val] [...]] [-sp paths]
[-commd] program [args]
```

where *lsf_options* specifies bsub options that LSF applies to the local host. Refer to the bsub(1) man page for a list of LSF options you can use.

To execute a multihost job using an appfile, enter:

```
% bsub [lsf_options] pam -mpi mpirun [-help]
[-version] [-jpvW] [-t spec] [-i spec] [-commd] -f
appfile
```

where *lsf_options* specifies bsub options that LSF applies to all hosts in the appfile, and *appfile* contains entries of the form:

```
-h sockeye -np 6 send_receive
-h jawbone -np 4 compute_pi
```

In this case, the hosts sockeye and jawbone are treated as symbolic names that refer to the hosts that LSF eventually allocates to each job respectively.

See “mpirun” on page 55 for more information about the `mpirun` options you can use.

4 Profiling

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

- Using counter instrumentation
- Using XMPI
- Using CXperf
- Using the profiling interface

Using counter instrumentation

Counter instrumentation provides cumulative statistics about your applications. Once you have created an instrumentation profile, you can view the data either in human-readable format or in internal format using the mpiview utility.

Creating an instrumentation profile

To create an instrumentation profile, enter:

```
% mpirun -i spec -np # program
```

where

-i *spec* Enables run-time instrumentation profiling for all processes. *spec* provides options used when profiling. See “MPI_INSTR” on page 51 for more information about options you can use.

You must specify the `-i` option before the program name.

-np # Specifies the number of processes to run.

program Specifies the name of the executable to run.

For example, to create an instrumentation profile for an application called compute_pi.f, enter:

```
% mpirun -i compute_pi -np 2 compute_pi
```

This invocation creates an instrumentation profile in two formats: compute_pi.instr (human-readable) and compute_pi.mpiview (internal).

HP MPI provides the nonstandard `MPIHP_Trace_on` and `MPIHP_Trace_off` routines to collect profile information for selected code sections only (by default, the entire application is profiled from `MPI_Init` to `MPI_Finalize`). You insert the `MPIHP_Trace_on` and `MPIHP_Trace_off` pair around code that you want to profile. Then, you build the application and invoke `mpirun` with the `:off` option.

Viewing the human-readable format

To view an instrumentation profile in human-readable format, print *prefix*.instr (in the case of the compute_pi.f application, this is compute_pi.instr).

The overhead time in the profile represents the time a process or routine spends inside MPI. For example, the time a process spends doing message packing.

The blocking time represents the time a process or routine is blocked waiting for a message to arrive before resuming execution.

The profile also includes density information that shows rank-by-peer data for MPI time and messages.

The contents of compute_pi.instr are shown below.

```
Version: HP MPI B6011/B6280 - HP-UX 10.20

Date:    Mon Feb  2 17:36:59 1998
Scale:   Wall Clock Seconds
Processes: 2
User:  33.65%
MPI:   66.35% [Overhead:66.35% Blocking:0.00%]

Total Message Count: 4

Minimum Message Range:            4 [0..32]
Maximum Message Range:            4 [0..32]
Average Message Range:            4 [0..32]

Top Routines:

MPI_Init        86.39% [Overhead:86.39% Blocking: 0.00%]
MPI_Bcast       12.96% [Overhead:12.96% Blocking: 0.00%]
MPI_Finalize     0.43% [Overhead: 0.43% Blocking: 0.00%]
MPI_Reduce       0.21% [Overhead: 0.21% Blocking: 0.00%]

Communication Hot Spots:

Minimum Pair:   6.58% Rank:0   Rank:0
Maximum Pair:   6.60% Rank:0   Rank:1
Average Pair:   6.59%

Rank:0  Rank:1  6.60% [Overhead: 6.60% Blocking: 0.00%]
Rank:0  Rank:0  6.58% [Overhead: 6.58% Blocking: 0.00%]

------------------------------------------------------------
----------------  Instrumentation Data   ----------------
------------------------------------------------------------

Application Summary by Rank:
```

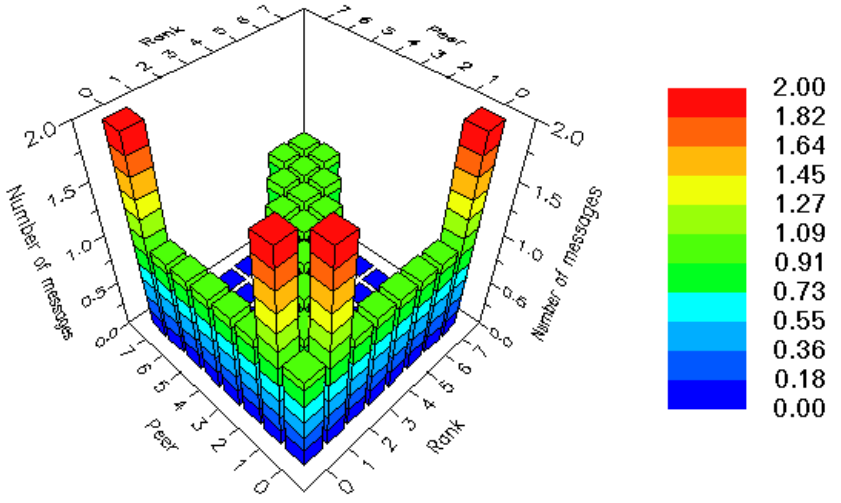
```
Rank    Duration     Overhead     Blocking   User    MPI
-----------------------------------------------------------
1       0.248998     0.221605     0.000000   11.00%  89.00%
0       0.249118     0.108919     0.000000   56.28%  43.72%
-----------------------------------------------------------

Routine Summary:

Routine         Calls   Overhead        Blocking
---------------------------------------------------------
MPI_Init        2       0.285536        0.000000
min                     0.086926        0.000000
max                     0.198610        0.000000
avg                     0.142768        0.000000
MPI_Bcast       2       0.042849        0.000000
min                     0.021393        0.000000
max                     0.021456        0.000000
avg                     0.021424        0.000000
MPI_Finalize    2       0.001434        0.000000
min                     0.000240        0.000000
max                     0.001194        0.000000
avg                     0.000717        0.000000
MPI_Reduce      2       0.000705        0.000000
min                     0.000297        0.000000
max                     0.000408        0.000000
avg                     0.000353        0.000000
---------------------------------------------------------

Routine Summary by Rank:

Routine         Rank    Calls   Overhead        Blocking
---------------------------------------------------------
MPI_Init        0       1       0.086926        0.000000
                1       1       0.198610        0.000000
MPI_Bcast       0       1       0.021456        0.000000
                1       1       0.021393        0.000000
MPI_Finalize    0       1       0.000240        0.000000
                1       1       0.001194        0.000000
MPI_Reduce      0       1       0.000297        0.000000
                1       1       0.000408        0.000000
---------------------------------------------------------

Routine Summary by Rank and Peer:

Routine         Rank    Peer    Calls   Overhead    Blocking
-----------------------------------------------------------
MPI_Bcast       0       0       1       0.021456    0.000000
                1       0       1       0.021393    0.000000
MPI_Reduce      0       0       1       0.000297    0.000000
                1       0       1       0.000408    0.000000
-----------------------------------------------------------

Message Summary:

Routine        Message Bin        Count
---------------------------------------------------------
```

```
MPI_Bcast       [0..32]           2
MPI_Reduce      [0..32]           2
-------------------------------------------------------

Message Summary by Rank:

Routine       Rank    Message Bin      Count
-------------------------------------------------------
MPI_Bcast     0       [0..32]          1
              1       [0..32]          1
MPI_Reduce    0       [0..32]          1
              1       [0..32]          1
-------------------------------------------------------

Message Summary by Rank and Peer:

Routine      Rank    Peer    Message Bin      Count
-------------------------------------------------------
MPI_Bcast    0       0       [0..32]          1
             1       0       [0..32]          1
MPI_Reduce   0       0       [0..32]          1
             1       0       [0..32]          1
-------------------------------------------------------
```

Using mpiview

To view an instrumentation profile in internal format, invoke the mpiview utility and load *prefix*.mpiview (in the case of the compute_pi.f application, this is compute_pi.mpiview). mpiview displays profile data in graphical form.

An example of one of the displays provided by mpiview is shown below. For more information about using mpiview, refer to the mpiview.1 man page.



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). 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 by taking snapshots while your application is 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 151 for a list of these settings.

Working with postmortem mode

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

To create a trace file, 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 49 for information about options you can specify. You must specify the `-t` option before the program name.
`-np` *#*	Specifies the number of processes to run.
program	Specifies the name of the executable to run.

When you use the `-t` option to enable trace generation, you must specify the prefix name used for each raw trace file as part of *spec*. Then, when `mpirun` is invoked, a raw trace dump, *prefix*.*n*, is created for each application process where *n* ranges from 0 to (# - 1). `MPI_Finalize` consolidates all the raw trace dump files into a single file (*prefix*.tr) that you can load into XMPI.

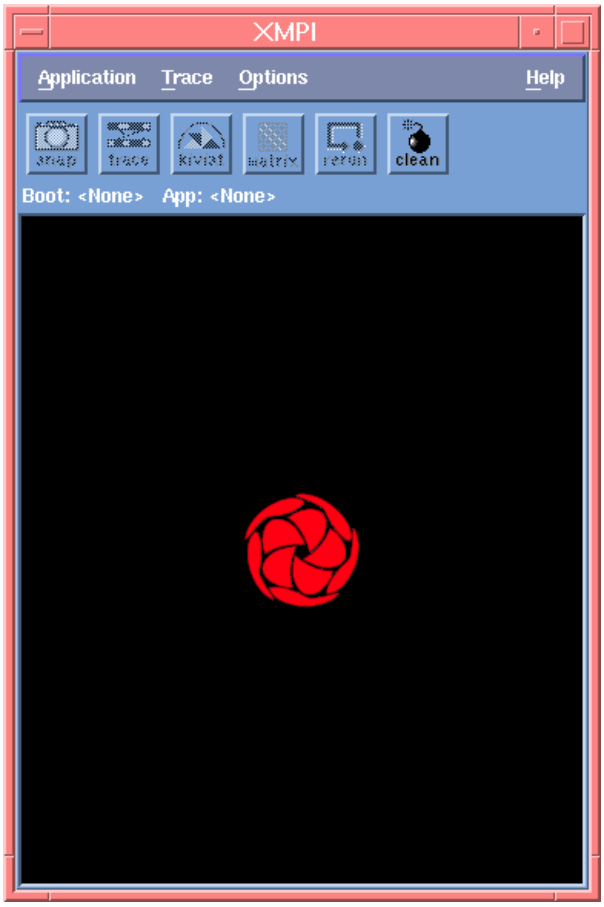
HP MPI provides the nonstandard `MPIHP_Trace_on` and `MPIHP_Trace_off` routines to help troubleshoot application problems. You insert the `MPIHP_Trace_on` and `MPIHP_Trace_off` pair around suspect code in your application. Then, you build the application and invoke `mpirun` with `-t:off` to enable application tracing. The trace information collected is only for the code between `MPIHP_Trace_on` and `MPIHP_Trace_off`. You can then run the trace file in XMPI to identify problems during application execution.

NOTE `MPIHP_Trace_on` and `MPIHP_Trace_off` are collective routines and must be called by all ranks in your application. Otherwise, the application will deadlock.

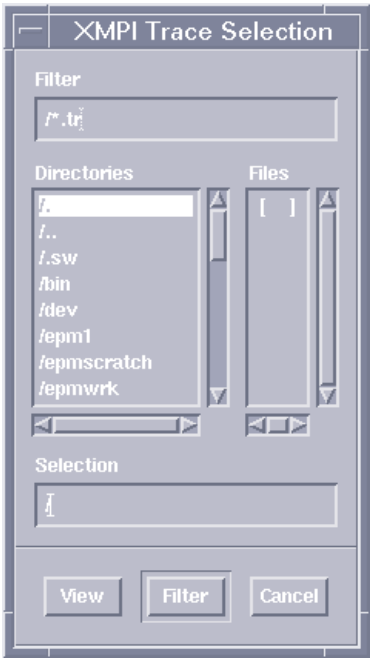
Viewing a trace file

Use these instructions to view a trace file:

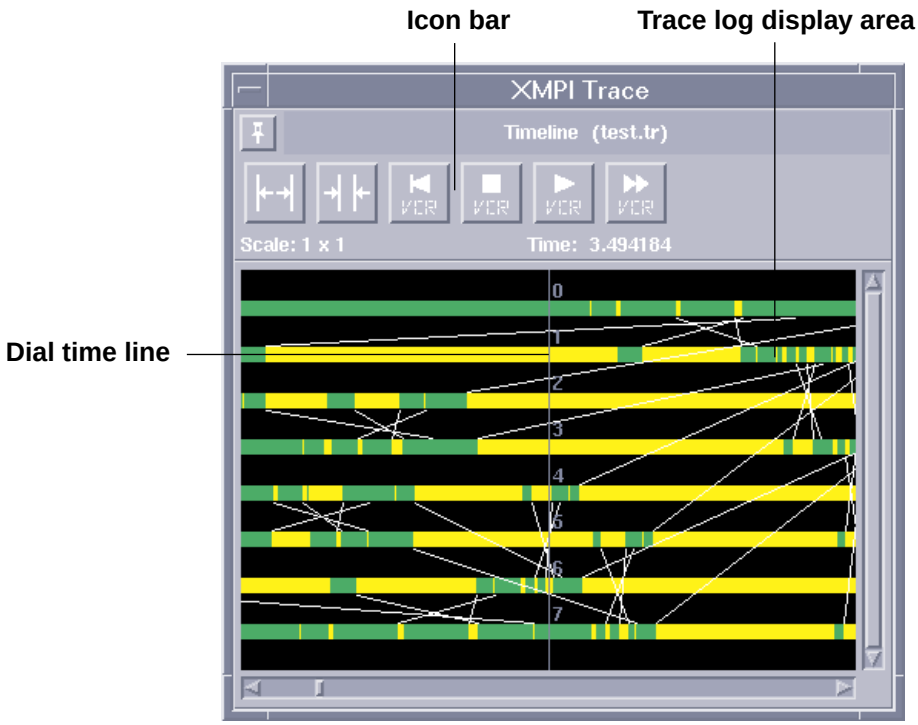
Step 1. Enter `xmpi` to open the XMPI main window (see "xmpi" on page 61 for information about other options you can specify).



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.



When viewing trace files containing multiple segments (that is, multiple `MPIHP_Trace_on` and `MPIHP_Trace_off` pairs), XMPI prompts you for the number of the segment you want to view. If you want to view a different segment later, simply reload the trace file and specify the new segment number when prompted.

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. 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. 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.

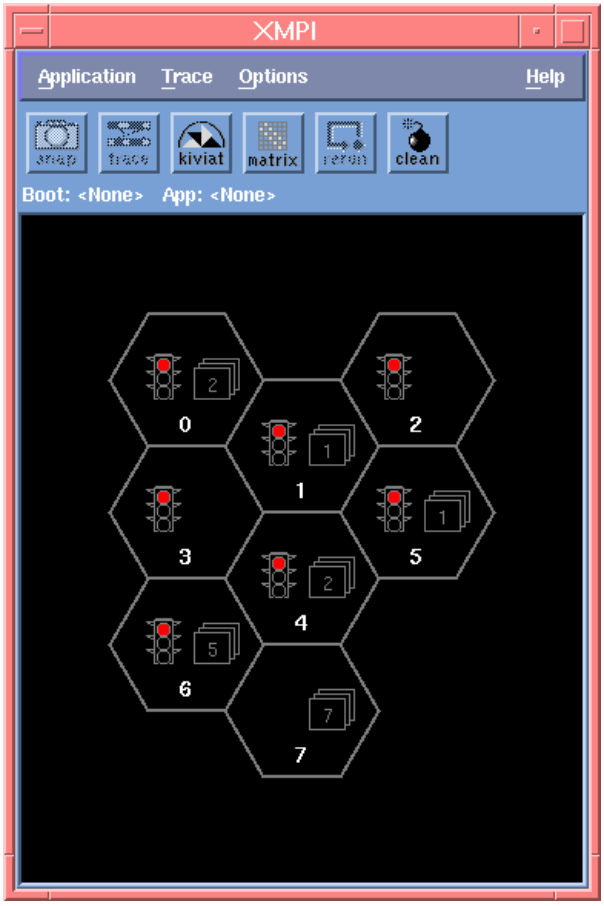
Some send and receive segments may not have a matching segment. In this case, a stub line is drawn out of the send segment or into the receive segment.

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

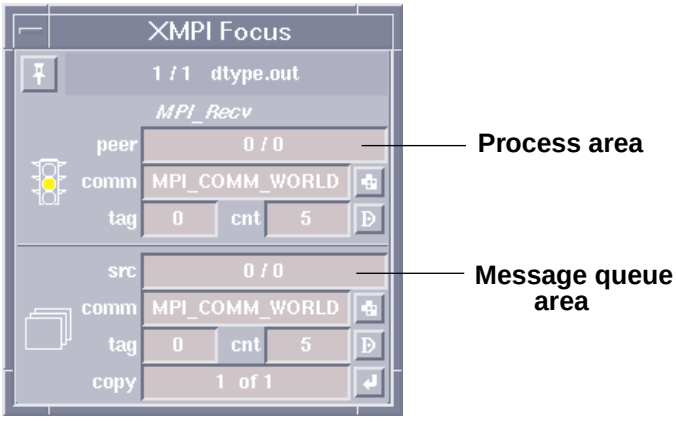
Use these instructions to view process information:

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.



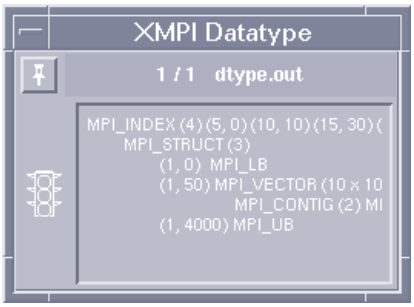
The XMPI Focus dialog consists of a process area and a message queue area.

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

peer	Displays the rank of the displayed function's peer 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 data type associated with the message when it was sent. This data type can be one of the predefined data types or a user-defined data type.

The data type shown corresponds to the current dial time of the trace file in the XMPI Trace dialog. As the trace file is played, the data type 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 data type 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 process 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 data type. 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.

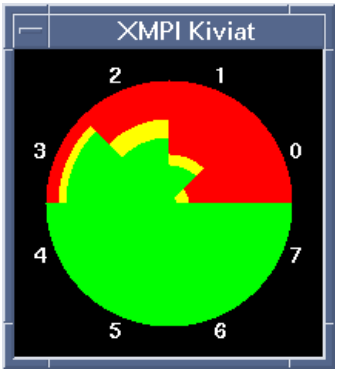
Step 3. Select Quit from the Application menu to close XMPI.

Viewing kiviat information

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 dialog 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

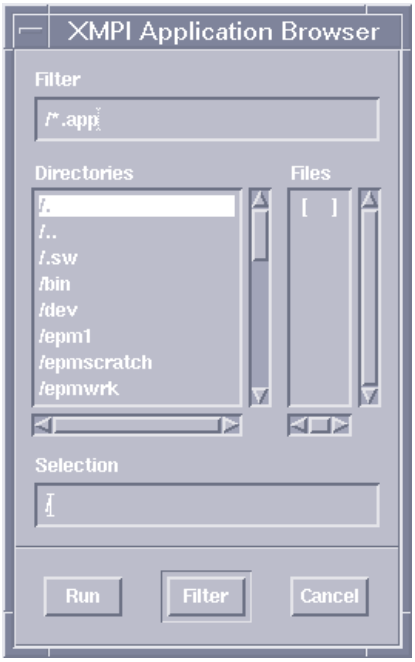
Interactive mode allows you to load and run an existing appfile to view state information for each process in your application.

Running an appfile

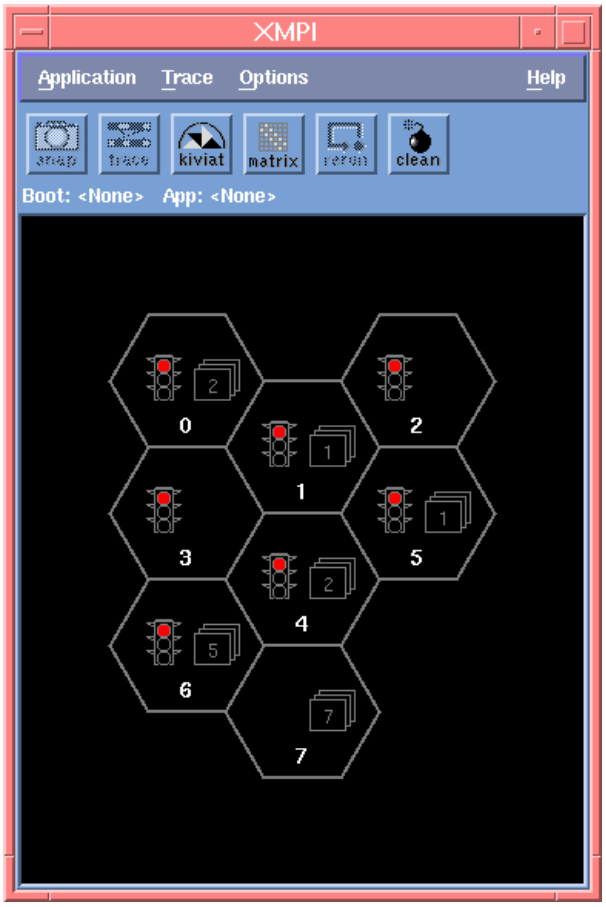
Use these instructions to run and view an appfile:

Step 1. Enter `xmpi` to open the XMPI main window (see "xmpi" on page 61 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.

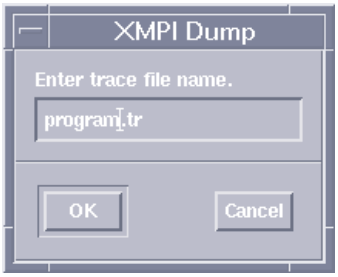
Interactive mode provides the snapshot utility to help debug applications that hang. If automatic snapshot is enabled, XMPI takes periodic snapshots of the application and displays state information for each process on the XMPI main window, the XMPI Focus dialog, and the XMPI Datatype dialog. You can use this information to view the state of each process while the application hangs.

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 take the next snapshot.

You can take snapshots only when an appfile is running. Also, you cannot replay snapshots like trace files.

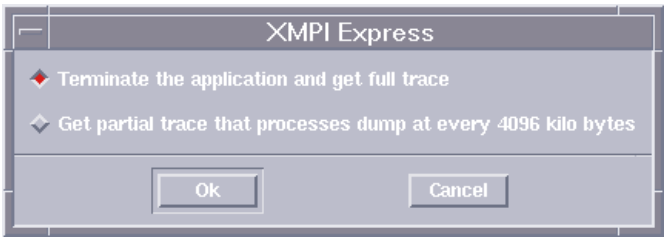
At any time while your application is running, you can select Dump from the Trace menu to open the XMPI Dump dialog.



The Dump option is only available if you have previously selected the Tracing button on the mpirun options trace dialog. Selecting Dump consolidates all raw trace-file data collected up to that point into a single .tr output file.

The single field specifies the name of the consolidated .tr output file. The value you specified for the Prefix field in the mpirun options trace dialog is automatically loaded. You can use this name or choose another. After you have created the .tr output file, you can resume snapshot monitoring.

You can also select Express from the Trace menu while your application is running to open the XMPI Express dialog.



As with the Dump option, the Express option is only available if you have previously selected the Tracing button on the mpirun options trace dialog.

The fields include:

Terminate the application and get full trace

Specifies that the contents of each process buffer (whether partial or full up to that point) are written to a raw trace file. These raw trace files are then consolidated in a .tr output file (previously specified in the Prefix field of the mpirun options trace dialog). Last, the .tr output file is loaded and displayed in the XMPI Trace dialog for viewing.

When you select this field, the XMPI Confirmation dialog displays asking if you are sure you want to terminate the application. You must select Yes before processing will continue.

After the .tr output file is loaded and displayed in the XMPI Trace dialog, you cannot resume snapshot monitoring (the application should have already terminated).

Get partial trace that processes dump at every 4096 kilobytes	Specifies that the contents of each process buffer are written to a raw trace file only after the buffer becomes full. These raw trace files are then consolidated to a .tr output file (previously specified in the Prefix field of the mpirun options trace dialog). Last, the .tr output file is loaded and displayed in the XMPI Trace dialog for viewing. After the .tr output file is loaded and displayed in the XMPI Trace dialog, you cannot resume snapshot monitoring even though the application may still be running.

When using interactive mode, XMPI gathers and displays data from the running appfile or a trace file.

When an application is running, the data source is the appfile, and automatic snapshot is enabled. Even though the application may be creating trace data, the snapshot function does not use it. Instead, the snapshot function acquires data from internal hooks in HP MPI.

At any point in interactive mode, you can load and view a trace file using the View or Express commands under the Trace menu.

When you use the View or Express commands to load and view a trace file, the data source switches to the loaded trace file, and the snapshot function is disabled. You must rerun your application to switch the data source from a trace file back to an appfile.

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. The XMPI Confirmation dialog displays asking if you are sure you want to terminate the application.

Step 5. Select Yes to terminate your application and close any associated dialogs. You can then run another application by selecting an appfile from the XMPI Application Browser dialog.

Changing default settings and viewing options

You should initially run your appfile using the XMPI default settings. You can change these default settings and your viewing options later if you like.

Use these instructions to change XMPI's default settings and your viewing options:

Step 1. Enter `xmpi` to open the XMPI main window (see "xmpi" on page 61 for information about other options you can specify).

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

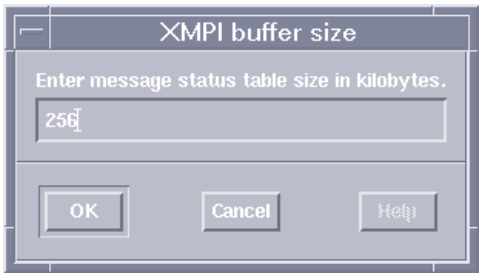
The fields include:

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. However, you can only update this information manually. 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.

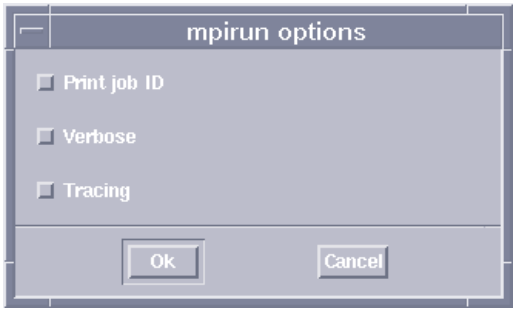
	You can enable or disable automatic snapshot while your application is running. This could be useful during troubleshooting when the application has run to a certain point and you want to disable automatic snapshot to study process state information.
Monitor interval in second	Determines how often XMPI takes a snapshot when automatic snapshot is enabled.

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



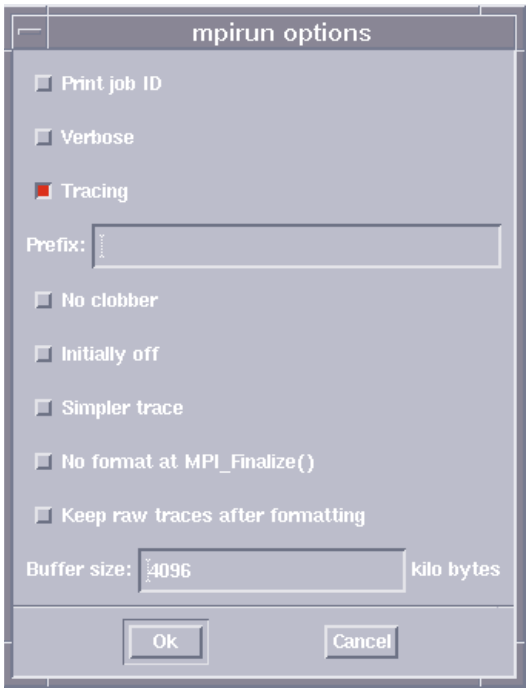
The single field specifies the size of each process buffer. When you run an application, state information for each process is stored in a separate buffer. You may need to increase buffer size if overflow problems occur.

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. This is a required field.
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 that do not complete a request.
No format at MPI_Finalize()	Specifies that raw trace files are not consolidated into a single .tr output file when `MPI_Finalize` is called. Raw trace-file consolidation can add substantially to the `MPI_Finalize` time when working with large applications.
Keep raw traces after formatting	Specifies that raw trace files are saved after they are consolidated by `MPI_Finalize`. The default is to delete raw trace files after consolidation.
Buffer size	Denotes the buffering size in kilobytes for dumping raw trace data. Actual buffering size may be rounded up by the system. The default buffering size is 4096 kilobytes. 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 increase the overhead for I/O. If this problem occurs, increase the buffering size.

Using CXperf

CXperf 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 CXperf, you can analyze the data in the performance data file using one or more of the following metrics:

- Wall clock time
- CPU time
- Execution counts
- Cache miss counts
- Latency time
- Dynamic call graph
- Migrations
- Context switches
- Page faults
- Instruction counts
- Data transaction lookaside buffer (TLB) misses
- Instruction TLB misses

You can display the data as a 3D profile, a 2D profile, a report, or a dynamic call graph. For more information, see *CXperf User's Guide* and *CXperf Command Reference*.

Using the profiling interface

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, 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 `PMPI_Wtime` to measure the elapsed time for each call. `MPI_Send` then calls `PMPI_Send` to actually send the message.

5 Tuning

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

- General tuning
- SPP-UX platform tuning

General tuning information applies to all applications running on HP-UX and SPP-UX platforms. SPP-UX tuning information applies only to applications running on that platform.

NOTE The tuning information in this chapter improves application performance in most but not all cases. Use the output from counter instrumentation or XMPI to determine which tuning changes are appropriate.

General tuning

When you developHP MPI applications, several factors can affect performance. These factors include:

- Message latency and bandwidth
- Multiple network interfaces
- Processor 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 or bandwidth or both:

- Reduce the number of process communications by designing coarse-grained applications.
- Use derived, contiguous data types for dense data structures to eliminate unnecessary byte-copy operations in certain cases. Use derived data types instead of `MPI_Pack` and `MPI_Unpack` if possible. HP MPI optimizes noncontiguous transfers of derived data types.
- Use collective operations whenever possible. This eliminates the overhead of using `MPI_Send` and `MPI_Recv` each time when one process communicates with others. Also, use the HP MPI collectives rather than customizing your own. HP MPI collectives have three-level optimizations when used in a NUMA environment.
- Specify the source process rank whenever possible when calling MPI routines. Using `MPI_ANY_SOURCE` may increase latency.

- Double-word align data buffers if possible. This improves byte-copy performance between sending and receiving processes because of double-word loads and stores.
- Use `MPI_Recv_init and MPI_Startall` instead of a loop of `MPI_Irecv` calls in cases where requests may not complete immediately.

  For example, suppose you write an application with the following code section:

```
j = 0
for (i=0; i<size; i++) {
   if (i==rank) continue;
   MPI_Irecv(buf[i], count, dtype, i, 0, comm, &requests[j++]);
}
MPI_Waitall(size-1, requests, statuses);
```

  Suppose that one of the iterations through `MPI_Irecv` does not complete before the next iteration of the loop. In this case, HP MPI tries to progress both requests. This progression effort could continue to grow if succeeding iterations also do not complete immediately, resulting in a higher latency.

  However, you could rewrite the code section as follows:

```
j = 0
for (i=0; i<size; i++) {
   if (i==rank) continue;
   MPI_Recv_init(buf[i], count, dtype, i, 0, comm,
&requests[j++]);
}
MPI_Startall(size-1, requests);
MPI_Waitall(size-1, requests, statuses);
```

  In this case, all iterations through `MPI_Recv_init` are progressed just once when `MPI_Startall` is called. This approach avoids the additional progression overhead when using `MPI_Irecv` and can reduce application latency.

Multiple network interfaces

You can use multiple network interfaces for interhost communication while still having intrahost exchanges. In this case, the intrahost exchanges use shared memory between processes mapped to different same-host IP addresses.

To use multiple network interfaces, you must specify which MPI processes are associated with each IP address in your appfile. To improve performance, use the `MPI_TOPOLOGY` environment variable to associate each network interface with the hypernode where it physically resides on SPP-UX.

For example, suppose you have two hosts called host0 and host1 that each communicate using the two HIPPI cards hippi0 and hippi1. Assume the network interfaces are named:

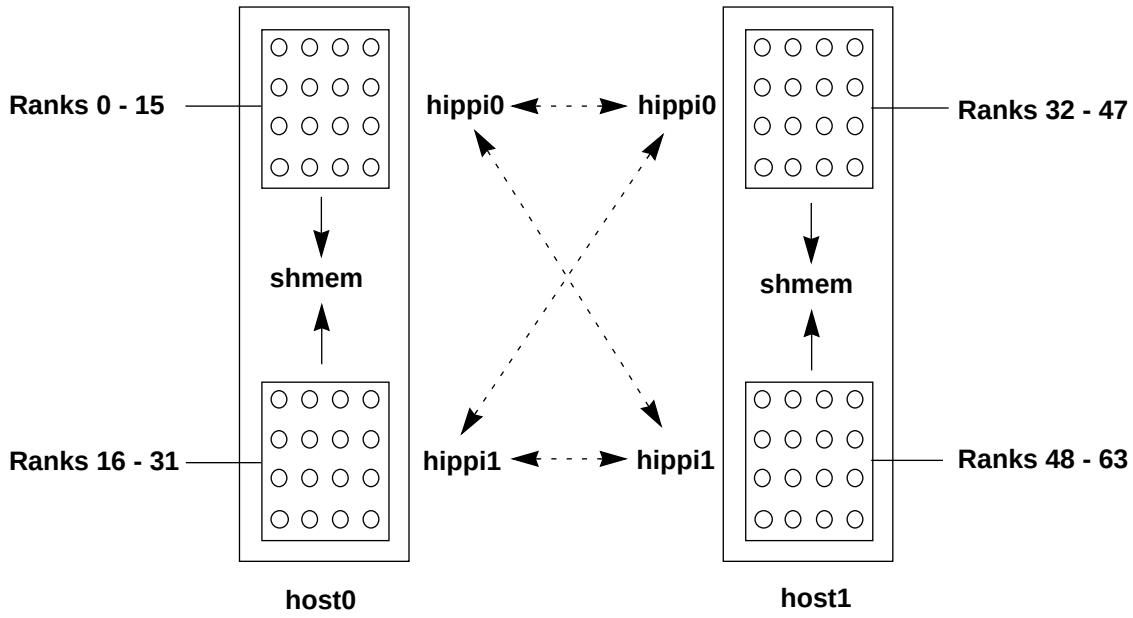
- host0-hippi0
- host0-hippi1
- host1-hippi0
- host1-hippi1

If your executable is called beavis.exe and uses 64 processes, your appfile should contain the following entries:

```
-h host0-hippi0 -e MPI_TOPOLOGY=/0:16,0 -np 16 beavis.exe
-h host0-hippi1 -e MPI_TOPOLOGY=/1:0,16 -np 16 beavis.exe
-h host1-hippi0 -e MPI_TOPOLOGY=/0:16,0 -np 16 beavis.exe
-h host1-hippi1 -e MPI_TOPOLOGY=/1:0,16 -np 16 beavis.exe
```

Now, when the appfile is run, 32 processes are run on host0 and 32 processes are run on host1 as shown in Figure 6.

Figure 6 Multiple network interfaces



Host0 processes with rank 0 - 15 communicate with processes with rank 16 - 31 through shared memory (shmem). Host0 processes also communicate through the host0-hippi0 network interface with host1 processes.

Processor subscription

Subscription refers to the match of processors and active processes on a host or subcomplex. Table 11 lists possible subscription types.

Table 11 Subscription types

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

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.

MPI routine selection

To achieve the lowest message latencies and highest message bandwidths for point-to-point synchronous communications, use the MPI blocking routines `MPI_Send` and `MPI_Recv`. For asynchronous communications, use the MPI nonblocking routines `MPI_Isend` and `MPI_Irecv`.

When using blocking routines, try to avoid pending requests. MPI must advance nonblocking messages, so calls to blocking receives must advance pending requests, occasionally resulting in lower application performance.

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

SPP-UX platform tuning

Three factors affect application performance when working with HP MPI applications on the SPP-UX platform. These factors include:

- Multilevel parallelism
- Process placement
- Topology optimization

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.

  Use the HP MPI thread-compliant library if your application has two or more threads that make MPI calls. See “Thread-compliant library” on page 29 for more information.

- Assign only one multithreaded process per hypernode when placing application processes. This ensures that enough processors are available as different process threads become active.

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. You can do this using 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 an X-Class server using a subcomplex called System. This subcomplex spans four hypernodes and contains the 20 processors listed below:

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

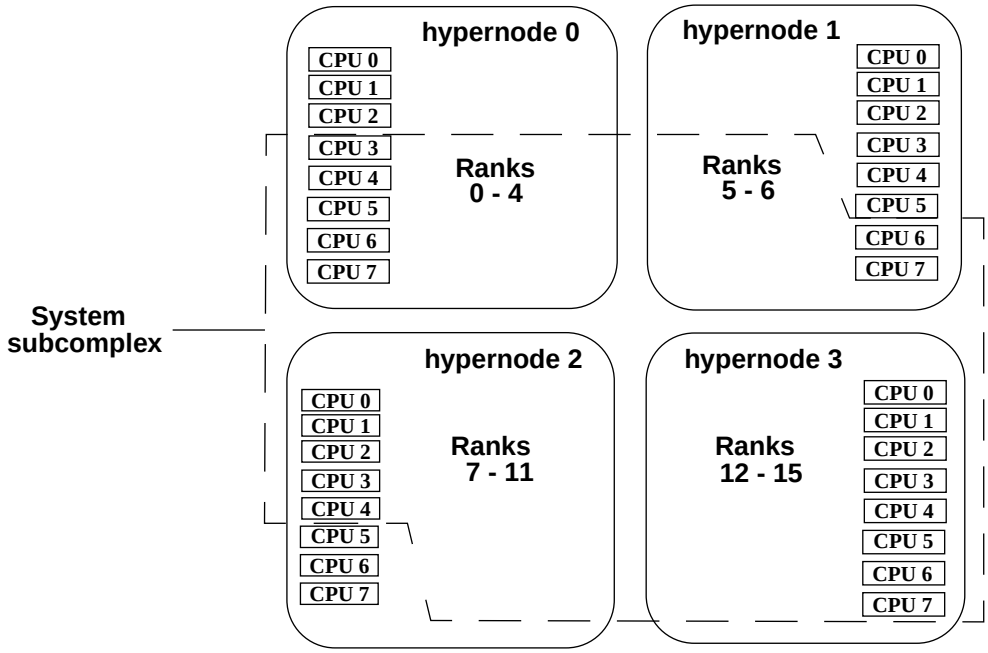
Suppose the application you want to run contains the 16 processes listed below:

- Set 1: Ranks 0-3
- Set 2: Ranks 4-7
- Set 3: Ranks 8-11
- Set 4: Ranks 12-15

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 7.

Figure 7 Default process placement



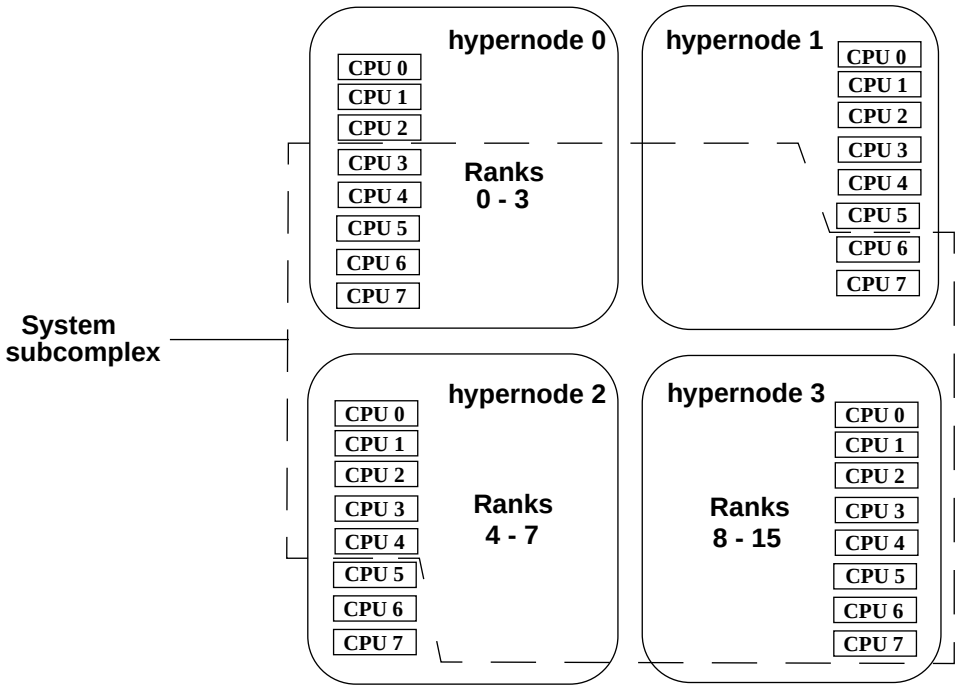
While this distribution prevents processor oversubscription, it does not provide optimum message-passing performance because the processes from sets two and three are split across hypernodes. Communications within these process groups may become a bottleneck when running the application.

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

- Hypernode 0 --> Ranks 0-3
- Hypernode 1 --> unused
- Hypernode 2 --> Ranks 4-7
- Hypernode 3 --> Ranks 8-15

This distribution results in a placement shown in Figure 8.

Figure 8 **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 47.

NOTE Make sure you use `MPI_TOPOLOGY` to place processes doing I/O on the hypernodes hosting the appropriate I/O controller. Placing these processes on noncontroller nodes results in lower I/O performance.

Topology optimization

The `MPI_Cart_create` and `MPI_Graph_create` routines are used to create Cartesian topologies. Both routines take a boolean input argument called *reorder*, which specifies whether processes are reordered to improve the mapping of the virtual topology onto the physical machine.

Reordering in this manner reduces the overall communication cost for a given topology. Specifically, latency is reduced as processes that communicate the most are co-located on the same host.

If *reorder* is set to true, you can generate a report of the optimizations performed by specifying the `-o` option in the `MPI_FLAGS` environment variable. See “MPI_FLAGS” on page 43 for more information.

For example, suppose you write a 128-way application called cartesian.c, and cartesian.c makes the call

```
MPI_Cart_create(MPI_COMM_WORLD, 2, [4 32], [true true],
true)
```

Next, you compile cartesian.c and decide to reorder the binary on an X-Class server with 128 CPUs. You also want to generate a report of the optimizations performed during the reordering process. You enter:

```
% setenv MPI_FLAGS -o MPI_TOPOLOGY System\
/0:16,16,16,16,16,16,16,16
```

to set the appropriate environment variables and then run the application using:

```
% cartesian -np 128
```

Part of the optimization report generated during reordering is shown below.

```
Default mapping of processes would result communication paths
        between hosts                          :   0
        between subcomplexes                   :   0
        between hypernodes                     : 136
        between CPUs within a hypernode/SMP    : 120

Default mapping results communication paths
        between hosts                          :   0
        between subcomplexes                   :   0
        between hypernodes                     :  32
        between CPUs within a hypernode/SMP    : 224
```

6 Debugging and troubleshooting

This chapter describes debugging and troubleshooting HP MPI applications. The topics covered are:

- Debugging HP MPI applications
- Troubleshooting HP MPI applications
- Frequently asked questions

Debugging HP MPI applications

HP MPI provides single-process debuggers to debug applications running on SPP-UX and HP-UX platforms. You access these debuggers by setting options in the `MPI_FLAGS` environment variable.

To use a single-process debugger:

Step 1. Set the `exdb`, `edde`, `egdb`, or `ecxdb` options in the `MPI_FLAGS` environment variable to use the XDB, DDE, GDB, or CXdb options respectively (refer to “MPI_FLAGS” on page 43 for information about these options).

On remote hosts, set DISPLAY to point to your console. In addition, use `xhost` to allow remote hosts to redirect their windows to your console.

Step 2. Run your application. When `MPI_Init` is executed, HP MPI starts one debugger session per process.

Step 3. Set a breakpoint anywhere following `MPI_Init` in each session.

Step 4. Set the global variable `MPI_DEBUG_CONT` to 1 using each session’s command-line interface or graphical user interface. For example:

```
(CXdb) fill MPI_DEBUG_CONT \; 1
(dde) set -language c MPI_DEBUG_CONT = 1
(xdb) print *MPI_DEBUG_CONT = 1
(gdb) set MPI_DEBUG_CONT = 1
```

Step 5. Issue the appropriate debugger command in each session to continue program execution. Each process runs and stops at the breakpoint after `MPI_Init` that you set earlier.

Step 6. Debug the execution of each process using the appropriate commands for your debugger.

Using the Diagnostics Library

HP MPI provides a diagnostics library (DLIB) for advanced run-time error checking and analysis. DLIB provides the following checks:

- Message signature analysis—Detects type mismatches in MPI calls. For example, in the two calls below, the send operation sends an integer, but the matching receive operation receives a floating-point number.

```
if (rank == 1) then
  MPI_Send(&buf1, 1, MPI_INT, 2, 17, MPI_COMM_WORLD);
else if (rank == 2)
  MPI_Recv(&buf2, 1, MPI_FLOAT, 1, 17, MPI_COMM_WORLD, &status);
```

- MPI object-space corruption—Detects attempts to write into objects such as MPI_Comm, MPI_Datatype, MPI_Request, MPI_Group, and MPI_Errhandler.
- Multiple buffer writes—Detects whether the data type specified in a receive or gather operation causes MPI to write to a user buffer more than once.

To disable these checks or enable formatted or unformatted printing of message data to a file, you must set the `MPI_DLIB_FLAGS` environment variable. See “MPI_DLIB_FLAGS” on page 45 for more information.

To use the diagnostics library, specify the `-ldmpi` option when compiling your application.

NOTE Using DLIB reduces application performance. Also, you cannot use DLIB with XMPI tracing and instrumentation.

Troubleshooting HP MPI applications

This section describes hints and limitations when you work with HP MPI applications. Check this information first when you troubleshoot 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.

Avoid using the `+autodbl` option when compiling Fortran 77 applications. This option may lead to unpredictable results.

Starting

When starting multihost applications, make sure that:

- All remote hosts are listed in your .rhosts file on each machine
- Application binaries are available on the necessary remote hosts and are executable on those machines
- The `-sp` option is passed to `mpirun` using an appfile if necessary
- The .cshrc file does not contain `tty` commands such as `stty` if you are using a /bin/csh-based shell.

Running

Run-time problems originate from many sources. These sources include:

- Propagation of environment variables
- Shared memory
- Interoperability
- Message buffering
- External input and output
- Fortran 90 programming features
- UNIX open file descriptors

Propagation of environment variables

When working with applications that run on multiple hosts, you must set values for environment variables on each host that participates in the job.

A recommended way to accomplish this is to set the `-e` option in the appfile:

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

Alternatively, you can set environment variables using the .cshrc file on each remote host if you are using a /bin/csh-based shell.

Shared memory

When an MPI application starts, each MPI process attempts to allocate a section of shared memory. This allocation can fail if the system-imposed limit on the maximum number of allowed shared-memory identifiers is exceeded or if the amount of available physical memory is not sufficient to fill the request.

After shared-memory allocation is done, every MPI process attempts to attach to the shared-memory region of every other process residing on the same host. This attachment can fail if the number of shared-memory segments attached to the calling process exceeds the system-imposed limit. In this case, use the `MPI_GLOBMEMSIZE` environment variable to reset your shared-memory allocation.

Furthermore, all processes must be able to attach to a shared-memory region at the same virtual address. For example, if the first process to attach to the segment attaches at address ADR, then the virtual-memory region starting at ADR must be available to all other processes. Placing `MPI_Init` to execute first can help avoid this problem. A process with a large stack size is also prone to this failure. Choose process stack size carefully.

Interoperability

Depending upon what server resources are available, applications may run on heterogeneous systems such that certain portions run on SPP-UX platforms and other portions run on HP-UX platforms.

For example, suppose you create an 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 standard, message buffering may or may not occur when processes communicate with each other using `MPI_Send`. Therefore, you should take care 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 to an external drive. 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 12 for the hello_world application. All multihost invocations use an appfile called hello_world.

Table 12 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 features

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 of 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. However, 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` waits until all processes have exited. If an application detects an MPI error that leads to program termination, it calls `MPI_Abort` instead.

You may want to code your error conditions using `MPI_Abort`, which cleans up the application.

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 and shared-memory segments. In this context, you use the job ID from `mpirun` to specify the application to terminate.

Frequently asked questions

This section describes frequently asked HP MPI questions.

1. How do I find out what hypernode each MPI process is running on in applications running under SPP-UX?

   ANSWER: Use the node_num(3) interface in the AIL library, the `pot` utility, or the `syspic` utility. The AIL library is searched automatically when you use the HP MPI compilation scripts `mpicc`, `mpif77`, `mpiCC`, and `mpif90`.

2. My application times out before invoking `MPI_Init`. I get the message:

```
mpirun: cannot accept connection: Connection timed out
```

   I am running on a single system, not a cluster. What might be causing this?

   ANSWER: `mpirun` makes some assumptions about how long it will take before a process calls `MPI_Init`. If the process does not call `MPI_Init` in time, `mpirun` assumes there is an error.

   In general, MPI programs should call `MPI_Init` before doing anything else. This helps ensure that all processes respond to `mpirun` in time, and that they will be able to attach shared memory at the same addresses.

3. When I build with HP MPI and then turn tracing on, the application takes a long time inside `MPI_Finalize`. This was not happening previously. What is causing this?

   ANSWER: `MPI_Finalize` now consolidates the raw trace generated by each process into a single output file (with a .tr extension). Previously, you had to invoke `mpitrget` explicitly to consolidate raw traces. You can instruct HP MPI to not merge traces by specifying the `nf` option using the `MPI_XMPI` environment variable. For example:

   % **`setenv MPI_XMPI`** *prefix*`:nf`

4. How does HP MPI clean up when something goes wrong?

   ANSWER: HP MPI uses several mechanisms to clean up job files. Note that all processes in your application must call `MPI_Finalize`.

- When a correct HP MPI program (that is, one that calls `MPI_Finalize`) exits successfully, the root host deletes the job file.
- If `mpirun` was used, it deletes the job file when the application terminates, whether successfully or not.
- When an application calls `MPI_Abort`, the job file is deleted.
- If you use `mpijob -j` to get more information on a job, and the processes of that job have all exited, `mpijob` issues a warning that the job has completed, and deletes the job file.

5. The documentation for HP MPI says that the syntax for the `MPI_TOPOLOGY` environment variable is [[sc][hypernode]:][topology]. Based on this, I would expect both of the following two commands to handle a program run on two nodes:

   % **setenv MPI_TOPOLOGY System/0:1,1**

   or

   % **setenv MPI_TOPOLOGY System/0:0,2**

   The first command works fine and gives me:

```
happy04 [89] mpirun -w -np 2 hello_world
Hello world! I'm 1 of 2 on happy04
Hello world! I'm 0 of 2 on happy04
```

   But the second command does not work and gives me the following error:

```
mpirun -w -np 2 hello_world
hello_world: Pid 3261: MPI_Init: MPI_TOPOLOGY: Logical startup
node 0 must have an assigned process in the topology process
list [see MPI.1, TROUBLESHOOTING]
hello_world: Pid 3261: MPI_Init: MPI_TOPOLOGY=System/0:0,2
hello_world: Pid 3261: MPI_Init: Aborting the application
```

   ANSWER: The problem is that the operating system can place processes on any node. In particular, it is biased against placing processes on node 0 because node 0 normally has a higher load than other nodes. So, the operating system spawns the root process of hello_world on node 1.

   However, in the second command, `MPI_TOPOLOGY` forces both processes to start on virtual node 0, resulting in startup failure. You can avoid this problem if you place the root process on node 0 by entering:

```
% mpa -node 0 hello_world -np 2
```

or

```
% mpa -node 0 mpirun -np 2 hello_world
```

You can also set MPI_TOPOLOGY to System/1:0,2 to use the default subcomplex. In this case, there is no need to use mpa, just mpirun. mpirun will migrate the application to node 1 as follows:

```
% mpirun -np 2 -e MPI_TOPOLOGY=System/1:0,2\
hello_world
```

Because this command does not depend on mpa, it is portable between HP-UX and SPP-UX (MPI_TOPOLOGY is ignored on HP-UX, so setting it there has no effect).

6. My MPI application hangs at MPI_Send. Why?

   ANSWER: Check to see if your code assumes buffering behavior for standard communication mode. Deadlock situations may occur when standard send operations are used.

7. My MPI processes need to run with mpa. How do I do that?

   ANSWER: The answer depends on whether you use mpirun or the *executable* -np # syntax to run your application. If you use the *executable* -np # syntax, you can invoke your application with mpa as follows:

   ```
   % mpa -DATA n -STACK m executable -np #
   ```

   If you use mpirun, you can create a shell script with your mpa execution line and include that script in your mpi file. For example:

   ```
   % mpirun -np 32 mpa -DATA n -STACK m executable
   ```

A Example applications

This appendix provides example applications that supplement the information in “MPI concepts” on page 4. The examples included are shown in Table 13.

Table 13 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
cart.C	C++	Generates a virtual topology.	-np = 4
communicator.c	C	Copies the default communicator MPI_COMM_WORLD.	-np = 2
multi_par.f	Fortran 77	Uses the alternating direction iterative (ADI) method on a 2-dimensional compute region.	-np >= 1
io.c	C	Writes data for each process to a separate file called iodata*x*, where *x* represents each process rank in turn. Then, the data in iodata*x* is read back.	-np >= 1
thread_safe.c	C	Tracks the number of client requests handled and prints a log of the requests to stdout.	-np >= 2

These examples and their Makefile 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:

Step 1. Change to a writable directory.

Step 2. Enter

```
% cp /opt/mpi/help/* .
```

Step 3. Enter `make` to build and run all the examples or `make` *example_name* to build and run a specific application example.

Step 4. Enter

```
% mpirun -j -w -np # program
```

where ***program*** specifies the path to the executable created in step 3.

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. The buffers are aligned and offset from each other to avoid cache conflicts caused by direct process-to-process byte-copy operations

To run this example:

- Define the CHECK macro to check data integrity.
- Increase the number of bytes to at least twice the cache size to obtain representative bandwidth measurements.

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

#define NLOOPS          1000
#define ALIGN           4096

main(argc, argv)

        int argc;
        char *argv[];

{
        int i, j;
        double start, stop;
        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;
```

ping_pong.c

```
/*
 * 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);

/*
 * 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
 */
                start = MPI_Wtime();
                for (i = 0; i < NLOOPS; i++) {
#ifdef CHECK
                        for (j = 0; j < nbytes; j++) buf[j] =
(char) (j + i);
#endif
    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);


#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
                }
                stop = MPI_Wtime();

                printf("%d bytes: %.2f usec/msg\n",
                  nbytes, (stop - start) / NLOOPS / 2 * 1000000);
```

```
                if (nbytes > 0) {
                        printf("%d bytes: %.2f MB/sec\n", nbytes,
                                nbytes / 1000000. /
                                ((stop - start) / NLOOPS / 2));
                }
        }
        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);
}
```

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 = 3.14159265358979323846264338327950288419716939937510d0)


      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, status, 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 -1
        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!
```

cart.C

This C++ program generates a virtual topology. The class Node represents a node in a 2-D torus. Each process is assigned a node or nothing. Each node holds integer data, and the shift operation exchanges the data with its neighbors. Thus, north-east-south-west shifting returns the initial data.

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

#define NDIMS   2

typedef enum { NORTH, SOUTH, EAST, WEST } Direction;

// A node in 2-D torus
class Node {
private:
  MPI_Comm      comm;
  int           dims[NDIMS], coords[NDIMS];
  int           grank, lrank;
  int           data;
public:
  Node(void);
  ~Node(void);
  void profile(void);
  void print(void);
  void shift(Direction);
};

// A constructor
Node::Node(void)
{
  int i, nnodes, periods[NDIMS];

// Create a balanced distribution
MPI_Comm_size(MPI_COMM_WORLD, &nnodes);
for (i = 0; i < NDIMS; i++) { dims[i] = 0; }
MPI_Dims_create(nnodes, NDIMS, dims);


// Establish a cartesian topology communicator
for (i = 0; i < NDIMS; i++) { periods[i] = 1; }
MPI_Cart_create(MPI_COMM_WORLD, NDIMS, dims, periods, 1, &comm);

// Initialize the data
MPI_Comm_rank(MPI_COMM_WORLD, &grank);
if (comm == MPI_COMM_NULL) {
  lrank = MPI_PROC_NULL;
  data = -1;
} else {
  MPI_Comm_rank(comm, &lrank);
```

```
  data = lrank;
  MPI_Cart_coords(comm, lrank, NDIMS, coords);
}
}

// A destructor
Node::~Node(void)
{
  if (comm != MPI_COMM_NULL) {
    MPI_Comm_free(&comm);
  }
}

// Shift function
void Node::shift(Direction dir)
{
if (comm == MPI_COMM_NULL) { return; }

int direction, disp, src, dest;
if (dir == NORTH) {
  direction = 0; disp = -1;
} else if (dir == SOUTH) {
  direction = 0; disp = 1;
} else if (dir == EAST) {
  direction = 1; disp = 1;
} else {
  direction = 1; disp = -1;
}

MPI_Cart_shift(comm, direction, disp, &src, &dest);
MPI_Status stat;
MPI_Sendrecv_replace(&data, 1, MPI_INT, dest, 0, src, 0, comm,
&stat);
}

// Synchronize and print the data being held
void Node::print(void)
{
  if (comm != MPI_COMM_NULL) {
    MPI_Barrier(comm);
    if (lrank == 0) { puts(""); } // line feed
    MPI_Barrier(comm);
    printf("(%d, %d) holds %d\n", coords[0], coords[1], data);
  }
}

// Print object's profile
void Node::profile(void)
{
  // Non-member does nothing
  if (comm == MPI_COMM_NULL) { return; }

  // Print "Dimensions" at first
  if (lrank == 0) {
    printf("Dimensions: (%d, %d)\n", dims[0], dims[1]);
  }
  MPI_Barrier(comm);
```

```
  // Each process prints its profile
  printf("global rank %d: cartesian rank %d, coordinate (%d,
%d)\n",
         grank, lrank, coords[0], coords[1]);
}

// Program body

//
// Define a torus topology and demonstrate shift operations.
//
void body(void)
{
  Node node;

  node.profile();

  node.print();

  node.shift(NORTH);
  node.print();
  node.shift(EAST);
  node.print();
  node.shift(SOUTH);
  node.print();
  node.shift(WEST);
  node.print();
}

//
// Main program---it is probably a good programming practice to
call
//   MPI_Init() and MPI_Finalize() here.
//
int main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);
  body();
  MPI_Finalize();
}
```

cart output

The output from running the cart executable is shown below. The application was run with -np = 4.

```
Dimensions: (2, 2)
global rank 0: cartesian rank 0, coordinate (0, 0)
global rank 2: cartesian rank 2, coordinate (1, 0)
global rank 3: cartesian rank 3, coordinate (1, 1)
global rank 1: cartesian rank 1, coordinate (0, 1)

(0, 0) holds 0
(0, 1) holds 1
(1, 0) holds 2
(1, 1) holds 3

(0, 0) holds 2
(0, 1) holds 3
(1, 0) holds 0
(1, 1) holds 1

(0, 0) holds 3
(0, 1) holds 2
(1, 0) holds 1
(1, 1) holds 0

(0, 0) holds 1
(0, 1) holds 0
(1, 0) holds 3
(1, 1) holds 2

(0, 0) holds 0
(1, 1) holds 3
(1, 0) holds 2
(0, 1) holds 1
```

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.f

The ADI method is often used to solve differential equations. In this example, multi_par.f, access to CPSlib is required, and you must set MPI_TOPOLOGY so the system allocates one process per hypernode. A script file, multi_par.sh, is included to help automate this task.

multi_par.f implements the following logic for a 2-dimensional compute region:

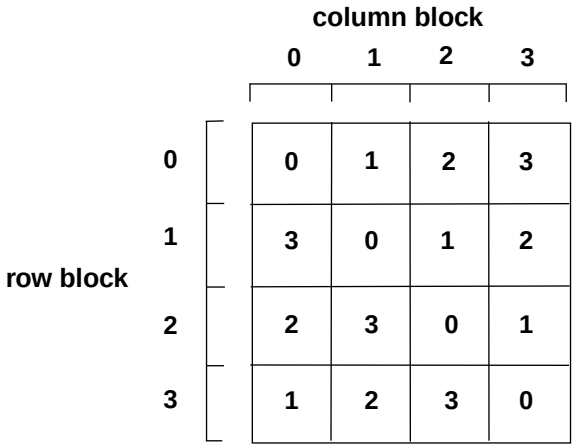
```
DO J=1,JMAX
  DO I=2,IMAX
    A(I,J)=A(I,J)+A(I-1,J)
  ENDDO
ENDDO

DO J=2,JMAX
  DO I=1,IMAX
    A(I,J)=A(I,J)+A(I,J-1)
  ENDDO
ENDDO
```

There are loop-carried dependencies in the first inner DO loop (the array's rows) and the second outer DO loop (the array's columns).

Partitioning the array into column sections supports parallelization of the first outer loop. Partitioning the array into row sections supports parallelization of the second outer loop. However, this approach requires a massive data exchange among processes because of run-time partition changes.

In this case, twisted-data layout partitioning is a better approach because the partitioning used for the parallelization of the first outer loop can accommodate the partitioning of the second outer loop. The partitioning of the array is shown in Figure 9.

Figure 9 Array partitioning



In this sample program, the rank *n* process is assigned to the partition *n* at distribution initialization. Because these partitions are not contiguous-memory regions, MPI's derived datatype is used to define the partition layout to the MPI system.

Each process starts with computing summations in row-wise fashion. For example, the rank 2 process starts with the block that is on the 0th-row block and 2nd-column block (denoted as [0,2]).

The block computed in the second step is [1,3]. Computing the first row elements in this block requires the last row elements in the [0,3] block (computed in the first step in the rank 3 process). Thus, the rank 2 process receives the data from the rank 3 process at the beginning of the second step. Note that the rank 2 process also sends the last row elements of the [0,2] block to the rank 1 process that computes [1,2] in the second step. By repeating these steps, all processes finish summations in row-wise fashion (the first outer-loop in the illustrated program).

The second outer-loop (the summations in column-wise fashion) is done in the same manner. For example, at the beginning of the second step for the column-wise summations, the rank 2 process receives data from the rank 1 process that computed the [3,0] block. The rank 2 process also sends the last column of the [2,0] block to the rank 3 process. Note that each process keeps the same blocks for both of the outer-loop computations.

This approach is good for distributed memory architectures on which repartitioning requires massive data communications that are expensive. However, on shared memory architectures, the partitioning of the compute region does not imply data distribution. The row- and column-block partitioning method requires just one synchronization at the end of each outer loop.

For distributed shared-memory architectures on X-class servers, the mix of the two methods can be effective. The sample program implements the twisted-data layout method with MPI and the row- and column-block partitioning method with CPSlib. Each MPI process spawns symmetric threads and assigns a block of rows or columns in its computing block to each of threads. Because the computation in the first outer loop does not have dependencies between columns, a thread that is assigned to a block of columns can execute without synchronization. In the second outer-loop, each thread is assigned a block of rows.

Because there is no need for an MPI process to access other processes' memory, it can use node-private memory. However, threads spawned by a process need to share memory. Given this, the strategy here is to execute MPI processes one per hypernode, assign the computation region on node-private memory, and spawn threads on the node it is running.

```
      implicit none
      include 'mpif.h'
      integer nrow                      ! # of rows
      integer ncol                      ! # of columns
      parameter(nrow=1000,ncol=1000)
      double precision array(nrow,ncol) ! compute region

c     Allocate the compute region in node-private memory
      $dir node_private(array)
      integer blk                       ! block iteration counter
      integer rb                        ! row block number
      integer cb                        ! column block number
      integer nrb                       ! next row block number
      integer ncb                       ! next column block number
      integer rbs(:)                    ! row block start subscripts
      integer rbe(:)                    ! row block end subscripts
      integer cbs(:)                    ! column block start subscripts
      integer cbe(:)                    ! column block end subscripts
      integer rdtype(:)                 ! row block communication datatypes
      integer cdtype(:)                 ! column block communication datatypes
      integer twdtype(:)                ! twisted distribution datatypes
      integer ablen(:)                  ! array of block lengths
      integer adisp(:)                  ! array of displacements
      integer adtype(:)                 ! array of datatypes
      allocatable rbs,rbe,cbs,cbe,rdtype,cdtype,twdtype,ablen,adisp,
     *     adtype
      integer rank                      ! rank iteration counter
      integer comm_size                 ! number of MPI processes
```

```
      integer comm_rank                ! sequential ID of MPI process
      integer ierr                     ! MPI error code
      integer mstat(mpi_status_size)   ! MPI function status
      integer src                      ! source rank
      integer dest                     ! destination rank
      integer dsize                    ! size of double precision in bytes
      double precision startt,endt,elapsed ! time keepers
      integer ncpus                    ! # of CPUs on the node
      integer acpus                    ! total # of CPUs
      integer params(4)                ! parameters for thread creation
      integer rc                       ! cps funtion return code
      external compcolumn,comprow         ! subroutines execute in threads
c     CPS functions
      integer cps_node_cpus,cps_ppcalln
c$dir sync_routine(cps_node_cpus,cps_ppcalln)
c
c     MPI initialization
c
      call mpi_init(ierr)
      call mpi_comm_size(mpi_comm_world,comm_size,ierr)
      call mpi_comm_rank(mpi_comm_world,comm_rank,ierr)
c
c     CPS setup
c
      ncpus=cps_node_cpus()
      if (ncpus.le.0) then
         write(7,*) 'More than 1 CPUs must be available on the node ',
     *        'on which rank',comm_rank,' is running'
         call mpi_abort(mpi_comm_world,1,ierr)
      else
         write(6,*) ncpus,' threads in rank',comm_rank,' process'
         call mpi_reduce(ncpus,acpus,1,mpi_integer,mpi_sum,0,
     *        mpi_comm_world,ierr)
      endif
c
c     Allocate threads on the same node on which calling processes
c     execute so all of them can access the compute region that is
c     assigned in the node-private memory. The following parameter
c     vector is passed to the symmetric thread-spawn function.
c
      params(1)=-1       ! alloc. threads on same node as calling thread
      params(2)=ncpus    ! min. # of threads
      params(3)=ncpus    ! max. # of threads
      params(4)=1        ! alloc. threads per node
c
c     Data initialization and start up
c
      if (comm_rank.eq.0) then
         write(6,*) 'Initializing',nrow,' x',ncol,' array...'
         call getdata(nrow,ncol,array)
         write(6,*) 'Start computation with total of',acpus,' threads'
         startt=mpi_wtime()
      endif
```

```
c
c     Compose MPI datatypes for row/column send-receive
c
c     Because rows are assigned contiguously in memory, row-wise
c     communication regions are defined as MPI's contiguous datatype
c     while column-wise regions are defined as MPI's vector datatype.
c
      allocate(rbs(0:comm_size-1),rbe(0:comm_size-1),cbs(0:comm_size-1),
     *     cbe(0:comm_size-1),rdtype(0:comm_size-1),
     *     cdtype(0:comm_size-1),twdtype(0:comm_size-1))
      do blk=0,comm_size-1
         call blockasgn(1,nrow,comm_size,blk,rbs(blk),rbe(blk))
         call mpi_type_contiguous(rbe(blk)-rbs(blk)+1,
     *        mpi_double_precision,rdtype(blk),ierr)
         call mpi_type_commit(rdtype(blk),ierr)
         call blockasgn(1,ncol,comm_size,blk,cbs(blk),cbe(blk))
         call mpi_type_vector(cbe(blk)-cbs(blk)+1,1,nrow,
     *        mpi_double_precision,cdtype(blk),ierr)
         call mpi_type_commit(cdtype(blk),ierr)
      enddo

c
c     Compose MPI datatypes for gather/scatter
c
c     Each block of the partitioning is defined as a set of fixed-length
c     vectors. Each process's partition is defined as a structure of such
c     blocks.
c
      allocate(adtype(0:comm_size-1),adisp(0:comm_size-1),
     *     ablen(0:comm_size-1))
      call mpi_type_extent(mpi_double_precision,dsize,ierr)
      do rank=0,comm_size-1
         do rb=0,comm_size-1
            cb=mod(rb+rank,comm_size)
            call mpi_type_vector(cbe(cb)-cbs(cb)+1,rbe(rb)-rbs(rb)+1,
     *           nrow,mpi_double_precision,adtype(rb),ierr)
            call mpi_type_commit(adtype(rb),ierr)
            adisp(rb)=((rbs(rb)-1)+(cbs(cb)-1)*nrow)*dsize
            ablen(rb)=1
         enddo
         call mpi_type_struct(comm_size,ablen,adisp,adtype,
     *        twdtype(rank),ierr)
         call mpi_type_commit(twdtype(rank),ierr)
         do rb=0,comm_size-1
            call mpi_type_free(adtype(rb),ierr)
         enddo

      enddo
      deallocate(adtype,adisp,ablen)

c
c     Scatter initial data using derived datatypes defined above
c     for the partitioning. MPI_send() and MPI_recv() determine the
c     layout of the data from these datatypes. This saves application
c     programs from having to manually pack/unpack the data. More importantly,
c     it provides opportunities for optimal communication
c     strategies.
```

```
c
      if (comm_rank.eq.0) then
         do dest=1,comm_size-1
            call mpi_send(array,1,twdtype(dest),dest,0,mpi_comm_world,
     *           ierr)
         enddo
      else
         call mpi_recv(array,1,twdtype(comm_rank),0,0,mpi_comm_world,
     *        mstat,ierr)
      endif

c
c     Computation
c
c
c     Sum up in each column.
c     Each MPI process, or a rank, computes blocks that it is assigned.
c     The column block number is assigned in the variable 'cb'. The
c     starting and ending subscripts of the column block 'cb' are
c     stored in 'cbs(cb)' and 'cbe(cb)' respectively. The row block
c     number is assigned in the variable 'rb'. The starting and ending
c     subscripts of the row block 'rb' are stored in 'rbs(rb)' and
c     'rbe(rb)' respectively.
c
      src=mod(comm_rank+1,comm_size)
      dest=mod(comm_rank-1+comm_size,comm_size)
      ncb=comm_rank
      do rb=0,comm_size-1
         cb=ncb

c
c     Compute a block with threads.
c     cps_ppcalln() spawns symmetric threads, executes a function in
c     parallel, and automatically joins the threads. Note that all MPI
c     processes execute this code
c
         rc=cps_ppcalln(params,compcolumn,8,ncpus,nrow,ncol,array,
     *        rbs(rb),rbe(rb),cbs(cb),cbe(cb))
         if (rc.le.0) then
            write(7,*) 'rank',comm_rank,
     *           ' cps_ppcalln(compcolumn) returned',rc
            call mpi_abort(mpi_comm_world,1,ierr)
         endif
         if (rb.lt.comm_size-1) then

c
c     Send the last row of the block to the rank that computes the
c     block next to the computed block. Receive the last row of the
c     block that the next block being computed depends on.
c
            nrb=rb+1
            ncb=mod(nrb+comm_rank,comm_size)
            call mpi_sendrecv(array(rbe(rb),cbs(cb)),1,cdtype(cb),dest,
     *           0,array(rbs(nrb)-1,cbs(ncb)),1,cdtype(ncb),src,0,
     *           mpi_comm_world,mstat,ierr)
         endif
      enddo
```

```
c
c     Sum up in each row.
c     The same logic as the loop above except rows and columns are
c     switched.
c
      src=mod(comm_rank-1+comm_size,comm_size)
      dest=mod(comm_rank+1,comm_size)
      do cb=0,comm_size-1
         rb=mod(cb-comm_rank+comm_size,comm_size)
         rc=cps_ppcalln(params,comprow,8,ncpus,nrow,ncol,array,
     *        rbs(rb),rbe(rb),cbs(cb),cbe(cb))
         if (rc.le.0) then
            write(7,*) 'rank',comm_rank,
     *           ' cps_ppcalln(comprow) returned',rc
            call mpi_abort(mpi_comm_world,1,ierr)
         endif
         if (cb.lt.comm_size-1) then
            ncb=cb+1
            nrb=mod(ncb-comm_rank+comm_size,comm_size)
            call mpi_sendrecv(array(rbs(rb),cbe(cb)),1,rdtype(rb),dest,
     *           0,array(rbs(nrb),cbs(ncb)-1),1,rdtype(nrb),src,0,
     *           mpi_comm_world,mstat,ierr)
         endif
      enddo

c
c     Gather computation results
c
      if (comm_rank.eq.0) then
         do src=1,comm_size-1
            call mpi_recv(array,1,twdtype(src),src,0,mpi_comm_world,
     *           mstat,ierr)
         enddo
         endt=mpi_wtime()
         elapsed=endt-startt
         write(6,*) 'Computation took',elapsed,' seconds'
      else
         call mpi_send(array,1,twdtype(comm_rank),0,0,mpi_comm_world,
     *        ierr)
      endif

c
c     Dump to a file
c
c      if (comm_rank.eq.0) then
c         print*,'Dumping to adi.out...'
c         open(8,file='adi.out',form='unformatted')
c         write(8,*) array
c         close(8,status='keep')
c      endif

c
c     Free the resources
c
      do rank=0,comm_size-1
         call mpi_type_free(twdtype(rank),ierr)
      enddo
```

```
      do blk=0,comm_size-1
         call mpi_type_free(rdtype(blk),ierr)
         call mpi_type_free(cdtype(blk),ierr)
      enddo
      deallocate(rbs,rbe,cbs,cbe,rdtype,cdtype,twdtype)

c
c     Finalize the MPI system
c
      call mpi_finalize(ierr)
      end

c*************************************************************      subroutine
blockasgn(subs,sube,blockcnt,nth,blocks,blocke)

c
c     This subroutine
c     is given a range of subscripts and the total number of blocks in
c     which the range is to be divided and assigns a subrange to the caller
c     that is n-th member of the blocks.
c
      implicit none
      integer subs           ! (in)      subscript start
      integer sube           ! (in)      subscript end
      integer blockcnt       ! (in)      block count
      integer nth            ! (in)      my block (begin from 0)
      integer blocks         ! (out)     assigned block start subscript
      integer blocke         ! (out)     assigned block end subscript

c
      integer d1,m1
c
      d1=(sube-subs+1)/blockcnt
      m1=mod(sube-subs+1,blockcnt)
      blocks=nth*d1+subs+min(nth,m1)
      blocke=blocks+d1-1
      if(m1.gt.nth)blocke=blocke+1
      end

c*************************************************************      subroutine
compcolumn(ncpus,nrow,ncol,array,rbs,rbe,cbs,cbe)

c
c     This subroutine
c     does summations of columns in a thread.
c
      implicit none
      integer ncpus                       ! # of cpus on node
      integer nrow                        ! length of row
      integer ncol                        ! length of column
      double precision array(nrow,ncol)   ! region
      integer rbs                         ! row block start subscript
      integer rbe                         ! row block end subscript
      integer cbs                         ! column block start subscript
      integer cbe                         ! column block end subscript

c
      integer stid,cps_stid               ! ID of the symmetric thread
```

```
c$dir sync_routine(cps_stid)
c
c     Local variables
c
      integer mycbs,mycbe               ! my column start/end index
      integer i,j

c
c     Assign a range of columns to this thread and compute
c
      stid=cps_stid()
      call blockasgn(cbs,cbe,ncpus,stid,mycbs,mycbe)
      do j=mycbs,mycbe
         do i=max(2,rbs),rbe
            array(i,j)=array(i-1,j)+array(i,j)
         enddo
      enddo
      end
c*************************************************************      subroutine
comprow(ncpus,nrow,ncol,array,rbs,rbe,cbs,cbe)

c
c     This subroutine
c     does summations of rows in a thread.
c
      implicit none
      integer ncpus                     ! # of cpus on node
      integer nrow                      ! length of row
      integer ncol                      ! length of column
      double precision array(nrow,ncol)  ! region
      integer rbs                       ! row block start subscript
      integer rbe                       ! row block end subscript
      integer cbs                       ! column block start subscript
      integer cbe                       ! column block end subscript

c
      integer stid,cps_stid             ! ID of the symmetric thread

c$dir sync_routine(cps_stid)
c
c     Local variables
c
      integer myrbs,myrbe               ! my row start/end index
      integer i,j

c
c     Assign a range of rows to this thread and compute
c
      stid=cps_stid()
      call blockasgn(rbs,rbe,ncpus,stid,myrbs,myrbe)
      do j=max(2,cbs),cbe
         do i=myrbs,myrbe
            array(i,j)=array(i,j-1)+array(i,j)
         enddo
      enddo
      end
```

multi_par.f

```
c**************************************************************      subroutine
getdata(nrow,ncol,array)

c
c     Enter dummy data
c
      integer nrow,ncol
      double precision array(nrow,ncol)
c
      do j=1,ncol
         do i=1,nrow
            array(i,j)=(j-1.0)*ncol+i
         enddo
      enddo
      end
```

io.c

In this C example, each process writes to a separate file called iodata*x*, where *x* represents each process rank in turn. Then, the data in iodata*x* is read back.

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

#define SIZE (65536)
#define FILENAME "iodata"

main(argc, argv)

     int argc;
     char **argv;

{
        int *buf, i, rank, nints, len, flag;
        char *filename;
        MPI_File fh;
        MPI_Status status;

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

        buf = (int *) malloc(SIZE);
        nints = SIZE/sizeof(int);
        for (i=0; i<nints; i++) buf[i] = rank*100000 + i;

        /* each process opens a separate file called FILENAME.'myrank' */

        filename = (char *) malloc(strlen(FILENAME) + 10);
        sprintf(filename, "%s.%d", FILENAME, rank);

        MPI_File_open(MPI_COMM_SELF, filename,
                      MPI_MODE_CREATE | MPI_MODE_RDWR,
                      MPI_INFO_NULL, &fh);

        MPI_File_set_view(fh, (MPI_Offset)0, MPI_INT, MPI_INT, "native",
                          MPI_INFO_NULL);
        MPI_File_write(fh, buf, nints, MPI_INT, &status);
        MPI_File_close(&fh);

        /* reopen the file and read the data back */

        for (i=0; i<nints; i++) buf[i] = 0;
        MPI_File_open(MPI_COMM_SELF, filename,
                      MPI_MODE_CREATE | MPI_MODE_RDWR,
                      MPI_INFO_NULL, &fh);
```

```
        MPI_File_set_view(fh, (MPI_Offset)0, MPI_INT, MPI_INT, "native",
                          MPI_INFO_NULL);
        MPI_File_read(fh, buf, nints, MPI_INT, &status);
        MPI_File_close(&fh);

        /* check if the data read is correct */
        flag = 0;
        for (i=0; i<nints; i++)
                if (buf[i] != (rank*100000 + i)) {
                        printf("Process %d: error, read %d, should be %d\n",
                               rank, buf[i], rank*100000+i);
                        flag = 1;
                }

        if (!flag) {
                printf("Process %d: data read back is correct\n", rank);
                MPI_File_delete(filename, MPI_INFO_NULL);
        }

        free(buf);
        free(filename);

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

io output

The output from running the io executable is shown below. The application was run with `-np` = 4.

```
Process 1: data read back is correct
Process 3: data read back is correct
Process 2: data read back is correct
Process 0: data read back is correct
```

thread_safe.c

In this C example, N clients loop MAX_WORK times. As part of a single work item, a client must request service from one of Nservers at random. Each server keeps a count of the requests handled and prints a log of the requests to stdout.

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

#define MAX_WORK        40
#define SERVER_TAG      88
#define CLIENT_TAG      99
#define REQ_SHUTDOWN    -1

static int service_cnt = 0;
int process_request(request)
int request;

{
      if (request != REQ_SHUTDOWN) service_cnt++;
      return request;
}

void* server(args)
void *args;

{
        int rank, request;
        MPI_Status status;
        rank = *((int*)args);

        while (1) {
                MPI_Recv(&request, 1, MPI_INT, MPI_ANY_SOURCE,
                         SERVER_TAG, MPI_COMM_WORLD, &status);

                if (process_request(request) == REQ_SHUTDOWN)
                break;

                    MPI_Send(&rank, 1, MPI_INT, status.MPI_SOURCE,
                             CLIENT_TAG, MPI_COMM_WORLD);

                printf("server [%d]: processed request %d for client %d\n",
                       rank, request, status.MPI_SOURCE);
       }

       printf("server [%d]: total service requests: %d\n", rank, service_cnt);
       return (void*) 0;
}
```

```
void client(rank, size)
int rank;
int size;

{
        int w, server, ack;
        MPI_Status status;

        for (w = 0; w < MAX_WORK; w++) {
               server = rand()%size;

        MPI_Sendrecv(&rank, 1, MPI_INT, server, SERVER_TAG, &ack,
                     1,MPI_INT,server,CLIENT_TAG,MPI_COMM_WORLD, &status);

        if (ack != server) {
                   printf("server failed to process my request\n");
                   MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER);
                }
        }
}

void shutdown_servers(rank)
int rank;

{
         int request_shutdown = REQ_SHUTDOWN;
         MPI_Barrier(MPI_COMM_WORLD);
         MPI_Send(&request_shutdown, 1, MPI_INT, rank, SERVER_TAG, MPI_COMM_WORLD);
}

main(argc, argv)
int argc;
char *argv[];

{
        int rank, size, rtn;
        pthread_t mtid;
        MPI_Status      status;
        int my_value, his_value;

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

        rtn = pthread_create(&mtid, 0, server, (void*)&rank);
        if (rtn != 0) {
                printf("pthread_create failed\n");
                MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER);
        }

        client(rank, size);
        shutdown_servers(rank);
```

```
        rtn = pthread_join(mtid, 0);
        if (rtn != 0) {
                printf("pthread_join failed\n");
                MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER);
        }

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

thread_safe output

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

```
server [1]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [0]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [1]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [1]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [0]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [1]: processed request 0 for client 0
```

```
server [0]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [0]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [1]: processed request 1 for client 1
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [0]: total service requests: 48
server [1]: total service requests: 32
```

B XMPI resource file

This appendix displays the contents of the XMPI Xresource 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 Xresource database from scratch, enter:

% **xrdb** ***filename***

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

% **x-rdb -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*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.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.foreground:black
XMPI*kiviat_draw.background:gray
XMPI*kiviat_draw.foreground:black
XMPI*matrix_draw.background:gray
XMPI*matrix_draw.foreground:black
XMPI*app_list.visibleItemCount:8
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

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.

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.

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.

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 data types User-defined structures that specify a sequence of basic data types and integer displacements for noncontiguous data. You create derived data types 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 data types.

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.

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

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

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.

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

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.

instrumentation Cumulative statistical information collected and stored in ascii format. Instrumentation is the recommended method for collecting profiling data.

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.

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 as 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 1.2 is the current standard supported by major vendors.

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).

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.

parallel efficiency An increase in speed 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.

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.

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

scatter One-to-many operation where the root's send buffer is partitioned into *n* segments and distributed to all processes such that 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 processors 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.

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.

thread compliant An implementation where an MPI process may be multithreaded. If it is, each thread can issue MPI calls. However, the threads themselves are not separately addressable.

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