<table>
<thead>
<tr>
<th>Edition</th>
<th>MPN</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eighth</td>
<td>B6060-96013</td>
<td>Revised with HP MPI V2.0, September, 2003.</td>
</tr>
<tr>
<td>Seventh</td>
<td>B6060-96008</td>
<td>Released with HP MPI V1.8, June, 2002.</td>
</tr>
<tr>
<td>Fifth</td>
<td>B6060-96001</td>
<td>Released with HP MPI V1.6, June, 2000.</td>
</tr>
<tr>
<td>Fourth</td>
<td>B6011-90001</td>
<td>Released with HP MPI V1.5, February, 1999.</td>
</tr>
<tr>
<td>Third</td>
<td>B6011-90001</td>
<td>Released with HP MPI V1.4, June, 1998.</td>
</tr>
<tr>
<td>First</td>
<td>B6011-90001</td>
<td>Released with HP MPI V1.1, January, 1997.</td>
</tr>
</tbody>
</table>
Notice

Reproduction, adaptation, or translation without prior written permission is prohibited, except as allowed under the copyright laws.

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

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

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

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

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

4. Profiling
   Using counter instrumentation .............................................. 75
   Creating an instrumentation profile ..................................... 75
   Viewing ASCII instrumentation data .................................... 76
   Using the profiling interface ........................................... 80
   Fortran profiling interface .............................................. 80

5. Tuning
   MPI_FLAGS options .......................................................... 85
   Message latency and bandwidth ......................................... 86
   Multiple network interfaces ............................................ 88
   Processor subscription .................................................... 90
   MPI routine selection ..................................................... 91
   Multilevel parallelism ..................................................... 92
   Coding considerations ................................................... 93

6. Debugging and troubleshooting
   Debugging HP MPI applications .......................................... 97
   Using Visual MPI ............................................................ 97
   Using a single-process debugger ........................................ 98
   Using a multi-process debugger ......................................... 99
   Using the diagnostics library ........................................... 102
   Enhanced debugging output .............................................. 103
   Backtrace functionality .................................................. 103
   Troubleshooting HP MPI applications ................................ 105
   Building ................................................................. 105
   Starting ................................................................. 106
   Running ................................................................. 106
   Completing .............................................................. 111
   Frequently asked questions ............................................. 112
   Time in MPI_Finalize .................................................... 112
   MPI clean up ............................................................ 112
   Application hangs in MPI_Send ........................................ 112

A. Example applications
   send_receive.f ............................................................ 119
B. Standard-flexibility in HP MPI

Glossary ................................................................. 167
Contents
Figures

Figure 3-1. Daemon communication .................................................69
Figure 4-1. ASCII instrumentation profile .........................................77
Figure 5-1. Multiple network interfaces .............................................89
Figure A-1. Array partitioning .........................................................136
Figures
Tables

Table 1. Revision history .................................................. ii
Table 2. Typographic conventions ...................................... xvi
Table 1-1. Six commonly used MPI routines .......................... 5
Table 1-2. MPI blocking and nonblocking calls ...................... 10
Table 2-1. Organization of the /opt/mpi directory ................... 23
Table 2-2. Man page categories .......................................... 24
Table 3-1. Default compilers for HP-UX ................................. 27
Table 3-2. Default compilers for Linux Itanium2 ..................... 28
Table 3-3. Default compilers for Linux IA-32 ......................... 28
Table 3-4. Default compilers for Tru64UNIX ........................... 28
Table 3-5. Compilation environment variables ....................... 29
Table 5-1. Subscription types .............................................. 90
Table 6-1. Non-buffered messages and deadlock ..................... 108
Table A-1. Example applications shipped with HP MPI ............... 116
Table B-1. HP MPI implementation of standard-flexible issues .... 164
Tables
Preface

This guide describes the HP MPI (version 2.0) 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, be familiar with MPI, and with the MPI 1.2 and MPI-2 standards (*MPI: A Message-Passing Interface Standard* and *MPI-2: Extensions to the Message-Passing Interface*, respectively).

You can access HTML versions of the MPI 1.2 and 2 standards at http://www.mpi-forum.org. *This guide supplements the material in the MPI standards and MPI: The Complete Reference.*

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.
Platforms supported

HP MPI 2.0 is supported on:

- Workstations
- Midrange servers
- High-end servers

HP MPI 2.0 for HP-UX is supported on HP-UX 11i or later operating systems on PA-RISC 2.0; and HP-UX 11i Version 1.6 or later operating systems on Itanium-based platforms.

HP MPI 2.0 for Linux is supported on Red Hat Linux V7.2 operating systems on Intel IA-32 and Itanium2 platforms. HP MPI 2.0 for Linux was built and tested with Kernel series 2.4 and glibc 2.2.

HP MPI 2.0 for Tru64UNIX is supported on AlphaServers.
Notational conventions

This section describes notational conventions used in this book.

Table 2  Typographic conventions

| **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. |
| **KeyCap**          | In paragraph text, **KeyCap** indicates the keyboard keys or the user-selectable buttons on the Graphical User Interface (GUI) that you must press to execute a command. |

---

**NOTE**  A note highlights important supplemental information.

---

**CAUTION**  A caution highlights procedures or information necessary to avoid damage to equipment, damage to software, loss of data, or invalid test results.

---

xvi
Documentation resources

Documentation resources include:

- HP MPI product information available at http://www.hp.com/go/mpi
- *MPI: The Complete Reference* (2 volume set), MIT Press
- MPI 1.2 and 2.0 standards available at http://www.mpi-forum.org:
  - *MPI: A Message-Passing Interface Standard* and
  - *MPI-2: Extensions to the Message-Passing Interface*
- TotalView documents available at http://www. etnus.com:
  - *TotalView Command Line Interface Guide*
  - *TotalView User’s Guide*
  - *TotalView Installation Guide*
- *Parallel Programming Guide for HP-UX Systems*
- The official site of the MPI forum at http://www.mpi-forum.org
- Argonne National Laboratory’s MPICH implementation of MPI at http://www-unix.mcs.anl.gov/Projects/mpi/index.html
- Argonne National Laboratory’s implementation of MPI I/O at http://www-unix.mcs.anl.gov/romio
- University of Notre Dame’s LAM implementation of MPI at http://www.lam-mpi.org/
- Vampir product information at http://www.pallas.com
- LSF product information at http://www.platform.com
- Quadrics product information at http://www.quadrics.com
Credits

HP MPI is based on MPICH from Argonne National Laboratory and LAM from the University of Notre Dame and Ohio Supercomputer Center.

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

This chapter provides a brief introduction about basic Message Passing Interface (MPI) concepts and the HP implementation of MPI.
This chapter contains the syntax for some MPI functions. Refer to *MPI: A Message-Passing Interface Standard* for syntax and usage details for all MPI standard functions. Also refer to *MPI: A Message-Passing Interface Standard* and to *MPI: The Complete Reference* for in-depth discussions of MPI concepts. The introductory topics covered in this chapter include:

- The message passing model
- MPI concepts
  - Point-to-point communication
  - Collective operations
  - MPI datatypes and packing
  - Multilevel parallelism
  - Advanced topics
The message passing model

Programming models are generally categorized by how memory is used. In the shared memory model each process accesses a shared address space, while in the message passing model an application runs as a collection of autonomous processes, each with its own local memory. In the message passing model processes communicate 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 used widely on parallel computers with distributed memory, and on clusters of servers. 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 memory 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.

In 1994, representatives from the computer industry, government labs, and academe developed a standard specification for interfaces to a library of message-passing routines. This standard is known as MPI 1.0 (*MPI: A Message-Passing Interface Standard*). Since this initial standard, versions 1.1 (June 1995), 1.2 (July 1997), and 2.0 (July 1997) have been produced. Versions 1.1 and 1.2 correct errors and minor omissions of MPI 1.0. MPI-2 (*MPI-2: Extensions to the Message-Passing Interface*) adds new functionality to MPI 1.2. You can find both standards in HTML format at http://www.mpi-forum.org.

MPI-1 compliance means compliance with MPI 1.2. MPI-2 compliance means compliance with MPI 2.0. Forward compatibility is preserved in the standard. That is, a valid MPI 1.0 program is a valid MPI 1.2 program and a valid MPI-2 program.
MPI concepts

The primary 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.

An MPI program consists of a set of processes and a logical communication medium connecting those processes. An MPI process cannot directly access memory in another MPI process. Inter-process communication requires calling MPI routines in both processes. MPI defines a library of routines through which MPI processes communicate. The MPI library routines provide a set of functions that support

- Point-to-point communications
- Collective operations
- Process groups
- Communication contexts
- Process topologies
- Datatype manipulation.

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

**CAUTION**

There should be no code before `MPI_Init` and after `MPI_Finalize`. Applications that violate this rule are non-portable 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 or broadcasting data from one process to other processes in a single operation. Use broadcast transfers to get better performance than with point-to-point transfers. The latter use `MPI_Send` to send data from each sending process and `MPI_Recv` to receive it at each receiving process.

The following sections briefly introduce the concepts underlying MPI library routines. For more detailed information refer to *MPI: A Message-Passing Interface Standard*. 

---

### Table 1-1 Six commonly used MPI routines

<table>
<thead>
<tr>
<th>MPI routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MPI_Init</code></td>
<td>Initializes the MPI environment</td>
</tr>
<tr>
<td><code>MPI_Finalize</code></td>
<td>Terminates the MPI environment</td>
</tr>
<tr>
<td><code>MPI_Comm_rank</code></td>
<td>Determines the rank of the calling process within a group</td>
</tr>
<tr>
<td><code>MPI_Comm_size</code></td>
<td>Determines the size of the group</td>
</tr>
<tr>
<td><code>MPI_Send</code></td>
<td>Sends messages</td>
</tr>
<tr>
<td><code>MPI_Recv</code></td>
<td>Receives messages</td>
</tr>
</tbody>
</table>
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 and is described in Chapter 3, “Point-to-Point Communication” in the MPI 1.0 standard.

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

\[ \text{total\_transfer\_time} = \text{latency} + \left( \frac{\text{message\_size}}{\text{bandwidth}} \right) \]

where

- \( \text{latency} \) 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.
- \( \text{message\_size} \) Specifies the size of the message in Mbytes.
- \( \text{bandwidth} \) Denotes the reciprocal of the time needed to transfer a byte. Bandwidth is normally expressed in Mbytes per second.

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 to transfer data. Communicators encapsulate a group of processes such that communication is restricted to processes within that group.

The default communicators provided by MPI are \texttt{MPI} COMM\_\texttt{WORLD} and \texttt{MPI} COMM\_\texttt{SELF}. \texttt{MPI} COMM\_\texttt{WORLD} contains all processes that are running when an application begins execution. Each process is the single member of its own \texttt{MPI} COMM\_\texttt{SELF} communicator.

Communicators that allow processes within a group to exchange data are termed intracomunicators. Communicators that allow processes in 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. There are several communication management functions; two of the more widely used are \texttt{MPI} COMM\_\texttt{size} and \texttt{MPI} COMM\_\texttt{rank}. The process rank is a unique
number assigned to each member process from the sequence 0 through 
(size-1), where size is the total number of processes in the 
communicator.

To determine the number of processes in a communicator, use the 
following syntax:

MPI_Comm_size (MPI_Comm comm, int *size);

where

comm Represents the communicator handle
size Represents the number of processes in the group of

comm

To determine the rank of each process in comm, use

MPI_Comm_rank(MPI_Comm comm, int *rank);

where

comm Represents the communicator handle
rank Represents an integer between zero and (size - 1)

A communicator is an argument to all communication routines. The C 
code example, “communicator.c” on page 133 displays the use 
MPI_Comm_dup, one of the communicator constructor functions, and 
MPI_Comm_free, the function that marks a communication object for 
deallocation.

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, and may only have started the message transfer operation, 
not necessarily completed it. The application may not safely reuse the 
message buffer after a nonblocking routine returns.

In nonblocking communications, the following sequence of events occurs:

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 are:

**Standard (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 (MPI_Bsend)** The sending process returns when the message is buffered in an application-supplied buffer.

Avoid using the MPI_Bsend mode because it forces an additional copy operation.

**Synchronous (MPI_Ssend)** The sending process returns only if a matching receive is posted and the receiving process has started to receive the message.

**Ready (MPI_Rsend)** The message is sent as soon as possible.

You can invoke any mode by using the appropriate routine name and passing the argument list. Arguments are the same for all modes.

For example, to code a standard blocking send, use

```c
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.

source Specifies the rank of the source 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.

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.

Examples “send_receive.f” on page 119, “ping_pong.c” on page 121, and “master_worker.f90” on page 127 all illustrate the use of standard blocking sends and receives.

NOTE

You should not assume message buffering between processes because the MPI standard does not mandate a buffering strategy. HP MPI does sometimes use buffering for MPI_Send and MPI_Rsend, but it is dependent on message size. Deadlock situations can occur when your code uses standard send operations and assumes buffering behavior for standard communication mode. Refer to “Frequently asked questions” on page 112 for an example of how to resolve a deadlock situation.
Nonblocking communication  MPI provides nonblocking counterparts for each of the four blocking send routines and for the receive routine. Table 1-2 lists blocking and nonblocking routine calls.

<table>
<thead>
<tr>
<th>Blocking mode</th>
<th>Nonblocking mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Send</td>
<td>MPI_Isend</td>
</tr>
<tr>
<td>MPI_Bsend</td>
<td>MPI_Ibsend</td>
</tr>
<tr>
<td>MPI_Ssend</td>
<td>MPI_Issend</td>
</tr>
<tr>
<td>MPI_Rsend</td>
<td>MPI_Irsend</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>MPI_Irecv</td>
</tr>
</tbody>
</table>

Nonblocking calls have the same arguments, with the same meaning as their blocking counterparts, plus an additional argument for a request.

To code a standard nonblocking send, use

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

where

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

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, that returns information about the received message, 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.

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

Collective communications have three basic patterns:

- **Broadcast and Scatter** Root sends data to all processes, including itself.
- **Gather** Root receives data from all processes, including itself.
- **Allgather and Alltoall** Each process communicates with each process, including itself.

The syntax of the MPI collective functions is designed to be consistent with point-to-point communications, but collective functions are more restrictive than point-to-point functions. Some of the important restrictions to keep in mind are:

- The amount of data sent must exactly match the amount of data specified by the receiver.
- Collective functions come in blocking versions only.
- Collective functions do not use a tag argument meaning that collective calls are matched strictly according to the order of execution.
Collective functions come in standard mode only.

For detailed discussions of collective communications refer to Chapter 4, “Collective Communication” in the MPI 1.0 standard. The following examples demonstrate the syntax to code two collective operations; a broadcast and a scatter:

To code a broadcast, use

```c
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.

For example “compute_pi.f” on page 125 uses `MPI_BCAST` to broadcast one integer from process 0 to every process in `MPI_COMM_WORLD`.

To code a scatter, use

```c
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

Computational operations do global reduction operations, such as sum, max, min, product, or user-defined functions across all members of a group. There are a number of global reduction functions:

- **Reduce** Returns the result of a reduction at one node.
- **All–reduce** Returns the result of a reduction at all nodes.
- **Reduce-Scatter** Combines the functionality of reduce and scatter operations.
- **Scan** Performs a prefix reduction on data distributed across a group.

Section 4.9, “Global Reduction Operations” in the MPI 1.0 standard describes each of these functions in detail.

Reduction operations are binary and are only valid on numeric data. Reductions are always associative but may or may not be commutative.

You can select a reduction operation from a predefined list (refer to section 4.9.2 in the MPI 1.0 standard) or define your own operation. The operations are invoked by placing the operation name, for example **MPI_SUM** or **MPI_PROD**, in **op** as described in the MPI_Reduce syntax below.

To implement a reduction, use

```c
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.
For example “compute_pi.f” on page 125 uses MPI_REDUCE to sum the elements provided in the input buffer of each process in MPI_COMM_WORLD, using MPI_SUM, and returns the summed value in the output buffer of the root process (in this case, process 0).

**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 do not overlap.

To implement a barrier, use

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

where

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

For example, “cart.C” on page 129 uses MPI_Barrier to synchronize data before printing.

**MPI datatypes and packing**

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).

When you want to transfer data that is not homogeneous, such as a structure, or that is not contiguous in memory, such as an array section, you can use derived datatypes or packing and unpacking functions:

Derived datatypes

```
Specifies a sequence of basic datatypes and integer displacements describing the data layout in memory.
You can use user-defined datatypes or predefined datatypes in MPI communication functions.
```
Packing and Unpacking functions

Provide MPI_Pack and MPI_Unpack functions so that a sending process can pack noncontiguous data into a contiguous buffer and a receiving process can unpack data received in a contiguous buffer and store it in noncontiguous locations.

Using derived datatypes is more efficient than using MPI_Pack and MPI_Unpack. However, derived datatypes cannot handle the case where the data layout varies and is unknown by the receiver, for example, messages that embed their own layout description.

Section 3.12, “Derived Datatypes” in the MPI 1.0 standard describes the construction and use of derived datatypes. The following is a summary of the types of constructor functions available in MPI:

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

After you create a derived datatype, you must commit it by calling MPI_Type_commit.

HP MPI optimizes collection and communication of derived datatypes.

Section 3.13, “Pack and unpack” in the MPI 1.0 standard describes the details of the pack and unpack functions for MPI. Used together, these routines allow you to transfer heterogeneous data in a single message, thus amortizing the fixed overhead of sending and receiving a message over the transmittal of many elements.

Refer to Chapter 3, “User-Defined Datatypes and Packing” in MPI: The Complete Reference for a discussion of this topic and examples of construction of derived datatypes from the basic datatypes using the MPI constructor functions.
Multilevel parallelism

By default, processes in an MPI application can only do one task at a time. Such processes are 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 process with multiple threads has one address space, but each process thread has 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.

The example program, “multi_par.f” on page 135 is an example of multilevel parallelism.

Advanced topics

This chapter only provides a brief introduction to basic MPI concepts. Advanced MPI topics include:

- Error handling
- Process topologies
- User-defined datatypes
- Process grouping
- Communicator attribute caching
- The MPI profiling interface

To learn more about the basic concepts discussed in this chapter and advanced MPI topics refer to *MPI: The Complete Reference* and *MPI: A Message-Passing Interface Standard*. 
2  Getting started

This chapter describes how to get started quickly using HP MPI. The semantics of building and running a simple MPI program are described, for single- and multiple-hosts. You learn how to configure your environment before running your program. You become familiar with the
file structure in your HP MPI directory.

The goal of this chapter is to demonstrate the basics to getting started using HP MPI.

For complete details about running HP MPI and analyzing and interpreting profiling data, refer to “Understanding HP MPI” on page 25 and “Profiling” on page 73. The topics covered in this chapter are:

- Configuring your environment
- Compiling and running your first application
  - Building and running on a single host
- Directory structure
Configuring your environment

If you move the HP MPI installation directory from its default location in /opt/mpi:

- Set the **MPI_ROOT** environment variable to point to the location where MPI is installed.
- Add **$MPI_ROOT/bin** to PATH.
- Add **$MPI_ROOT/share/man** to MANPATH.

MPI must be installed in the same directory on every execution host.

For HP MPI 2.0 for Tru64UNIX only:

- Add **$MPI_ROOT/lib/alpha** to **LD_LIBRARY_PATH**.
- As a user aid, **$MPI_ROOT/bin/setup** is provided as a quick tool to modify the **PATH**, **MANPATH**, and **LD_LIBRARY_PATH** environment variables. Additionally, this script will define compilation specific variables to eliminate the requirement for users to supply **-I** and **-L** arguments when using standard cc, f90, and f77 compilers.

```shell
% source $MPI_ROOT/bin/setup
```

**NOTE**

If you have HP MPI installed on an HP-UX or Tru64UNIX system and want to determine its version, use the **what** command.

The **what** command returns

- The path where HP MPI is installed
- The HP MPI version number
- The date this version was released
- The product number
- The operating system version

For example:

```shell
% what $MPI_ROOT/bin/mpicc
$MPI_ROOT/bin/mpicc:
HP MPI 02.00.00.00 (dd/mm/yyyy) B6060BA - HP-UX 11.i
```
If you have HP MPI installed on a Linux system and want to determine its version, use the following command:

```
% strings -a $MPI_ROOT/bin/mpicc | grep '@(#)'
```
Compiling and running your first application

To quickly become familiar with compiling and running HP MPI programs, start with the C version of a 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 \textit{host}” where \( r \) is a process's rank, \( s \) is the size of the communicator, and \textit{host} is the host on which the program is run. The processor name is the host name for this implementation.

The source code for hello_world.c is stored in \$\text{MPI\_ROOT}/help and is shown below.

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

void main(argc, argv)

int argc;
char *argv[];
{
    int rank, size, len;
    char name[\text{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

This example teaches you the basic compilation and run steps to execute hello_world.c on your local host with four-way parallelism. To build and run hello_world.c on a local host named jawbone:
Step 1. Change to a writable directory.

Step 2. Compile the hello_world executable file:

```
$MPI_ROOT/bin/mpicc -o hello_world $MPI_ROOT/help/hello_world.c
```

Step 3. Run the hello_world executable file:

```
$MPI_ROOT/bin/mpirun -np 4 hello_world
```

where `-np 4` specifies the number of processes to run is 4.

Step 4. Analyze hello_world output.

HP MPI prints the output from running the hello_world executable in non-deterministic order. The following is an example of the output:

```
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
```

For information on running more complex applications, refer to “Running applications” on page 34.
Directory structure

All HP MPI files are stored in the /opt/mpi directory. The directory structure is organized as described in Table 2-1.

If you move the HP MPI installation directory from its default location in /opt/mpi, set the MPI_ROOT environment variable to point to the new location. Refer to “Configuring your environment” on page 19.

Table 2-1  Organization of the /opt/mpi directory

<table>
<thead>
<tr>
<th>Subdirectory</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>bin</td>
<td>Command files for the HP MPI utilities</td>
</tr>
<tr>
<td>help</td>
<td>Source files for the example programs</td>
</tr>
<tr>
<td>include</td>
<td>Header files</td>
</tr>
<tr>
<td>lib/pa2.0</td>
<td>MPI PA-RISC 32-bit libraries</td>
</tr>
<tr>
<td>lib/pa20_64</td>
<td>MPI PA-RISC 64-bit libraries</td>
</tr>
<tr>
<td>lib/hpux32</td>
<td>MPI Itanium 32-bit libraries</td>
</tr>
<tr>
<td>lib/hpux64</td>
<td>MPI Itanium 64-bit libraries</td>
</tr>
<tr>
<td>lib/linux_ia32</td>
<td>MPI Linux 32-bit libraries</td>
</tr>
<tr>
<td>lib/linux_ia64</td>
<td>MPI Linux 64-bit libraries</td>
</tr>
<tr>
<td>lib/alpha</td>
<td>MPI Tru64UNIX 64-bit libraries</td>
</tr>
<tr>
<td>newconfig/</td>
<td>Configuration files and release notes</td>
</tr>
<tr>
<td>share/man/man1*</td>
<td>Man pages for the HP MPI utilities</td>
</tr>
<tr>
<td>share/man/man3*</td>
<td>Man pages for HP MPI library</td>
</tr>
<tr>
<td>doc</td>
<td>Release notes</td>
</tr>
</tbody>
</table>
The man pages located in the $MPI_ROOT/share/man/man1* subdirectory can be grouped into three categories: general, compilation, and run time. There is one general man page, MPI.1, that is an overview describing general features of HP MPI. The compilation and run-time man pages are those that describe HP MPI utilities.

Table 2-2 describes the three categories of man pages in the man1* subdirectory that comprise man pages for HP MPI utilities.

<table>
<thead>
<tr>
<th>Category</th>
<th>man pages</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>MPI.1</td>
<td>Describes the general features of HP MPI</td>
</tr>
<tr>
<td>Compilation</td>
<td>mpicc.1, mpiCC.1, mpif77.1, mpif90.1</td>
<td>Describes the available compilation utilities. Refer to “Compiling applications” on page 27 for more information</td>
</tr>
<tr>
<td>Runtime</td>
<td>mpiclean.1, mpidebug.1, mpienv.1, mpiexec.1, mpijob.1, mpiattrsafe.1, mpirun.1, mpistdio.1</td>
<td>Describes runtime utilities, environment variables, debugging, thread-safe and diagnostic libraries.</td>
</tr>
</tbody>
</table>
This chapter provides information about the HP MPI implementation of MPI. The topics covered include details about compiling and running your HP MPI applications:
• Compiling applications
  — Compilation utilities
  — Autodouble functionality
  — 64-bit support
  — Thread-compliant library

• Running applications
  — Running on multiple hosts using remote shell
  — Running on multiple hosts using prun (Quadrics system)
  — Types of applications
  — Runtime environment variables
  — Runtime utility commands
  — HyperFabric/HyperMessaging Protocol (HMP)
  — Communicating using daemons
  — IMPI
  — Native language support
Compiling applications

The compiler you use to build HP MPI applications depends upon which programming language you use. The HP MPI compiler utilities are shell scripts that invoke the appropriate native compiler. You can pass the pathname of the MPI header files using the \(-I\) option and link an MPI library (for example, the diagnostic or thread-compliant library) using the \(-Wl,\ -L\) or \(-l\) option.

By default, HP MPI compiler utilities include a small amount of debug information in order to allow the TotalView debugger to function. However, certain compiler options are incompatible with this debug information. Use the \(-\text{notv}\) option to exclude debug information. The \(-\text{notv}\) option will also disable TotalView usage on the resulting executable. The \(-\text{notv}\) option applies to archive libraries only.

HP MPI 2.0 now offers a \(-\text{show}\) option to compiler wrappers. When compiling by hand, run as \texttt{mpicc -show}\ and a line will print displaying exactly what the job was going to do.

Compilation utilities

HP MPI provides separate compilation utilities and default compilers for the languages shown in the following tables.

<table>
<thead>
<tr>
<th>Language</th>
<th>Utility</th>
<th>Default compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>mpicc</td>
<td>/opt/ansic/bin/cc</td>
</tr>
<tr>
<td>C++</td>
<td>mpiCC</td>
<td>/opt/aCC/bin/aCC</td>
</tr>
<tr>
<td>Fortran 77</td>
<td>mpi77</td>
<td>/opt/fortran/bin/f77</td>
</tr>
<tr>
<td>Fortran 90</td>
<td>mpi90</td>
<td>/opt/fortran90/bin/f90</td>
</tr>
</tbody>
</table>
Understanding HP MPI
Compiling applications

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

Table 3-2 Default compilers for Linux Itanium2

<table>
<thead>
<tr>
<th>Language</th>
<th>Utility</th>
<th>Default compiler if /opt/intel/compiler70 exists</th>
<th>Default compiler if /opt/intel/compiler70 does not exist</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>mpicc</td>
<td>ecc</td>
<td>/usr/bin/gcc</td>
</tr>
<tr>
<td>C++</td>
<td>mpiCC</td>
<td>ecc</td>
<td>/usr/bin/g++</td>
</tr>
<tr>
<td>Fortran 77</td>
<td>mpif77</td>
<td>efc</td>
<td>/usr/bin/g77</td>
</tr>
<tr>
<td>Fortran 90</td>
<td>mpif90</td>
<td>efc</td>
<td>f90</td>
</tr>
</tbody>
</table>

Table 3-3 Default compilers for Linux IA-32

<table>
<thead>
<tr>
<th>Language</th>
<th>Utility</th>
<th>Default compiler if /opt/intel/compiler70 exists</th>
<th>Default compiler if /opt/intel/compiler70 does not exist</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>mpicc</td>
<td>icc</td>
<td>/usr/bin/gcc</td>
</tr>
<tr>
<td>C++</td>
<td>mpiCC</td>
<td>icc</td>
<td>/usr/bin/g++</td>
</tr>
<tr>
<td>Fortran 77</td>
<td>mpif77</td>
<td>ifc</td>
<td>/usr/bin/g77</td>
</tr>
<tr>
<td>Fortran 90</td>
<td>mpif90</td>
<td>ifc</td>
<td>f90</td>
</tr>
</tbody>
</table>

Table 3-4 Default compilers for Tru64UNIX

<table>
<thead>
<tr>
<th>Language</th>
<th>Utility</th>
<th>Default compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>mpicc</td>
<td>/usr/bin/cc</td>
</tr>
<tr>
<td>C++</td>
<td>mpiCC</td>
<td>/usr/bin/cxx</td>
</tr>
<tr>
<td>Fortran 77</td>
<td>mpif77</td>
<td>/usr/bin/f77</td>
</tr>
<tr>
<td>Fortran 90</td>
<td>mpif90</td>
<td>/usr/bin/f90</td>
</tr>
</tbody>
</table>

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, set the corresponding environment variables shown in Table 3-5.

### Table 3-5 Compilation environment variables

<table>
<thead>
<tr>
<th>Utility</th>
<th>Environment variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpicc</td>
<td>MPI_CC</td>
</tr>
<tr>
<td>mpiCC</td>
<td>MPI_CXX</td>
</tr>
<tr>
<td>mpif77</td>
<td>MPI_F77</td>
</tr>
<tr>
<td>mpif90</td>
<td>MPI_F90</td>
</tr>
</tbody>
</table>

### Autodouble functionality

HP MPI 2.0 supports Fortran programs compiled 64-bit with any of the following options:

For HP-UX:
- +i8
- +r8
- +autodbl4
- +autodbl

For Linux Itanium2:
- -i2
  Set default KIND of integer variables is 2.
- -i4
  Set default KIND of integer variables is 4.
- -i8
  Set default KIND of integer variables is 8.
- -r8
  Set default size of REAL to 8 bytes.
- -r16
  Set default size of REAL to 16 bytes.
• -autodouble
  Same as -r8.

For Tru64UNIX:
• -r8
  Defines REAL declarations, constants, functions, and intrinsics as DOUBLE PRECISION (REAL*8), and defines COMPLEX declarations, constants, functions, and intrinsics as DOUBLE COMPLEX (COMPLEX*16). This option is the same as the -real_size 64 option.

• -r16
  Defines REAL and DOUBLE PRECISION declarations, constants, functions, and intrinsics as REAL*16. For f90, it also defines COMPLEX and DOUBLE COMPLEX declarations, constants, functions, and intrinsics as COMPLEX*32. This option is the same as the -real_size 128 option.

• -i8
  Makes default integer and logical variables 8-bytes long (same as the -integer_size 64 option). The default is -integer_size 32.

The decision of how the Fortran arguments will be interpreted by the MPI library is made at link time.

If the mpif90 compiler wrapper is supplied with one of the above options at link time, the necessary object files will automatically link, informing MPI how to interpret the Fortran arguments.

---

**NOTE**

This autodouble feature is supported in the regular and multithreaded MPI libraries, but not in the diagnostic library.

---

The following MPI functions accept user-defined functions:

• MPI_Op_create()
• MPI_Errhandler_create()
• MPI_Keyval_create()
• MPI_Comm_create_errhandler()
• MPI_Comm_create_keyval()
• MPI_Win_create_errhandler()
• MPI_Win_create_keyval()

The user-defined callback passed to these functions should accept normal-sized arguments. These functions are called internally by the library where normally-sized data types will be passed to them.

64-bit support

HP-UX 11.i and higher 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.i and higher. Both 32- and 64-bit versions of the library are shipped with HP-UX 11.i and higher. For HP-UX 11i and higher, 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. Use the following syntax:

```
[mpicc | mpiCC] [+DA2.0W | +DD64] -o filename filename.c
```

When you use `mpif90`, compile with the `+DA2.0W` option to link the 64-bit version of the library. Otherwise, `mpif90` links the 32-bit version. For example, to compile the program `myprog.f90` and link the 64-bit library enter:

```
% mpif90 +DA2.0W -o myprog myprog.f90
```

Thread-compliant library

HP MPI provides a thread-compliant library. By default, the non thread-compliant library (libmpi) is used when running HP MPI jobs. Linking to the thread-compliant library (libmtmpi) is now required only for applications that have multiple threads making MPI calls simultaneously. In previous releases, linking to the thread-compliant library was required for multithreaded applications even if only one thread was making a MPI call at a time. See Table B-1 on page 164.
Application types that no longer require linking to the thread-compliant library include:

- Implicit compiler-generated parallelism (e.g. +O3 +Oparallel in HP-UX)
- Thread parallel MLIB applications
- OpenMP
- pthreads (Only if no two threads call MPI at the same time. Otherwise, use the thread-compliant library for pthreads.)
Building Applications

This example shows how to build hello_world.c prior to running.

**Step 1.** Change to a writable directory.

**Step 2.** Compile the hello_world executable.

For shared libraries:

```bash
% $MPI_ROOT/bin/mpicc -o hello_world $MPI_ROOT/help/
    hello_world.c
```

For archive libraries:

On HP-UX:

```bash
% $MPI_ROOT/bin/mpicc -o hello_world $MPI_ROOT/help/
    hello_world.c -Wl,-aarchive_shared
```

On Linux:

```bash
% $MPI_ROOT/bin/mpicc -o hello_world $MPI_ROOT/help/
    hello_world.c -static
```

On Tru64UNIX:

```bash
% $MPI_ROOT/bin/mpicc -o hello_world $MPI_ROOT/help/
    hello_world.c -non_shared
```
Running applications

This section introduces the methods to run your HP MPI application. Using one of the `mpirun` methods is required. The examples below demonstrate two basic methods. Refer to “mpirun (mpirun.all)” on page 51 for all the `mpirun` command line options.

There are three methods you can use to start your application:

- Use `mpirun` with the `-np #` option and the name of your program. For example,

  ```bash
  $MPI_ROOT/bin/mpirun -np 4 hello_world
  ```

  starts an executable file named `hello_world` with four processes. This is the recommended method to run applications on a single host with a single executable file.

- Use `mpirun` with an appfile. For example,

  ```bash
  $MPI_ROOT/bin/mpirun -f appfile
  ```

  where `-f appfile` specifies a text file (appfile) that is parsed by `mpirun` and contains process counts and a list of programs.

  You can use an appfile when you run a single executable file on a single host and you must use this appfile method when you run on multiple hosts or run multiple executables. For details about building your appfile, refer to “Creating an appfile” on page 59.

- Use `mpirun` with `-prun` using the Quadrics Elan3 communication processor on Linux or Tru64UNIX. For example,

  ```bash
  $MPI_ROOT/bin/mpirun [mpirun options] -prun [prun options]
  ```

  This method is only supported when linking with shared libraries.

  This method allows full MPI-2 functionality. Some features like `mpirun -stdio` processing are still unavailable.

  The `-np` option is not allowed with `-prun`. The following `mpirun` options are allowed with `-prun:`
[-universe_size=#] [-sp <paths>] [-T] [-prot] [-spawn]
[-1sided] [-e var[=val]] -prun <prun options> <program>
[<args>]

Running on multiple hosts using remote shell

This example teaches you to run the hello_world.c application that you built in Building Applications (above) using two hosts to achieve four-way parallelism. For this example, the local host is named jawbone and a remote host is named wizard. To run hello_world.c on two hosts, use the following procedure, replacing jawbone and wizard with the names of your machines:

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. In addition to the entries in the .rhosts file, ensure that your remote machine permissions are set up so that you can use the remsh command to that machine. Refer to the HP-UX remsh(1) man page for details.

You can use the MPI_REMSH environment variable to specify a command other than remsh to start your remote processes. Refer to “MPI_REMSH” on page 49. Ensure that the correct commands and permissions are set up on all hosts.

Step 2. Insure that the executable is accessible from each host either by placing it in a shared directory or by copying it to a local directory on each host.

Step 3. Create an appfile.

An appfile is a text file that contains process counts and a list of programs. In this example, create an appfile named my_appfile containing the following two lines:

- h jawbone -np 2 /path/to/hello_world
- h wizard -np 2 /path/to/hello_world

The appfile file should contain a separate line for each host. Each line specifies the name of the executable file and the number of processes to run on the host. The -h option is followed by the name of the host where the specified processes must be run. Instead of using the host name, you may use its IP address.

Step 4. Run the hello_world executable file:
% $MPI_ROOT/bin/mpirun -f my_appfile

The -f option specifies the filename that follows it is an appfile. mpirun parses the appfile, line by line, for the information to run the program. In this example, mpirun runs the hello_world program with two processes on the local machine, jawbone, and two processes on the remote machine, wizard, as dictated by the -np 2 option on each line of the appfile.

Step 5. Analyze hello_world output.

HP MPI prints the output from running the hello_world executable in non-deterministic order. The following is an example of the output:

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

Notice that processes 0 and 1 run on jawbone, the local host, while processes 2 and 3 run on wizard. HP MPI guarantees that the ranks of the processes in MPI_COMM_WORLD are assigned and sequentially ordered according to the order the programs appear in the appfile. The appfile in this example, my_appfile, describes the local host on the first line and the remote host on the second line.

Running on multiple hosts using prun (Quadrics system)

This example teaches you to run the hello_world.c application that you built in Building Applications (above) using two hosts to achieve four-way parallelism on a Quadrics system. For this example, the local host is named jawbone and a remote host is named wizard. To run hello_world.c on two hosts, use the following procedure, replacing jawbone and wizard with the names of your machines:

Step 1. Insure that the executable is accessible from each host either by placing it in a shared directory or by copying it to a local directory on each host.

Step 2. Run the hello_world executable file:

% $MPI_ROOT/bin/mpirun -prun -N 2 -n 4 /path/to/hello_world

All options after -prun are processed directly by prun. In this example, -N to prun specifies 2 hosts are to be used and -n starts 4 total processes.
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:

% $MPI_ROOT/bin/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:

% $MPI_ROOT/bin/mpicc -o poisson poisson.c
% $MPI_ROOT/bin/mpirun -np 5 poisson

prun also supports running applications with SPMD. Please refer to the prun documentation at http://www.quadrics.com.

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 list of programs to be run and the number of processes to be created for each program.

A simple invocation of an MPMD application looks like this:

% $MPI_ROOT/bin/mpirun -f appfile
where appfile is the text file parsed by mpirun and contains a list of programs and process counts.

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 (refer to “Creating an appfile” on page 59 for details).

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

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

```
% $MPI_ROOT/bin/mpicc -o poisson_master poisson_master.c
% $MPI_ROOT/bin/mpicc -o poisson_child poisson_child.c
% $MPI_ROOT/bin/mpirun -f appfile
```

See “Creating an appfile” on page 59 for more information about using appfiles.

**Runtime environment variables**

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

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

Environment variables are always local to the system where mpirun runs. To propagate environment variables to remote hosts, specify each variable in an appfile using the -e option. See “Creating an appfile” on page 59 for more information.

Environment variables can also be set globally on the mpirun command line:

```
% $MPI_ROOT/bin/mpirun -e MPI_FLAGS=y -f appfile
```
In the above example, if some `MPI_FLAGS` setting was specified in the appfile, then the global setting on the command line would override the setting in the appfile. To add to an environment variable rather than replacing it, use the following command:

```
% $MPI_ROOT/bin/mpiexec -e MPI_FLAGS=%MPI_FLAGS,y -f appfile
```

In the above example, if the appfile specified `MPI_FLAGS=z`, then the resulting `MPI_FLAGS` seen by the application would be \( z, y \).

The environment variables that affect the behavior of HP MPI at runtime are listed below and described in the following sections:

- `MPI_COMM
- `MPI_DLIB_FLAGS
- `MPI_FLAGS
- `MP_GANG
- `MPI_GLOBMEMSIZE
- `MPI_INSTR
- `MPI_LOCALIP
- `MPI_MT_FLAGS
- `MPI_NOBACKTRACE
- `MPI_REMSH
- `MPI_SHMEMCNTL
- `MPI_TMPDIR
- `MPI_WORKDIR
- `TOTALVIEW

**MPI_COMM**

`MPI_COMM` routes all off-host communication through daemons rather than between processes. The `MPI_COMM` syntax is as follows:

`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.

Refer to “Communicating using daemons” on page 68 for more information.

**MPI_DLIB_FLAGS**

MPI_DLIB_FLAGS controls runtime options when you use the diagnostics library. The MPI_DLIB_FLAGS syntax is a comma separated list as follows:

```
[ns,][h,][strict,][nmsg,][nwarn,][dump:prefix,]
[dumpf:prefix][xNUM]
```

where

- **ns**  Disables message signature analysis.
- **h**  Disables default behavior in the diagnostic library that ignores user specified error handlers. The default considers all errors to be fatal.
- **strict**  Enables MPI object-space corruption detection. Setting this option for applications that make calls to routines in the MPI-2 standard may produce false error messages.
Understanding HP MPI
Running applications

### nmsg
Disables detection of multiple buffer writes during receive operations and detection of send buffer corruptions.

### nwarn
Disables the warning messages that the diagnostic library generates by default when it identifies a receive that expected more bytes than were sent.

### dump:prefix
Dumps (unformatted) all sent and received messages to prefixmsgs.rank where rank is the rank of a specific process.

### dumpf:prefix
Dumps (formatted) all sent and received messages to prefixmsgs.rank where rank is the rank of a specific process.

### xNUM
Defines a type-signature packing size. NUM is an unsigned integer that specifies the number of signature leaf elements. For programs with diverse derived datatypes the default value may be too small. If NUM is too small, the diagnostic library issues a warning during the MPI_Finalize operation.

Refer to “Using the diagnostics library” on page 102 for more information.

### MPI_FLAGS

**MPI_FLAGS** modifies the general behavior of HP MPI. The MPI_FLAGS syntax is a comma separated list as follows:

```
[edde,][exdb,][egdb,][eadb,][ewdb,][eladdebug,][l,][f,][i,][s[a|p][#,]][y[#],][o,][+E2,][C,][D,][E,][T,][z]
```

where

- **edde** Starts the application under the dde debugger. The debugger must be in the command search path. See “Debugging HP MPI applications” on page 97 for more information.
exdb    Starts the application under the xdb debugger. The debugger must be in the command search path. See “Debugging HP MPI applications” on page 97 for more information.

egdb    Starts the application under the gdb debugger. The debugger must be in the command search path. See “Debugging HP MPI applications” on page 97 for more information.

eadb    Starts the application under adb—the absolute debugger. The debugger must be in the command search path. See “Debugging HP MPI applications” on page 97 for more information.

ewdb    Starts the application under the wdb debugger. The debugger must be in the command search path. See “Debugging HP MPI applications” on page 97 for more information.

eladebug Starts the application under the ladebug debugger. The debugger must be in the command search path. See “Debugging HP MPI applications” on page 97 for more information.

l    Reports memory leaks caused by not freeing memory allocated when an HP MPI job is run. For example, when 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.

f    Forces MPI errors to be fatal. Using the f option sets the MPI_ERRORS_ARE_FATAL error handler, ignoring the programmer’s choice of error handlers. This option can help you detect nondeterministic error problems in your code.

If your code has a customized error handler that does not report that an MPI call failed, you will not know that a failure occurred. Thus your application could be
catching an error with a user-written error handler (or with MPI_ERRORS_RETURN) which masks a problem.

i Turns on language interoperability concerning the MPI_BOTTOM constant.

MPI_BOTTOM Language Interoperability—Previous versions of HP MPI were not compliant with Section 4.12.6.1 of the MPI-2 Standard which requires that sends/receives based at MPI_BOTTOM on a data type created with absolute addresses must access the same data regardless of the language in which the data type was created. If compliance with the standard is desired, set MPI_FLAGS=i to turn on language interoperability concerning the MPI_BOTTOM constant. Compliance with the standard can break source compatibility with some MPICH code.

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 for the MPI library is sp604800, which issues a SIGPROF once a week. If the application uses both signals for its own purposes, you must disable the heart-beat signals. A time value of zero seconds disables the heart beats.

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.

Generating a UNIX signal introduces a performance penalty every time the application processes are interrupted. As a result, while some applications will benefit from it, others may experience a decrease in performance. As part of tuning the performance of an application, you can control the behavior of the heart-beat signals by changing their time period or by turning them off. This is accomplished by setting the time period of the s option in the MPI_FLAGS environment variable (for example: s600). Time is in seconds.
You can use the `s[a]p#` option with the thread-compliant library as well as the standard non-thread-compliant library. Setting `s[a]p#` for the thread-compliant library has the same effect as setting `MPI_MT_FLAGS=ct` when you use a value greater than 0 for `#`. The default value for the thread-compliant library is `sp0`. `MPI_MT_FLAGS=ct` takes priority over the default `MPI_FLAGS=sp0`.

Refer to “MPI_MT_FLAGS” on page 48 and “Thread-compliant library” on page 31 for additional information.

Set `MPI_FLAGS=sa1` to guarantee that `MPI_Cancel` works for canceling sends.

`y[#]` enables spin-yield logic. `#` is the spin value and is an integer between zero and 10,000. The spin value specifies the number of milliseconds a process should block waiting for a message before yielding the CPU to another process.

How you apply spin-yield logic depends on how well synchronized your processes are. For example, if you have a process that wastes CPU time blocked, waiting for messages, you can use spin-yield to ensure that the process relinquishes the CPU to other processes. Do this in your appfile, by setting `y[#]` to `y0` for the process in question. This specifies zero milliseconds of spin (that is, immediate yield).

Specifying `y` without a spin value is equivalent to `MPI_FLAGS=y10000`.

If the time a process is blocked waiting for messages is short, you can possibly improve performance by setting a spin value (between 0 and 10,000,) that ensures the process does not relinquish the CPU until after the message is received, thereby reducing latency.

The system treats a nonzero spin value as a recommendation only. It does not guarantee that the value you specify is used.
Refer to “Appfiles” on page 58 for details about how to create an appfile and assign ranks.

- 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.

+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.

D
Dumps shared memory configuration information. Use this option to get shared memory values that are useful when you want to set the MPI_SHMCNTL flag.

E[on|off]
Function parameter error checking is turned off by default. It can be turned on by setting MPI_FLAGS=Eon.

T
Prints the user and system times for each MPI rank.

z
Enables zero-buffering mode. Set this flag to convert MPI_Send and MPI_Rsend calls in your code to MPI_Ssend, without rewriting your code. Refer to Troubleshooting, “Application hangs in MPI_Send” on page 112, for information about how using this option can help uncover nonportable code in your MPI application.

MP_GANG
MP_GANG enables gang scheduling on HP-UX systems only. Gang scheduling improves the latency for synchronization by ensuring that all runnable 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-schedule policy is fair overall. In addition, gangs are scheduled for a single time slice, which is the same for all processes in the system.
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.

The MP_GANG syntax is as follows:

```
[ON|OFF]
```

where

- **ON**: Enables gang scheduling.
- **OFF**: Disables gang scheduling.

For multihost configurations, you need to set MP_GANG for each appfile entry. Refer to the -e option in “Creating an appfile” on page 59.

You can also use the HP-UX utility mpsched(1) to enable gang scheduling. Refer to the HP-UX gang_sched(7) and mpsched(1) manpages for more information.

**MPI_GLOBMEMSIZE**

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

```
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 host. Otherwise, swapping overhead will degrade application performance.

**MPI_INSTR**

MPI_INSTR enables counter instrumentation for profiling HP MPI applications. The MPI_INSTR syntax is a colon-separated list (no spaces between options) as follows:

```
prefix[...][:l][:nc][:off]
```

where

- **prefix**: Specifies the instrumentation output file prefix. The rank zero process writes the application’s measurement data to `prefix`.instr in ASCII. 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.

l  Locks ranks to cpus and uses the cpu's cycle counter for less invasive timing. If used with gang scheduling, the :l is ignored.

nc  Specifies no clobber. If the instrumentation output file exists, MPI_Init aborts.

off  Specifies counter instrumentation is initially turned off and only begins after all processes collectively call MPIHP_Trace_on.

Refer to “Using counter instrumentation” on page 75 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. Refer to “mpirun (mpirun.all)” on page 51 for more information.

Counter instrumentation and trace-file generation are mutually exclusive profiling techniques.

---

**NOTE**

When you enable instrumentation for multihost runs, and invoke mpirun either on a host where at least one MPI process is running, or on a host remote from all your MPI processes, HP MPI writes the instrumentation output file (prefix.instr) to the working directory on the host that is running rank 0.

---

**MPI_LOCALIP**

MPI_LOCALIP specifies the host IP address that is assigned throughout a session. Ordinarily, mpirun determines the IP address of the host it is 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 as follows:

```
xxx.xxx.xxx.xxx
```

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

**MPI_MT_FLAGS**

`MPI_MT_FLAGS` controls runtime options when you use the thread-compliant version of HP MPI. The `MPI_MT_FLAGS` syntax is a comma-separated list as follows:

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

where

- **ct** creates a hidden communication thread for each rank in the job. When you enable 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 executes.

- **fun** asserts that a process can be multithreaded, but only the main thread makes MPI calls (that is, all calls are funneled to the main thread).

- **serial** asserts that a process can be multithreaded, and multiple threads can make MPI calls, but calls are serialized (that is, only one call is made at a time).

- **mult** asserts that multiple threads can call MPI at any time with no restrictions.

Setting `MPI_MT_FLAGS=ct` has the same effect as setting `MPI_FLAGS=s[a][p]#`, when the value of `#` that is greater than 0.

`MPI_MT_FLAGS=ct` takes priority over the default `MPI_FLAGS=sp0` setting. Refer to “MPI_FLAGS” on page 41.

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 specified is processed (in this case, the `mult` option). If no runtime option is specified, the default is `mult`.

For more information about using `MPI_MT_FLAGS` with the thread-compliant library, refer to “Thread-compliant library” on page 31.
MPI_NOBACKTRACE

On PA-RISC systems, a stack trace is printed when the following signals occur within an application:

- SIGILL
- SIGBUS
- SIGSEGV
- SIGSYS

In the event one of these signals is not caught by a user signal handler, HP MPI will display a brief stack trace that can be used to locate the signal in the code.

Signal 10: bus error
PROCEDURE TRACEBACK:
(0) 0x0000489c bar + 0xc [././a.out]
(1) 0x000048c4 foo + 0x1c [././a.out]
(2) 0x000049d4 main + 0xa4 [././a.out]
(3) 0xc013750c _start + 0xa8 [/usr/lib/libc.2]
(4) 0x0003b50 $START$ + 0x1a0 [././a.out]

This feature can be disabled for an individual signal handler by declaring a user-level signal handler for the signal. To disable for all signals, set the environment variable MPI_NOBACKTRACE:

% setenv MPI_NOBACKTRACE

See “Backtrace functionality” on page 103 for more information.

MPI_REMSH

MPI_REMSH specifies a command other than the default remsh to start remote processes. The mpirun, mpijob, and mpiclean utilities support MPI_REMSH. For example, you can set the environment variable to use a secure shell:

% setenv MPI_REMSH /bin/ssh

The alternative remote shell command should be a drop-in replacement for /usr/bin/remsh, that is, the argument syntax for the alternative shell should be the same as for /usr/bin/remsh.
**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 a comma separated list as follows:

\[ \text{nenv, frag, generic} \]

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 after mailbox and envelope allocation.
- **generic** Specifies the size in bytes of the generic-shared memory region. The default is 12.5 percent of shared memory after mailbox and envelope allocation.

**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

\[ \text{directory} \]

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

**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:

\[ \text{directory} \]

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

When you use the TotalView debugger, HP MPI uses your PATH variable to find TotalView. You can also set the absolute path and TotalView specific options in the TOTALVIEW environment variable. This environment variable is used by mpirun.

setenv TOTALVIEW /opt/totalview/bin/totalview [totalview_options]

Runtime utility commands

HP MPI provides a set of utility commands to supplement the MPI library routines. These commands are listed below and described in the following sections:

- mpirun (mpirun.all)
  
  This section also includes discussion of Appfiles, the Multipurpose daemon process, and Generating multihost instrumentation profiles.

- prun
- mpiexec
- mpijob
- mpiclean

mpirun (mpirun.all)

The HP MPI start-up provides the following advantages:

- Provides support for shared libraries
- The -e option enables environment variable settings to be specified on the command line

The HP MPI start-up mpirun requires that MPI be installed in the same directory on every execution host. The default is the location from which mpirun is executed. This can be overridden with the MPI_ROOT environment variable. We recommend setting the MPI_ROOT environment variable prior to starting mpirun. See “Configuring your environment” on page 19.

We recommend using the mpirun launch utility. However, for users that are unable to install MPI on all hosts, HP MPI provides a self-contained launch utility, mpirun.all.
The restrictions for `mpirun.all` include:

- Applications must be linked statically
- Start-up may be slower
- TotalView is unavailable to executables launched with `mpirun.all`
- Files will be copied to a temporary directory on target hosts
- The remote shell must accept stdin

**mpirun syntax has five formats:**

- For applications where all processes execute the same program on the same host:

  ```shell
  ```

  For example:

  ```shell
  % $MPI_ROOT/bin/mpirun -j -np 3 send_receive
  ```

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

- For applications that consist of multiple programs or that run on multiple hosts:

  ```shell
  ```

  In this case, each program in the application is listed in a file called an appfile. Refer to “Appfiles” on page 58 for more information.

  For example:

  ```shell
  % $MPI_ROOT/bin/mpirun -t my_trace -f my_appfile
  ```

  enables tracing, specifies the prefix for the tracing output file is my_trace, and runs an appfile named my_appfile.

- Applications using the Quadrics Elan3 communication processor on Linux or Tru64UNIX require the `-prun` option:

  ```shell
  % $MPI_ROOT/bin/mpirun [mpirun options] -prun [prun options]
  ```

  This method is only supported when linking with shared libraries.
This method allows full MPI-2 functionality. Some features like mpirun -stdio processing are still unavailable.

The -np option is not allowed with -prun. The following mpirun options are allowed with -prun:

```
[-universe_size=#] [-sp <paths>] [-T] [-prot] [-spawn]
[-1sided] [-e var[=val]] -prun <prun options> <program>
[<args>]
```

- To invoke LSF for applications where all processes execute the same program on the same host:

```
bsub [lsf_options] pam -mpi mpirun [mpirun_options] program [args]
```

In this case, LSF assigns a host to the MPI job.

For example:

```
% bsub pam -mpi $MPI_ROOT/bin/mpirun -np 4 compute_pi
```

requests a host assignment from LSF and runs the compute_pi application with four processes.

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. LSF handles the job scheduling and the allocation of the necessary hosts and HP MPI handles the task of starting up the application's processes on the hosts selected by LSF.

By default mpirun starts the MPI processes on the hosts specified by the user, in effect handling the direct mapping of host names to IP addresses. When you use LSF to start MPI applications, the host names, specified to mpirun or implicit when the -h option is not used, are treated as symbolic variables that refer to the IP addresses that LSF assigns. Use LSF to do this mapping by specifying a variant of mpirun to execute your job.
Load Sharing Facility (LSF) including PAM is currently available in PA-RISC only, but runs on Itanium2 systems in 11i (PA) mode. Thus, the Itanium2 version of HP MPI requires the use of the PA-RISC file libmpirm.sl in order to run LSF. This file is located in the $MPI_ROOT/lib/pa2.0 directory.

To invoke LSF for applications that run on multiple hosts:

```bash
bsub [lsf_options] pam -mpi mpirun [mpirun_options] -f appfile [-- extra_args_for_appfile]
```

In this case, 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 $MPI_ROOT/bin/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 the following items:

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

Host assignments are returned for the two symbolic links voyager and enterprise.

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

where [mpirun_options] for all of the preceding examples are:

- `-lsided`

  Enables one-sided communication.

- `-ck`

  Behaves like the `-p` option, but supports two additional checks of your MPI application; it checks if the specified host machines and programs are available, and also checks for access or permission problems.
Routes all off-host communication through daemons rather than between processes. Refer to “Communicating using daemons” on page 68 for more information.

-d

 Turns on debug mode.

-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.

-f appfile

 Specifies the appfile that mpirun parses to get program and process count information for the run. Refer to “Creating an appfile” on page 59 for details about setting up your appfile.

-h host

 Specifies a host on which to start the processes (default is local_host).

-ha

 Eliminates a teardown when ranks exit abnormally. Further communications involved with ranks that went away return error class MPI_ERR_EXITED, but do not force the application to teardown, as long as the MPI_Errhandler is set to MPI_ERRORS_RETURN. Some restrictions apply:

- Cannot be used with HyperFabric
- Communication is done via TCP/IP (Does not use shared memory for intranode communication.)
- Cannot be used with the diagnostic library.
- No instrumentation

-help

 Prints usage information for the utility.
Forces HMP to be used. Will cause the application to abort if HMP is unavailable.

-`i spec` Enables runtime instrumentation profiling for all processes. `spec` specifies options used when profiling. The options are the same as those for the environment variable `MPI_INSTR`. For example, the following is a valid command line:

```bash
% $MPI_ROOT/bin/mpirun -i mytrace:l:nc -f appfile
```

Refer to “MPI_INSTR” on page 46 for an explanation of `-i` options.

-`j` Prints the HP MPI job ID.

-`l user` Specifies the username on the target host (default is local username).

-`np #` Specifies the number of processes to run.

-`p` Turns on pretend mode. That is, the system goes through the motions of starting an HP MPI application but does not create processes. This is useful for debugging and checking whether the appfile is set up correctly.

-`prot` Prints the communication protocol between each host (i.e. TCP/IP, HyperFabric, or shared memory).

-`prun` Enables start-up with Elan usage. Only supported when linking with shared libraries. Some features like `mpirun -stdio processing` are unavailable. The `-np option is not allowed with `-prun`. When `mpirun [mpirun_options] -prun [prun_options]` is used,
the following options are allowed:

```
```

**-sp paths**

Sets the target shell PATH environment variable to paths. Search paths are separated by a colon.

**-spawn**

Enables dynamic processes.

**-t spec**

Enables runtime trace generation for all processes. spec specifies options used when tracing. For example, the following is a valid command line:

```
% $MPI_ROOT/bin/mpirun -t mytrace:off:nc -f appfile.
```

**-T**

Prints user and system times for each MPI rank.

**-tv**

Specifies that the application runs with the TotalView debugger. This option is not supported when you run mpirun under LSF.

**-v**

Turns on verbose mode.

**-version**

Prints the major and minor version numbers.

**args**

Specifies command-line arguments to the program—A space separated list of arguments.

**-- extra_args_for_appfile**

Specifies extra arguments to be applied to the programs listed in the appfile—A space separated list of arguments. Use this option at the end of your
command line to append extra arguments to each line of your appfile. Refer to the example in “Adding program arguments to your appfile” on page 59 for details.

**program**

Specifies the name of the executable file to run.

**IMPI_options**

Specifies this `mpirun` is an IMPI client. Refer to “IMPI” on page 70 for more information on IMPI, as well as a complete list of IMPI options.

**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.

**-stdio=[options]**

Specifies standard IO options. Refer to “External input and output” on page 110 for more information on standard IO, as well as a complete list of stdio options.

---

**CAUTION**

The `-help`, `-version`, `-p`, and `-tv` options are not supported with the `bsub pam -mpi mpirun` startup method.

---

**Appfiles**

An appfile is a text file that contains process counts and a list of programs. When you invoke `mpirun` with the name of the appfile, `mpirun` parses the appfile to get information for the run. You can use an appfile when you run a single executable file on a single host, and you must use an appfile when you run on multiple hosts or run multiple executable files.
Creating an appfile  The format of entries in an appfile is line oriented. Lines that end with the backslash (\) 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.

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 file is stored. The default is to search the 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. The default is not to set environment variables. When you use -e with the -h option, the environment variable is set to val on the remote host.

- **-l user** Specifies the user name on the target host. The default is the current user name.

- **-sp paths** Sets the target shell PATH environment variable to paths. Search paths are separated by a colon. Both -sp path and -e PATH=path do the same thing. If both are specified, the -e PATH=path setting is used.

- **-np #** Specifies the number of processes to run. The default value for # is 1.

- **program** Specifies the name of the executable to run. mpirun searches for the executable in the paths defined in the PATH environment variable.

- **args** Specifies command line arguments to the program. Options following a program name in your appfile are treated as program arguments and are not processed by mpirun.

Adding program arguments to your appfile  When you invoke mpirun using an appfile, arguments for your program are supplied on each line of your appfile—Refer to “Creating an appfile” on page 59. HP MPI also provides an option on your mpirun command line to provide
additional program arguments to those in your appfile. This is useful if you wish to specify extra arguments for each program listed in your appfile, but do not wish to edit your appfile.

To use an appfile when you invoke mpirun, use one of the following as described in “mpirun (mpirun.all)” on page 51:

mpirun [mpirun_options] -f appfile [-- extra_args_for_appfile]

- bsub [lsf_options] pam -mpi mpirun [mpirun_options] -f appfile [-- extra_args_for_appfile]

The -- extra_args_for_appfile option is placed at the end of your command line, after appfile, to add options to each line of your appfile.

CAUTION

Arguments placed after -- are treated as program arguments, and are not processed by mpirun. Use this option when you want to specify program arguments for each line of the appfile, but want to avoid editing the appfile.

For example, suppose your appfile contains

-h voyager -np 10 send_receive arg1 arg2
-h enterprise -np 8 compute_pi

If you invoke mpirun using the following command line:

mpirun -f appfile -- arg3 -arg4 arg5

- The send_receive command line for machine voyager becomes:
  send_receive arg1 arg2 arg3 -arg4 arg5
- The compute_pi command line for machine enterprise becomes:
  compute_pi arg3 -arg4 arg5

When you use the -- extra_args_for_appfile option, it must be specified at the end of the mpirun command line.

Setting remote environment variables To set environment variables on remote hosts use the -e option in the appfile. For example, to set the variable MPI_FLAGS:

-h remote_host -e MPI_FLAGS=val [-np #] program [args]
Environment variables can also be set globally on the `mpirun` command line:

```
% $MPI_ROOT/bin/mpirun -e MPI_FLAGS=y -f appfile
```

In the above example, if some `MPI_FLAGS` setting was specified in the appfile, then the global setting on the command line would override the setting in the appfile. To add to an environment variable rather than replacing it, use the following command:

```
% $MPI_ROOT/bin/mpirun -e MPI_FLAGS=%MPI_FLAGS,y -f appfile
```

In the above example, if the appfile specified `MPI_FLAGS=z`, then the resulting `MPI_FLAGS` seen by the application would be `z`, `y`.

**Assigning ranks and improving communication**  The ranks of the processes in `MPI_COMM_WORLD` are assigned and sequentially ordered according to the order the programs appear in the appfile.

For example, if your appfile contains

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

HP MPI assigns ranks 0 through 9 to the 10 processes running `send_receive` and ranks 10 through 17 to the 8 processes running `compute_pi`.

You can use this sequential ordering of process ranks to your advantage when you optimize for performance on multihost systems. You can split process groups according to communication patterns to reduce or remove interhost communication hot spots.

For example, if you have the following:

- A multi-host run of four processes
- Two processes per host on two hosts
- Communication between ranks 0—2 and 1—3 is slow.

You could use an appfile that contains the following:

```
-h hosta -np 2 program1
-h hostb -np 2 program2
```

However, this places processes 0 and 1 on hosta and processes 2 and 3 on hostb, resulting in interhost communication between the ranks identified as having slow communication:
A more optimal appfile for this example would be

- `h hosta -np 1 program1`
- `h hostb -np 1 program2`
- `h hosta -np 1 program1`
- `h hostb -np 1 program2`

This places ranks 0 and 2 on hosta and ranks 1 and 3 on hostb. This placement allows intrahost communication between ranks that are identified as communication hot spots. Intrahost communication yields better performance than interhost communication.

**Multipurpose daemon process**

HP MPI incorporates a multipurpose daemon process that provides start-up, communication, and termination services. The daemon operation is transparent. HP MPI sets up one daemon per host (or appfile entry) for communication. Refer to “Communicating using daemons” on page 68 for daemon details.
NOTE

Because HP MPI sets up one daemon per host (or appfile entry) for communication, when you invoke your application with -np x, HP MPI generates x+1 processes.

Generating multihost instrumentation profiles

To generate tracing output files for multihost applications, you must invoke mpirun on a host where at least one MPI process is running. HP MPI writes the trace file (prefix.tr) to the working directory on the host where mpirun runs.

When you enable instrumentation for multihost runs, and invoke mpirun either on a host where at least one MPI process is running, or on a host remote from all your MPI processes, HP MPI writes the instrumentation output file (prefix.instr) to the working directory on the host that is running rank 0.

prun

It is possible to start applications using the Elan on Linux and Tru64UNIX systems without mpirun. The following is an example using prun without mpirun:

% prun [options] application

This method has restrictions. It does not support MPI-2 dynamic processes or one-sided communication. We recommend certain environment variables be set before using this method. They are:

- For Linux:
  
  LD_LIBRARY_PATH=$MPI_ROOT/lib/linux_[ia32|ia64]

  Shared libraries will be linked by default. prun will not execute if this is not set.

- For Tru64UNIX:
  
  LD_LIBRARY_PATH=$MPI_ROOT/lib/alpha

  Shared libraries will be linked by default. prun will not execute if this is not set.

- For both Linux and Tru64UNIX:
**LIBELAN_SHM_ENABLE=0**

This tells the Elan system not to allocate its own shared memory. Since we allocate our own shared memory, the Elan shared memory would be ignored.

**NOTE**

Some versions of Quadrics have a bug that causes multithreaded applications to hang. Do not set `LIBELAN_SHM_ENABLE` if you are running multithreaded applications.

**mpiexec**

The MPI-2 standard defines `mpiexec` as a simple method to start MPI applications. It supports less features than `mpirun`, but it is portable.

`mpiexec` syntax has three formats:

- `mpiexec` offers arguments similar to a `MPI_Spawn` call, with arguments as shown in the following form:

  `mpiexec [-n maxprocs][-soft ranges][-host host][-arch arch][-wdir dir][-path dirs][-file file]command-args`

  For example:

  ```
  $MPI_ROOT/bin/mpiexec -n 8 ./myprog.x 1 2 3
  ```

  creates an 8 rank MPI job on the local host consisting of 8 copies of the program `myprog.x`, each with the command line arguments 1, 2, and 3.

- It also allows arguments like a `MPI_Spawn_multiple` call, with a colon separated list of arguments, where each component is like the form above.

  `mpiexec above : above : ... : above`

  For example:

  ```
  $MPI_ROOT/bin/mpiexec -n 4 ./myprog.x : -host host2 -n 4 /path/to/myprog.x
  ```

  creates a MPI job with 4 ranks on the local host and 4 on `host2`.

- Finally, the third form allows the user to specify a file containing lines of data like the arguments in the first form.
mpiexec [-configfile file]

For example:

% $MPI_ROOT/bin/mpiexec -configfile cfile

gives the same results as in the second example, but using the
-configfile option (assuming the file cfile contains 
-host host2 -n 4 ./myprog.x
-wdir /some/path ./myprog.x)

where [mpiexec_options] are:

-n maxprocs Create maxprocs MPI ranks on the specified host.
-soft range-list Ignored in HP MPI.
-host host Specifies the host on which to start the ranks.
-arch arch Ignored in HP MPI.
-wdir dir Working directory for the created ranks.
-path dirs PATH environment variable for the created ranks.
-file file Ignored in HP MPI.

This last option is used separately from the options above.

-configfile file Specify a file of lines containing the above options.

mpiexec does not support prun startup.

mpijob

mpijob lists the HP MPI jobs running on the system. Invoke mpijob on
the same host as you initiated mpirun. mpijob syntax is shown below:

mpijob [-help] [-a] [-u] [-j id] [id 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. You can list a
 number of job IDs in a space-separated list.

When you invoke mpijob, it 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.

An `mpijob` output using the `-a` and `-u` options is shown below listing jobs for all users and sorting them by user name.

<table>
<thead>
<tr>
<th>JOB</th>
<th>USER</th>
<th>NPROCS</th>
<th>PROGNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>22623</td>
<td>charlie</td>
<td>12</td>
<td>/home/watts</td>
</tr>
<tr>
<td>22573</td>
<td>keith</td>
<td>14</td>
<td>/home/richards</td>
</tr>
<tr>
<td>22617</td>
<td>mick</td>
<td>100</td>
<td>/home/jagger</td>
</tr>
<tr>
<td>22677</td>
<td>ron</td>
<td>4</td>
<td>/home/wood</td>
</tr>
</tbody>
</table>

When you specify the `-j` option, `mpijob` reports the following 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 Indicates whether the process is running (an x is used) or has been terminated.

PROGNAME Program names used in the HP MPI application.

`mpijob` does not support `prun` startup.

mpiclean

*mpiclean* kills processes in an HP MPI application. Invoke *mpiclean* on the host on which you initiated `mpirun`.

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.

*mpiclean* syntax has two forms:

1. `mpiclean [-help] [-v] -j id [id id ....]`
2. `mpiclean [-help] [-v] -m`

where
Understanding HP MPI

Running applications

-**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 in a space-separated list. Obtain the job ID using the -j option when you invoke mpirun.

The first syntax is used for all servers and is the preferred method to kill an MPI application. You can only kill jobs that are your own.

The second syntax is used when an application aborts during MPI_Init, and the termination of processes does not destroy the allocated shared-memory segments.

mpiclean does not support prun startup.

**HyperFabric/HyperMessaging Protocol (HMP)**

HyperMessaging Protocol (HMP) is a messaging-based protocol that significantly enhances performance of parallel and technical applications by optimizing the processing of various communication tasks across interconnected hosts. It provides low latency, high bandwidth, and low CPU overhead networking. HMP is part of the HyperFabric driver. HMP uses HyperFabric switches and HyperFabric network interface cards. The HMP protocol can coexist with the TCP/IP protocol over HyperFabric.

The HMP functionality shipped with HP MPI 2.0 is turned off by default. (**MPI_HMP=off**)  
There are four possible values for **MPI_HMP**: **on**, **off**, **ON**, and **OFF**.

The file /etc/mpi.conf can be created and set to define the system-wide default for HMP functionality. Setting **MPI_HMP** within the file to **on** or **off** is advisory only, and can be overridden by the user with the use of the environment variable. Setting **MPI_HMP** within the file to **ON** or **OFF** is forced and will override the user environment variable. An example of the mpi.conf file is shipped with the product and is located at **opt/mpi/etc**.
The environment variable `MPI_HMP` can be set to `on`, `off`, `ON`, or `OFF` by the user on a per-job basis. The user can override system defaults of `on` or `off` (advisory), but not system defaults of `ON` or `OFF` (forced). Some combinations of settings (in the file and variable) are illegal and will generate errors.

**NOTE**

All HMP enabled nodes must be on the same HyperFabric network in order to allow this functionality.

The preferred method for enabling HMP is use of the `mpirun` option `-hmp` which will enable HMP on every host.

If you developed your applications on a system without HMP installed, the resulting executables cannot use HMP. When HMP is installed, you will have to link or relink your applications to enable HMP support. We recommend building your applications using our scripts to ensure your executable is built with support for HMP.

Existing compilation scripts that do not use our wrappers will have to relink using the `-show` option.

If you develop on a system without HyperFabric hardware, you can still swinstall HyperFabric software to allow creation of HMP applications.


**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, n² sockets are required to allow processes on one host to communicate with processes on other hosts. When you use this 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, by specifying the `--commd` option to the `mpirun` command. 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.

Figure 3-1 shows the structure for daemon communication.

![Diagram of daemon communication](image.png)

To use daemon communication, specify the `--commd` option in the `mpirun` command. 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. Refer to “`mpirun (mpirun.all)`” on page 51 and “`MPI_COMMD`” on page 39 for more information.

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

HP MPI sets up one daemon per host (or appfile entry) for communication. If you invoke your application with `-np x`, HP MPI generates x+1 processes.

IMPI

The Interoperable MPI protocol (IMPI) extends the power of MPI by allowing applications to run on heterogeneous clusters of machines with various architectures and operating systems, while allowing the program to use a different implementation of MPI on each machine.

This is accomplished without requiring any modifications to the existing MPI specification. That is, IMPI does not add, remove, or modify the semantics of any of the existing MPI routines. All current valid MPI programs can be run in this way without any changes to their source code.

In IMPI, all messages going out of a host go through the daemon. The messages between daemons have the fixed message format. The protocols in different IMPI implementations are the same.

Currently, IMPI is not supported in multi-threaded library. If the user application is a multi-threaded program, it is not allowed to start as an IMPI job.

An IMPI server is available for download from Notre Dame at: http://www.lsc.nd.edu/research/impi

The IMPI syntax is:

```bash
mpirun [-client # ip:port]
```

where

- `-client` Specifies this `mpirun` is an IMPI client.
- `#` Specifies the client number. The first # is 0.
- `ip` Specifies the IP address of the IMPI server.
- `port` Specifies the port number of the IMPI server.
Native language support

By default, diagnostic messages and other feedback from HP MPI are provided in English. Support for other languages is available through the use of the Native Language Support (NLS) catalog and the internationalization environment variable NLSPATH.

The default NLS search path for HP MPI is $NLSPATH. Refer to the environ(5) man page for NLSPATH usage.

When an MPI language catalog is available, it represents HP MPI messages in two languages. The messages are paired so that the first in the pair is always the English version of a message and the second in the pair is the corresponding translation to the language of choice.

Refer to the hpnls (5), environ (5), and lang (5) man pages for more information about Native Language Support.
This chapter provides information about utilities you can use to analyze HP MPI applications. The topics covered are:

- Using counter instrumentation
— Creating an instrumentation profile
— Viewing ASCII instrumentation data
• Using the profiling interface
Using counter instrumentation

Counter instrumentation is a lightweight method for generating cumulative runtime statistics for your MPI applications. When you create an instrumentation profile, HP MPI creates an ASCII format.

You can create instrumentation profiles for applications linked with the standard HP MPI library, and for applications linked with HP MPI version 2.0, you can also create profiles for applications linked with the thread-compliant library. Instrumentation is not supported for applications linked with the diagnostic library (-ldmpi).

Creating an instrumentation profile

Create an instrumentation profile using one of the following methods:

- Use the following syntax:

  mpirun -i spec -np # program

  Refer to “Compiling and running your first application” on page 21 and “mpirun (mpirun.all)” on page 51 for more details about implementation and syntax.

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

  % $MPI_ROOT/bin/mpirun -i compute_pi -np 2 compute_pi

  This invocation creates an instrumentation profile in the following format: compute_pi.instr (ASCII).

- Specify a filename prefix using the MPI_INSTR environment variable. Refer to “MPI_INSTR” on page 46 for syntax information.

  For example,

  % setenv MPI_INSTR compute_pi

  Specifies the instrumentation output file prefix as compute_pi.

  Specifications you make using mpirun -i override any specifications you make using the MPI_INSTR environment variable.
Profiling

Using counter instrumentation

MPIHP_Trace_on and MPIHP_Trace_off

By default, the entire application is profiled from MPI_Init to MPI_Finalize. However, HP MPI provides the nonstandard MPIHP_Trace_on and MPIHP_Trace_off routines to collect profile information for selected code sections only.

To use this functionality:

1. Insert the MPIHP_Trace_on and MPIHP_Trace_off pair around code that you want to profile.

2. Build the application and invoke mpirun with the -i off option. -i off specifies that counter instrumentation is enabled but initially turned off (refer to “mpirun (mpirun.all)” on page 51 and “MPI_INSTR” on page 46). Data collection begins after all processes collectively call MPIHP_Trace_on. HP MPI collects profiling information only for code between MPIHP_Trace_on and MPIHP_Trace_off.

---

CAUTION

MPIHP_Trace_on and MPIHP_Trace_off are collective routines and must be called by all ranks in your application. Otherwise, the application deadlocks.

---

Viewing ASCII instrumentation data

The ASCII instrumentation profile is a text file with the .instr extension. For example, to view the instrumentation file for the compute_pi.f application, you can print the prefix.instr file. If you defined prefix for the file as compute_pi, as you did when you created the instrumentation file in “Creating an instrumentation profile” on page 75, you would print compute_pi.instr.

The ASCII instrumentation profile provides the version, the date your application ran, and summarizes information according to application, rank, and routines. Figure 4-1 on page 77 is an example of an ASCII instrumentation profile.

The information available in the prefix.instr file includes:

- Overhead time—The time a process or routine spends inside MPI. For example, the time a process spends doing message packing.
- Blocking time—The time a process or routine is blocked waiting for a message to arrive before resuming execution.

**NOTE**  
If spin-yield time is changed, overhead and blocking times become less accurate.

- Communication hot spots—The processes in your application between which the largest amount of time is spent in communication.
- Message bin—The range of message sizes in bytes. The instrumentation profile reports the number of messages according to message length.

**NOTE**  
You do not get message size information for MPI_Alltoallv instrumentation.

Figure 4-1 displays the contents of the example report compute_pi.instr.

**Figure 4-1**  
**ASCII instrumentation profile**

Version: HP MPI 01.08.00.00 B6060BA - HP-UX 11.0  
Date: Mon Apr 01 15:59:10 2002  
Processes: 2  
User time: 6.57%  
MPI time: 93.43% [Overhead:93.43% Blocking:0.00%]
### Profiling

**Using counter instrumentation**

#### Application Summary by Rank (second):

<table>
<thead>
<tr>
<th>Rank</th>
<th>Proc CPU Time</th>
<th>User Portion</th>
<th>System Portion</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.040000</td>
<td>0.010000( 25.00%)</td>
<td>0.030000( 75.00%)</td>
</tr>
<tr>
<td>1</td>
<td>0.030000</td>
<td>0.010000( 33.33%)</td>
<td>0.020000( 66.67%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rank</th>
<th>Proc Wall Time</th>
<th>User</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.126335</td>
<td>0.008332( 6.60%)</td>
<td>0.118003( 93.40%)</td>
</tr>
<tr>
<td>1</td>
<td>0.126355</td>
<td>0.008260( 6.54%)</td>
<td>0.118095( 93.46%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rank</th>
<th>Proc MPI Time</th>
<th>Overhead</th>
<th>Blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.118003</td>
<td>0.118003(100.00%)</td>
<td>0.000000( 0.00%)</td>
</tr>
<tr>
<td>1</td>
<td>0.118095</td>
<td>0.118095(100.00%)</td>
<td>0.000000( 0.00%)</td>
</tr>
</tbody>
</table>

#### Routine Summary by Rank:

<table>
<thead>
<tr>
<th>Rank</th>
<th>Routine Statistic</th>
<th>Calls</th>
<th>Overhead(ms)</th>
<th>Blocking(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MPI_Bcast</td>
<td>1</td>
<td>5.397081</td>
<td>0.000000</td>
</tr>
<tr>
<td></td>
<td>MPI_Finalize</td>
<td>1</td>
<td>1.238942</td>
<td>0.000000</td>
</tr>
<tr>
<td></td>
<td>MPI_Init</td>
<td>1</td>
<td>107.195973</td>
<td>0.000000</td>
</tr>
<tr>
<td></td>
<td>MPI_Reduce</td>
<td>1</td>
<td>4.171014</td>
<td>0.000000</td>
</tr>
</tbody>
</table>
Profiling
Using counter instrumentation

<table>
<thead>
<tr>
<th>Function</th>
<th>Rank</th>
<th>Time (s)</th>
<th>CRC (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Bcast</td>
<td>1</td>
<td>5.388021</td>
<td>0.000000</td>
</tr>
<tr>
<td>MPI_Finalize</td>
<td>1</td>
<td>1.325965</td>
<td>0.000000</td>
</tr>
<tr>
<td>MPI_Init</td>
<td>1</td>
<td>107.228994</td>
<td>0.000000</td>
</tr>
<tr>
<td>MPI_Reduce</td>
<td>1</td>
<td>4.152060</td>
<td>0.000000</td>
</tr>
</tbody>
</table>

Message Summary by Rank Pair:

<table>
<thead>
<tr>
<th>SRank</th>
<th>DRank</th>
<th>Messages (minsize,maxsize)/[bin]</th>
<th>Totalbytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>(4, 4)</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>[0..64]</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SRank</th>
<th>DRank</th>
<th>Messages (minsize,maxsize)/[bin]</th>
<th>Totalbytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>(8, 8)</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>[0..64]</td>
<td>8</td>
</tr>
</tbody>
</table>
Using the profiling interface

The MPI profiling interface provides a mechanism by which implementors of profiling tools can collect performance information without access to the underlying MPI implementation source code.

Because HP MPI provides several options for profiling your applications, you may not need the profiling interface to write your own routines. HP MPI makes use of MPI profiling interface mechanisms to provide the diagnostic library for debugging. In addition, HP MPI provides tracing and lightweight counter instrumentation. For details, refer to

- “Using counter instrumentation” on page 75
- “Using the diagnostics library” on page 102

The profiling interface allows you to intercept calls made by the user program to the MPI library. For example, you may want to measure the time spent in each call to a certain library routine or create a log file. You can collect your information of interest and then call the underlying MPI implementation through a name shifted entry point.

All routines in the HP MPI library begin with the MPI_ prefix. Consistent with the “Profiling Interface” section of the MPI 1.2 standard, routines are also accessible using the PMPI_ prefix (for example, MPI_Send and PMPI_Send access the same routine).

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 the MPI library routine using its PMPI_ prefix.

Fortran profiling interface

To facilitate improved Fortran performance, we no longer implement Fortran calls as wrappers to C calls. Consequently, profiling routines built for C calls will no longer cause the corresponding Fortran calls to be wrapped automatically. In order to profile Fortran routines, separate wrappers need to be written for the Fortran calls.

For example:
#include <stdio.h>
#include <mpi.h>

int MPI_Send(void *buf, int count, MPI_Datatype type,
             int to, int tag, MPI_Comm comm)
{
    printf("Calling C MPI_Send to %d\n", to);
    return PMPI_Send(buf, count, type, to, tag, comm);
}

#pragma _HP_SECONDARY_DEF mpi_send mpi_send_
void mpi_send(void *buf, int *count, int *type, int *to,
              int *tag, int *comm, int *ierr)
{
    printf("Calling Fortran MPI_Send to %d\n", *to);
    pmpi_send(buf, count, type, to, tag, comm, ierr);
}
Profiling

Using the profiling interface
This chapter provides information about tuning HP MPI applications to improve performance. The topics covered are:

- MPI_FLAGS options
Tuning

- Message latency and bandwidth
- Multiple network interfaces
- Processor subscription
- MPI routine selection
- Multilevel parallelism
- Coding considerations

The tuning information in this chapter improves application performance in most but not all cases. Use this information together with the output from counter instrumentation to determine which tuning changes are appropriate to improve your application's performance.

When you develop HP MPI applications, several factors can affect performance, whether your application runs on a single computer or in an environment consisting of multiple computers in a network. These factors are outlined in this chapter.
MPI_FLAGS options

The function parameter error checking is turned off by default. It can be turned on by setting MPI_FLAGS=Eon.

If you are running an application stand-alone on a dedicated system, setting MPI_FLAGS=y allows MPI to busy spin, thereby improving latency. See “MPI_FLAGS” on page 41 for more information on the y option.
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.
- 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:

```c
j = 0
for (i=0; i<size; i++) {
    if (i!=rank) continue;
    MPI_Irecv(buf[i], count, dtype, i, 0, comm, &requests[j++]);
```
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:

```c
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.

For example, when you have two hosts, host0 and host1, each communicating using two ethernet cards, ethernet0 and ethernet1, you have four host names as follows:

- host0-ethernet0
- host0-ethernet1
- host1-ethernet0
- host1-ethernet1

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

```
-h host0-ethernet0 -np 16 beavis.exe
-h host0-ethernet1 -np 16 beavis.exe
-h host1-ethernet0 -np 16 beavis.exe
-h host1-ethernet1 -np 16 beavis.exe
```
Now, when the appfile is run, 32 processes run on host0 and 32 processes run on host1 as shown in Figure 5-1.

**Figure 5-1** 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-ethernet0 and the host0-ethernet1 network interfaces with host1 processes.
Processor subscription

Subscription refers to the match of processors and active processes on a host. Table 5-1 lists possible subscription types.

<table>
<thead>
<tr>
<th>Subscription type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Under subscribed</td>
<td>More processors than active processes</td>
</tr>
<tr>
<td>Fully subscribed</td>
<td>Equal number of processors and active processes</td>
</tr>
<tr>
<td>Over subscribed</td>
<td>More active processes than processors</td>
</tr>
</tbody>
</table>

When a host 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.

In a situation where your system is oversubscribed but your MPI application is not, you can use gang scheduling to improve performance. Refer to “MP_GANG” on page 45 for details. This is only available on HP-UX systems.
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.
Multilevel parallelism

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

- Use the MPI library to provide coarse-grained parallelism and a parallelizing compiler to provide fine-grained (that is, thread-based) parallelism. An appropriate mix of coarse- and fine-grained parallelism provides better overall performance.

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

The following are suggestions and items to consider when coding your MPI applications to improve performance:

- Use HP MPI collective routines instead of coding your own with point-to-point routines because HP MPI's collective routines are optimized to use shared memory where possible for performance.

  Use commutative MPI reduction operations.
  
  — Use the MPI predefined reduction operations whenever possible because they are optimized.
  
  — When defining your own reduction operations, make them commutative. Commutative operations give MPI more options when ordering operations allowing it to select an order that leads to best performance.

- Use MPI derived datatypes when you exchange several small size messages that have no dependencies.

- Minimize your use of MPI_Test() polling schemes to minimize polling overhead.

- Code your applications to avoid unnecessary synchronization. In particular, strive to avoid MPI_Barrier calls. Typically an application can be modified to achieve the same end result using targeted synchronization instead of collective calls. For example, in many cases a token-passing ring may be used to achieve the same coordination as a loop of barrier calls.
6 Debugging and troubleshooting

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

- Using Visual MPI
Debugging and troubleshooting

- Debugging HP MPI applications
  - Using a single-process debugger
  - Using a multi-process debugger
  - Using the diagnostics library
  - Enhanced debugging output
  - Backtrace functionality
- Troubleshooting HP MPI applications
  - Building
  - Starting
  - Running
  - Completing
- Frequently asked questions
Debugging HP MPI applications

HP MPI allows you to use single-process debuggers to debug applications. The available debuggers are ADB, DDE, XDB, WDB and GDB. You access these debuggers by setting options in the `MPI_FLAGS` environment variable. HP MPI also supports the multithread, multiprocess debugger, TotalView on HP-UX 11i and higher.

In addition to the use of debuggers, HP MPI provides a diagnostic library (DLIB) for advanced error checking and debugging. HP MPI also provides options to the environment variable `MPI_FLAGS` that report memory leaks (`l`), force MPI errors to be fatal (`f`), print the MPI job ID (`j`), and other functionality.

This section discusses single- and multi-process debuggers and the diagnostic library; refer to “MPI_FLAGS” on page 41 for information about using the `MPI_FLAGS` option.

Using Visual MPI

Visual MPI is an MPI analysis tool focused on error detection and visualization, with automatic correlation to application source code. While Visual MPI includes a range of features, there are several highlights: ease of use (near-zero initial learning curve), automated analysis capabilities, and reporting of a range of programming errors. For more information about Visual MPI, refer to the documents available at http://www.hp.com/go/mpi and in the Visual MPI online help.

NOTE

Visual MPI usage requires that your application is linked with the MPI shared libraries, and be started with the `mpirun` command.

Visual MPI is supported on HP-UX 11i Versions 1.6 and 2.0 (Itanium-based platforms only), Linux Intel IA-32 and Itanium, and Tru64UNIX 5.1 or higher.

Visual MPI for Tru64UNIX requires the subset containing the Base System, as well as the Software Development subset OSFPGMR.
Using a single-process debugger

Because HP MPI creates multiple processes and ADB, DDE, XDB, WDB, GDB, and LADEBUG only handle single processes, HP MPI starts one debugger session per process. HP MPI creates processes in MPI_Init, and each process instantiates a debugger session. Each debugger session in turn attaches to the process that created it. HP MPI provides MPI_DEBUG_CONT to avoid a possible race condition while the debugger session starts and attaches to a process. MPI_DEBUG_CONT is an environment variable that HP MPI uses to temporarily halt debugger progress beyond MPI_Init. By default, MPI_DEBUG_CONT is set to 0 and you must reset it to 1 to allow the debug session to continue past MPI_Init.

The following procedure outlines the steps to follow when you use a single-process debugger:

**Step 1.** Set the `eadb`, `exdb`, `edde`, `ewdb`, `egdb`, or `eladebug` option in the `MPI_FLAGS` environment variable to use the ADB, XDB, DDE, WDB, GDB, or LADEBUG debugger respectively. Refer to “MPI_FLAGS” on page 41 for information about `MPI_FLAGS` options.

**Step 2.** 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 3.** Run your application.

When your application enters MPI_Init, HP MPI starts one debugger session per process and each debugger session attaches to its process.

**Step 4.** Set a breakpoint anywhere following MPI_Init in each session.

**Step 5.** Set the global variable `MPI_DEBUG_CONT` to 1 using each session’s command line interface or graphical user interface. The syntax for setting the global variable depends upon which debugger you use:

- (adb) `mpi_debug_cont/w 1`
- (dde) `set mpi_debug_cont = 1`
- (xdb) `print *MPI_DEBUG_CONT = 1`
- (wdb) `set MPI_DEBUG_CONT = 1`
- (gdb) `set MPI_DEBUG_CONT = 1`
- (ladebug) `set MPI_DEBUG_CONT = 1`
NOTE

For the ladebug debugger, /usr/bin/X11 may need to be added to the command search path.

Step 6. Issue the appropriate debugger command in each session to continue program execution.

Each process runs and stops at the breakpoint you set after MPI_Init.

Step 7. Continue to debug each process using the appropriate commands for your debugger.

CAUTION

To improve performance, HP MPI supports a process-to-process, one-copy messaging approach. This means that one process can directly copy a message into the address space of another process. Because of this process-to-process bcopy (p2p_bcopy) implementation, a kernel thread is created for each process that has p2p_bcopy enabled. This thread deals with page and protection faults associated with the one-copy operation.

This extra kernel thread can cause anomalous behavior when you use DDE on HP-UX 11i and higher. If you experience such difficulty, you can disable p2p_bcopy by setting the MPI_2BCOPY environment variable to 1.

Using a multi-process debugger

hp MPI supports the TotalView debugger on HP-UX version 11i and higher. The preferred method when you run TotalView with HP MPI applications is to use the mpirun runtime utility command.

For example,

% $MPI_ROOT/bin/mpicc myprogram.c -g
% $MPI_ROOT/bin/mpirun -tv -np 2 a.out

In this example, myprogram.c is compiled using the HP MPI compiler utility for C programs (refer to “Compiling and running your first application” on page 21). The executable file is compiled with source line information and then mpirun runs the a.out MPI program:
Debugging and troubleshooting

Debugging HP MPI applications

amd64 | x86-64 | ia64

- 

- Specifies that the compiler generate the additional information needed by the symbolic debugger.

- np 2

- Specifies the number of processes to run (2, in this case).

- tv

- Specifies that the MPI ranks are run under TotalView.

Alternatively, use mpirun to invoke an appfile:

% $MPI_ROOT/bin/mpirun -tv -f my_appfile

- tv

- Specifies that the MPI ranks are run under TotalView.

- f appfile

- Specifies that mpirun parses my_appfile to get program and process count information for the run. Refer to “Creating an appfile” on page 59 for details about setting up your appfile.

Refer to “mpirun (mpirun.all)” on page 51 for details about mpirun.

Refer to the “MPI_FLAGS” on page 41 and the TotalView documentation for details about MPI_FLAGS and TotalView command line options, respectively.

By default, mpirun searches for TotalView in your PATH settings. You can also define the absolute path to TotalView using the TOTALVIEW environment variable:

% setenv TOTALVIEW /opt/totalview/bin/totalview [totalview-options]

The TOTALVIEW environment variable is used by mpirun.

NOTE

When attaching to a running MPI application, you should attach to the MPI daemon process to enable debugging of all the MPI ranks in the application. You can identify the daemon process as the one at the top of a hierarchy of MPI jobs (the daemon also usually has the lowest PID among the MPI jobs).

Limitations

The following limitations apply to using TotalView with HP MPI applications:
1. All the executable files in your multihost MPI application must reside on your local machine, that is, the machine on which you start TotalView. Refer to “TotalView multihost example” on page 101 for details about requirements for directory structure and file locations.

2. TotalView sometimes displays extra HP-UX threads that have no useful debugging information. These are kernel threads that are created to deal with page and protection faults associated with one-copy operations that HP MPI uses to improve performance. You can ignore these kernel threads during your debugging session.

To improve performance, HP MPI supports a process-to-process, one-copy messaging approach. This means that one process can directly copy a message into the address space of another process. Because of this process-to-process bcopy (p2p_bcopy) implementation, a kernel thread is created for each process that has p2p_bcopy enabled. This thread deals with page and protection faults associated with the one-copy operation.

**TotalView multihost example**

The following example demonstrates how to debug a typical HP MPI multihost application using TotalView, including requirements for directory structure and file locations.

The MPI application is represented by an appfile, named my_appfile, which contains the following two lines:

```
-h local_host -np 2 /path/to/program1
-h remote_host -np 2 /path/to/program2
```

my_appfile resides on the local machine (local_host) in the /work/mpiapps/total directory.

To debug this application using TotalView (in this example, TotalView is invoked from the local machine):

1. Place your binary files in accessible locations.
   - /path/to/program1 exists on local_host
   - /path/to/program2 exists on remote_host
To run the application under TotalView, the directory layout on your local machine, with regard to the MPI executable files, must mirror the directory layout on each remote machine. Therefore, in this case, your setup must meet the following additional requirement:

- `/path/to/program2` exists on `local_host`

2. In the `/work/mpiapps/total` directory on `local_host`, invoke TotalView by passing the `-tv` option to `mpirun`:

   ```bash
   % $MPI_ROOT/bin/mpirun -tv -f my_appfile
   ```

**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.

  ```c
  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, set the `MPI_DLIB_FLAGS` environment variable options appropriately. See “MPI_DLIB_FLAGS” on page 40 for more information.

To use the diagnostics library, specify the `-ldmpi` option when you compile your application.
NOTE

Using DLIB reduces application performance. DLIB is not thread-compliant. Also, you cannot use DLIB with instrumentation.

Enhanced debugging output

HP MPI 2.0 provides improved readability and usefulness of MPI processes stdout and stderr. More intuitive options have been added for handling standard input:

- Directed: Input is directed to a specific MPI process.
- Broadcast: Input is copied to the stdin of all processes.
- Ignore: Input is ignored.

The default behavior is standard input is ignored.

Additional options are available to avoid confusing interleaving of output:

- Line buffering, block buffering, or no buffering
- Prepending of processes ranks to their stdout and stderr
- Simplification of redundant output

Backtrace functionality

HP MPI 2.0 handles several common termination signals differently than earlier versions of HP MPI. If any of the following signals are generated by an MPI application, a stack trace is printed prior to termination:

- SIGBUS - bus error
- SIGSEGV - segmentation violation
- SIGILL - illegal instruction
- SIGSYS - illegal argument to system call

The backtrace is helpful in determining where the signal was generated and the call stack at the time of the error. If a signal handler is established by the user code before calling MPI_Init, no backtrace will be printed for that signal type and the user’s handler will be solely
responsible for handling the signal. Any signal handler installed after
MPI_Init will also override the backtrace functionality for that signal
after the point it is established. If multiple processes cause a signal, each
of them will print a backtrace.

In some cases, the prepending and buffering options available in HP MPI
2.0’s standard IO processing are useful in providing more readable
output.

The default behavior is to print a stack trace.

Backtracing can be turned off entirely by setting the environment
variable MPI_NOBACKTRACE. See “MPI_NOBACKTRACE” on page 49.

Backtracing is only supported on HP PA-RISC systems.
Troubleshooting HP MPI applications

This section describes limitations in HP MPI, some common difficulties you may face, and hints to help you overcome those difficulties and get the best performance from your HP MPI applications. Check this information first when you troubleshoot problems. The topics covered are organized by development task and also include answers to frequently asked questions:

- Building
- Starting
- Running
- Completing
- Frequently asked questions

To get information about the version of HP MPI installed on your system, use the `what` command. The following is an example of the command and its output:

```
% what $MPI_ROOT/bin/mpicc
$MPI_ROOT/bin/mpicc:
HP MPI 02.00.00.00 (dd/mm/yyyy) B6060BA - HP-UX 11.i
```

This command returns the HP MPI version number, the date this version was released, HP MPI product numbers, and the operating system version.

Building

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

If you use your own build script, specify all necessary input libraries. To determine what libraries are needed, check the contents of the compilation utilities stored in the HP MPI $MPI_ROOT/bin subdirectory.
HP MPI supports a 64-bit version of the MPI library on platforms running HP-UX 11i and higher. Both 32- and 64-bit versions of the library are shipped with HP-UX 11i and higher. For HP-UX 11i and higher, you cannot mix 32-bit and 64-bit executables in the same application.

HP MPI does not support Fortran applications that are compiled with the following option:

- `+autodblpad`— Fortran 77 programs

**Starting**

**CAUTION**
Starting a MPI executable without the `mpirun` utility is no longer supported. For example, applications previously started by using `a.out -np # [args]` must now be started using `mpirun -np # a.out [args]`.

When starting multihost applications, make sure that:

- All remote hosts are listed in your `.rhosts` file on each machine and you can `remsh` to the remote machines. The `mpirun` command has the `-ck` option you can use to determine whether the hosts and programs specified in your MPI application are available, and whether there are access or permission problems. Refer to “mpirun (mpirun.all)” on page 51.

- Application binaries are available on the necessary remote hosts and are executable on those machines

- The `-sp` option is passed to `mpirun` to set the target shell PATH environment variable. You can set this option in your `appfile`

- 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 and may include:

- Shared memory
- Message buffering
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.

Message buffering

According to the MPI standard, message buffering may or may not occur when processes communicate with each other using `MPI_Send`. `MPI_Send` buffering is at the discretion of the MPI implementation. 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. For
example, a sequence of operations (labeled "Deadlock") as illustrated in Table 6-1 would result in such a deadlock. Table 6-1 also illustrates the sequence of operations that would avoid code deadlock.

### Table 6-1 Non-buffered messages and deadlock

<table>
<thead>
<tr>
<th></th>
<th>Deadlock</th>
<th>No Deadlock</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Process 1</strong></td>
<td><strong>Process 2</strong></td>
<td><strong>Process 1</strong></td>
</tr>
<tr>
<td>MPI_Send(2,...)</td>
<td>MPI_Send(1,...)</td>
<td>MPI_Send(2,...)</td>
</tr>
<tr>
<td>MPI_Recv(2,...)</td>
<td>MPI_Recv(1,...)</td>
<td>MPI_Recv(2,...)</td>
</tr>
</tbody>
</table>

### 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:

```bash
-h remote_host -e var=val [-np #] program [args]
```

Refer to “Creating an appfile” on page 59 for details. Alternatively, you can set environment variables using the `.cshrc` file on each remote host if you are using a `/bin/csh`-based shell.

### Interoperability

Depending upon what server resources are available, applications may run on heterogeneous systems.

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 `K_server` and an V-Class server called `V_server`, you create the following appfile:

```bash
-h K_server -np 4 sum_accelerations
-h V_server -np 8 calculate_average
```
Then, you invoke `mpirun` passing it the name of the appfile you created. 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 because HP MPI assigns ranks in MPI_COMM_WORLD according to the order the programs appear in the appfile.

**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.
External input and output

You can use stdin, stdout, and stderr in your applications to read and write data. By default, HP MPI does not perform any processing on either stdin or stdout. The controlling tty determines stdio behavior in this case.

HP MPI does provide optional stdio processing features. stdin can be targeted to a particular process, or can be broadcast to every process. stdout processing includes buffer control, prepending MPI rank numbers, and combining repeated output.

HP MPI standard IO options can be set by using the following options to mpirun:

```
mpirun -stdio=bline[#] | bnone[#] | b[#], [p], [r[#]], [i[#]]
```

where

- **i** Broadcasts standard input to all MPI processes.
- **i [#]** Directs standard input to the process with global rank #.

The following modes are available for buffering:

- **b [#>0]** Specifies that the output of a single MPI process is placed to the standard out of mpirun after # bytes of output have been accumulated.
- **bnone [#>0]** The same as b[#] except that the buffer is flushed both when it is full and when it is found to contain any data. Essentially provides no buffering from the user’s perspective.
- **bline [#>0]** Displays the output of a process after a line feed is encountered, or the # byte buffer is full.

The default value of # in all cases is 10k bytes

The following option is available for prepending:

- **p** Enables prepending. The global rank of the originating process is prepended to stdout and stderr output. Although this mode can be combined with any buffering mode, prepending makes the most sense with the modes b and bline.
The following option is available for combining repeated output:

\[ r \ [\#>1] \]

Combines repeated identical output from the same process by prepending a multiplier to the beginning of the output. At most, \# maximum repeated outputs are accumulated without display. This option is used only with `bline`. The default value of \# is infinity.

### 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. `mpiclean` uses the job ID from `mpirun` `-j` to specify the application to terminate.
Frequently asked questions

This section describes frequently asked HP MPI questions. These questions address the following issues:

- Time in MPI_Finalize
- MPI clean up
- Application hangs in MPI_Send

Time in MPI_Finalize

**QUESTION:** When I build with HP MPI and then turn tracing on, the application takes a long time inside MPI_Finalize. What is causing this?

**ANSWER:** When you turn tracing on MPI_Finalize spends time consolidating the raw trace generated by each process into a single output file (with a .tr extension).

MPI clean up

**QUESTION:** 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 you use mpirun, it deletes the job file when the application terminates, whether successfully or not.
- When an application calls MPI_Abort, MPI_Abort deletes the job file.
- 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.

Application hangs in MPI_Send

**QUESTION:** My MPI application hangs at MPI_Send. Why?
ANSWER: Deadlock situations can occur when your code uses standard send operations and assumes buffering behavior for standard communication mode. You should not assume message buffering between processes because the MPI standard does not mandate a buffering strategy. HP MPI does sometimes use buffering for MPI_Send and MPI_Rsend, but it is dependent on message size and at the discretion of the implementation.

QUESTION: How can I tell if the deadlock is because my code depends on buffering?

ANSWER: To quickly determine whether the problem is due to your code being dependent on buffering, set the z option for MPI_FLAGS. MPI_FLAGS modifies the general behavior of HP MPI, and in this case converts MPI_Send and MPI_Rsend calls in your code to MPI_Ssend, without you having to rewrite your code. MPI_Ssend guarantees synchronous send semantics, that is, a send can be started whether or not a matching receive is posted. However, the send completes successfully only if a matching receive is posted and the receive operation has started to receive the message sent by the synchronous send.

If your application still hangs after you convert MPI_Send and MPI_Rsend calls to MPI_Ssend, you know that your code is written to depend on buffering. You should rewrite it so that MPI_Send and MPI_Rsend do not depend on buffering.

Alternatively, use nonblocking communication calls to initiate send operations. A nonblocking send-start call returns before the message is copied out of the send buffer, but a separate send-complete call is needed to complete the operation. Refer also to “Sending and receiving messages” on page 7 for information about blocking and nonblocking communication. Refer to “MPI_FLAGS” on page 41 for information about MPI_FLAGS options.
A   Example applications

This appendix provides example applications that supplement the conceptual information throughout the rest of this book about MPI in general and HP MPI in particular. Table A-1 summarizes the examples in this appendix. The example codes are also included in the
Example applications

$\text{MPI_ROOT/help subdirectory in your HP MPI product.}$

<table>
<thead>
<tr>
<th>Name</th>
<th>Language</th>
<th>Description</th>
<th>-np argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>send_receive.f</td>
<td>Fortran 77</td>
<td>Illustrates a simple send and receive operation.</td>
<td>-np $\geq$ 2</td>
</tr>
<tr>
<td>ping_pong.c</td>
<td>C</td>
<td>Measures the time it takes to send and receive data between two processes.</td>
<td>-np = 2</td>
</tr>
<tr>
<td>compute_pi.f</td>
<td>Fortran 77</td>
<td>Computes pi by integrating $f(x)=4/(1+x^2)$.</td>
<td>-np $\geq$ 1</td>
</tr>
<tr>
<td>master_worker.f90</td>
<td>Fortran 90</td>
<td>Distributes sections of an array and does computation on all sections in parallel.</td>
<td>-np $\geq$ 2</td>
</tr>
<tr>
<td>cart.C</td>
<td>C++</td>
<td>Generates a virtual topology.</td>
<td>-np = 4</td>
</tr>
<tr>
<td>communicator.c</td>
<td>C</td>
<td>Copies the default communicator MPI_COMM_WORLD.</td>
<td>-np = 2</td>
</tr>
<tr>
<td>multi_par.f</td>
<td>Fortran 77</td>
<td>Uses the alternating direction iterative (ADI) method on a 2-dimensional compute region.</td>
<td>-np $\geq$ 1</td>
</tr>
</tbody>
</table>
Table A-1  Example applications shipped with HP MPI (Continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Language</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>io.c</td>
<td>C</td>
<td>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.</td>
</tr>
<tr>
<td>thread_safe.c</td>
<td>C</td>
<td>Tracks the number of client requests handled and prints a log of the requests to stdout.</td>
</tr>
<tr>
<td>sort.C</td>
<td>C++</td>
<td>Generates an array of random integers and sorts it.</td>
</tr>
<tr>
<td>compute_pi_spawn.f</td>
<td>Fortran 77</td>
<td>A single initial rank spawns 3 new ranks that all perform the same computation as in compute_pi.f</td>
</tr>
</tbody>
</table>

These examples and the Makefile are located in the $MPI_ROOT/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 follow the following procedure:

**Step 1.** Change to a writable directory.

**Step 2.** Copy all files from the help directory to the current writable directory:

\[ \texttt{cp } \$\text{MPI\_ROOT}/\text{help}/\ast \ \texttt{.} \]

**Step 3.** Compile all the examples or a single example.
Example applications

To compile and run all the examples in the /help directory, at your UNIX prompt enter:

```
% make
```

To compile and run the thread_safe.c program only, at your UNIX prompt enter:

```
% make thread_safe
```
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)
Example applications

send_receive.f

```fortran
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 2 of 10 is alive
Process 3 of 10 is alive
Process 4 of 10 is alive
Process 5 of 10 is alive
Process 6 of 10 is alive
Process 7 of 10 is alive
Process 8 of 10 is alive
Process 9 of 10 is alive
Status info: source = 0 tag = 2001 count = 10
9 received 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
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.

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

#define NLOOPS 1000
#define ALIGN 4096

int main(argc, 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;

    /*
     * 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.
     */

    /*
     * 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.
     */
```
* 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 ...
", 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++) 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.
Example applications

ping_pong.c

        ping-pong 0 bytes ...
    0 bytes: 1.03 usec/msg
compute_pi.f

This Fortran 77 example computes pi by integrating $f(x) = \frac{4}{1 + x^2}$.

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 of the calculation.

```fortran
program main
  include 'mpif.h'

  double precision PI25DT
  parameter(PI25DT = 3.141592653589793238462643d0)

  double precision  mypi, pi, h, sum, x, f, a
  integer n, myid, numprocs, i, ierr

  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 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
```
Example applications

compute_pi.f

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)
    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 2 of 10 is alive
Process 3 of 10 is alive
Process 4 of 10 is alive
Process 5 of 10 is alive
Process 6 of 10 is alive
Process 7 of 10 is alive
Process 8 of 10 is alive
Process 9 of 10 is alive
pi is approximately: 3.1416009869231249
Error is: 0.0000083333333318
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.

```fortran
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, ierr
    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
endo

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.

```c
#include <stdio.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);
    }
```
Example applications

`cart.C`

```c
    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("\n"); } // 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 1: cartesian rank 1, coordinate (0, 1)
global rank 3: cartesian rank 3, coordinate (1, 1)
global rank 2: cartesian rank 2, coordinate (1, 0)
Example applications
cart.C

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

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

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

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

(0, 0) holds 0
(1, 0) holds 2
(0, 1) holds 1
(1, 1) holds 3
communicator.c

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

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

int main(argc, argv)
{
    int argc, 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);

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

    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();
    return(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 Alternating Direction Iterative (ADI) method is often used to solve differential equations. In this example, multi_par.f, a compiler that supports OPENMP directives is required in order to achieve multi-level parallelism.

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

```fortran
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 on the first dimension (array’s row) in the first innermost DO loop and the second dimension (array’s column) in the second outermost DO loop.

A simple method for parallelizing the fist outer-loop implies a partitioning of the array in column blocks, while another for the second outer-loop implies a partitioning of the array in row blocks.

With message-passing programming, such a method will require massive data exchange among processes because of the partitioning change. "Twisted data layout" partitioning is better in this case because the
partitioning used for the parallelization of the first outer-loop can accommodate the other of the second outer-loop. The partitioning of the array is shown in Figure A-1.

**Figure A-1  Array partitioning**

<table>
<thead>
<tr>
<th>column block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>row block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
</tr>
</tbody>
</table>

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, 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 OPENMP thread directives. In the first case, the data dependency is easily satisfied as each thread computes down a different set of columns. In the second case we still want to compute down the columns for cache reasons, but to satisfy the data dependency, each thread computes a different portion of the same column and the threads work left to right across the rows together.

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

Appendix A
Example applications
multi_par.f

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
allocate
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
external compcolumn,comprow    ! subroutines execute in
treads

MPI initialization
  call mpi_init(ierr)
call mpi_comm_size(mpi_comm_world,comm_size,ierr)
call mpi_comm_rank(mpi_comm_world,comm_rank,ierr)

Data initialization and start up
  if (comm_rank.eq.0) then
    write(6,*) 'Initializing',nrow,' x',ncol,' array...
    call getdata(nrow,ncol,array)
    write(6,*) 'Start computation'
  endif
call mpi_barrier(MPI_COMM_WORLD,ierr)
startt=mpi_wtime()

Compose MPI datatypes for row/column send-receive
  Note that the numbers from rbs(i) to rbe(i) are the indices
  of the rows belonging to the i'th block of rows. These
  indices
  specify a portion (the i'th portion) of a column and the
datatype rdtype(i) is created as an MPI contiguous datatype
to refer to the i'th portion of a column. Note this is a
contiguous datatype because fortran arrays are stored
column-wise.
  For a range of columns to specify portions of rows, the
to the situation
Example applications

multi_par.f

is similar: the numbers from cbs(j) to cbe(j) are the indices of the columns belonging to the j'th block of columns. These indices specify a portion (the j'th portion) of a row, and the datatype cdtype(j) is created as an MPI vector datatype to refer to the j'th portion of a row. Note this a vector datatype because adjacent elements in a row are actually spaced nrow elements apart in memory.

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))
cipher mpi_type_contiguous(rbe(blk)-rbs(blk)+1,
* mpi_double_precision,rdtype(blk),ierr)
cipher mpi_type_commit(rdtype(blk),ierr)
cipher mpi_type_contiguous(cbe(blk)-cbs(blk)+1,1,nrow,
* mpi_double_precision,cdtype(blk),ierr)
cipher mpi_type_commit(cdtype(blk),ierr)
endo

compose MPI datatypes for gather/scatter

each block of the partitioning is defined as a set of fixed length vectors. Each process's partition is defined as a struct of such blocks.

allocate(adtype(0:comm_size-1),adisp(0:comm_size-1),
* ablen(0:comm_size-1))
cipher 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)
cipher mpi_type_vector(cbe(cb)-cbs(cb)+1,rbe(rb)-rbs(rb)+1,
* nrow,mpi_double_precision,adtype(rb),ierr)
cipher mpi_type_commit(adtype(rb),ierr)
adisp(rb)=((rbs(rb)-1)+(cbs(cb)-1)*nrow)*dsize
  ablen(rb)=1
endo
cipher mpi_type_struct(comm_size,ablen,adisp,adtype,
twdtype(rank),ierr)
cipher mpi_type_commit(twdtype(rank),ierr)
do rb=0,comm_size-1
cipher mpi_type_free(adtype(rb),ierr)
Scatter initial data with using derived datatypes defined above for the partitioning. MPI_send() and MPI_recv() will find out the layout of the data from those datatypes. This saves application programs to manually pack/unpack the data, and more importantly, gives opportunities to the MPI system for optimal communication strategies.

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

Computation

Sum up in each column. Each MPI process, or a rank, computes blocks that it is assigned. The column block number is assigned in the variable 'cb'. The starting and ending subscripts of the column block 'cb' are stored in 'cbs(cb)' and 'cbe(cb)', respectively. The row block number is assigned in the variable 'rb'. The starting and ending subscripts of the row block 'rb' are stored in 'rbs(rb)' and 'rbe(rb)', respectively, as well.

src=mod(comm_rank+1,comm_size)
dest=mod(comm_rank-1+comm_size,comm_size)
cbf=comm_rank
do rb=0,comm_size-1
  cb=ncb
  Compute a block. The function will go thread-parallel if the compiler supports OPENMP directives.
  call compcolumn(nrow,ncol,array, * rbs(rb),rbe(rb),cbs(cb),cbe(cb))
Example applications

multi_par.f

if (rb.lt.comm_size-1) then
  c
  Send the last row of the block to the rank that is to
  compute the
  block next to the computed block. Receive the last row of
  the
  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

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)
  call comprow(nrow,ncol,array,
  *                  rbs(rb),rbe(rb),cbs(cb),cbe(cb))
  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),cbs(cb)),1,rdtype(rb),dest,0,
    *          array(rbs(nrb),cbs(ncb)-1),1,rdtype(ncb),src,0,
    *              mpi_comm_world,mstat,ierr)
  endif
endo

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)
  call comprow(nrow,ncol,array,
  *                  rbs(rb),rbe(rb),cbs(cb),cbe(cb))
  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),cbs(cb)),1,rdtype(rb),dest,0,
    *          array(rbs(nrb),cbs(ncb)-1),1,rdtype(ncb),src,0,
    *              mpi_comm_world,mstat,ierr)
  endif
endo

endt=mpi_wtime()

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
  write(*,*) 'Computation took',elastic,' seconds'
else
  call
Example applications

multi_par.f

mpi_send(array,1,twdtype(comm_rank),0,0,mpi_comm_world, *         ierr)
endif

c Dump to a file
c
if (comm_rank.eq.0) then
print*, 'Dumping to adi.out...'
open(8,file='adi.out')
write(8,* array
close(8,status='keep')
endif
c
Free the resources
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
Finalize the MPI system
c
call mpi_finalize(ierr)
end

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

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
integer blocke ! (out) assigned block end

integer d1,ml

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

Example applications

multi_par.f

end

******************************************************************************

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

This subroutine:

does summations of columns in a thread.

implicit none

integer nrow                       ! # of rows
integer ncol                       ! # of columns
double precision array(nrow,ncol)  ! compute 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

Local variables

integer i,j

The OPENMP directive below allows the compiler to split the
values for "j" between a number of threads. By making i
and j private, each thread works on its own range of columns "j",
and works down each column at its own pace "i".

Note no data dependency problems arise by having the
threads all
working on different columns simultaneously.

C$OMP PARALLEL DO PRIVATE(i,j)
do j=cbs,cbe
  do i=max(2,rbs),rbe
    array(i,j)=array(i-1,j)+array(i,j)
  enddo
enddo
C$OMP END PARALLEL DO
end

******************************************************************************

subroutine comprow(nrow,ncol,array,rbs,rbe,cbs,cbe)

This subroutine:

does summations of rows in a thread.
Example applications
multi_par.f

implicit none
integer nrow                       ! # of rows
integer ncol                       ! # of columns
double precision array(nrow,ncol)  ! compute region
integer rbs                        ! row block start
subscript
text of paragraph
integer rbe                        ! row block end
subscript
text of paragraph
integer cbs                        ! column block start
subscript
text of paragraph
integer cbe                        ! column block end
subscript
text of paragraph

c     Local variables
c
integer i,j

c     The OPENMP directives below allow the compiler to split the
c     values for "i" between a number of threads, while "j" moves
forward lock-step between the threads. By making j shared
and i private, all the threads work on the same column "j"
at
"i" any given time, but they each work on a different portion
of that column.
c
This is not as efficient as found in the compcolumn
subroutine,
but is necessary due to data dependencies.
c
C$OMP PARALLEL PRIVATE(i)
do j=max(2,cbs),cbe
C$OMP DO
   do i=rbs,rbe
      array(i,j)=array(i,j-1)+array(i,j)
   endo
C$OMP END DO
endo
C$OMP END PARALLEL
end

c****************************************************************
******
subroutine getdata(nrow,ncol,array)
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
endo
end

multi_par.f output

The output from running the multi_par.f executable is shown below. The application was run with -np = 1.

Initializing 1000 x 1000 array...
Start computation
Computation took 4.088211059570312E-02 seconds
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 is read back.

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

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

/* Each process writes to separate files and reads them back. The file name is "iodata" and the process rank is appended to it. */

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 */
    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_read(fh, buf, nints, MPI_INT, &status);
    MPI_File_close(&fh);
}
```
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 0: data read back is correct
Process 1: data read back is correct
Process 2: data read back is correct
Process 3: data read back is correct
Example applications
thread_safe.c

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 N servers at random. Each server keeps a count of the requests handled and prints a log of the requests to stdout. Once all the clients are done working, the servers are shutdown.

```c
#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);
}
```
Example applications
thread_safe.c

```c
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");
      printf("pthread_create failed\n");
      MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER);
  }
```
Example applications

thread_safe.c

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 \texttt{-np = 2}.

server \{1\}: 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 0 for client 0
server \{0\}: processed request 0 for client 0
server \{1\}: processed request 1 for client 1
server \{1\}: processed request 0 for client 0
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 \{0\}: processed request 1 for client 1
server \{1\}: 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 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 \{1\}: 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 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 [0]: 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 [1]: processed request 0 for client 0
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 [1]: 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 0 for client 0
server [1]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [1]: total service requests: 38
server [1]: total service requests: 42
sort.C

This program does a simple integer sort in parallel. The sort input is built using the "rand" random number generator. The program is self-checking and can run with any number of ranks.

#define NUM_OF_ENTRIES_PER_RANK 100

#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <mpi.h>
#include <limits.h>
#include <iostream.h>
#include <fstream.h>

//
// Class declarations.
//

class Entry {
private:
  int value;
public:
  Entry()
    { value = 0; }
  Entry(int x)
    { value = x; }
  Entry(const Entry &e)
    { value = e.getValue(); }
  Entry& operator=(const Entry &e)
    { value = e.getValue(); return (*this); }
  int getValue() const { return value; }
  int operator> (const Entry &e) const
    { return (value > e.getValue()); }
};

class BlockOfEntries {
private:
  Entry **entries;
Example applications

sort.C

int numOfEntries;

public:
BlockOfEntries(int *numOfEntries_p, int offset);
-BlockOfEntries();
int getNumOfEntries()
{ return numOfEntries; }
void setLeftShadow(const Entry &e)
{ *(entries[0]) = e; }
void setRightShadow(const Entry &e)
{ *(entries[numOfEntries-1]) = e; }

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

void singleStepOddEntries();
void singleStepEvenEntries();
void verifyEntries(int myRank, int baseline);
void printEntries(int myRank);

};

//--
//-- Class member definitions.
//--
const Entry MAXENTRY(INT_MAX);
const Entry MINENTRY(INT_MIN);

//--BlockOfEntries::BlockOfEntries
//--
//--Function:- create the block of entries.
//--
BlockOfEntries::BlockOfEntries(int *numOfEntries_p, int myRank)
{
//--
//-- Initialize the random number generator's seed based on the
caller's rank;
//-- thus, each rank should (but might not) get different random
values.
//--
    srand((unsigned int) myRank);
Example applications
sort.C

```c
numOfEntries = NUM_OF_ENTRIES_PER_RANK;
*numOfEntries_p = numOfEntries;

//
// Add in the left and right shadow entries.
//
numOfEntries += 2;

//
// Allocate space for the entries and use rand to initialize the values.
//
entries = new Entry *[numOfEntries];
for(int i = 1; i < numOfEntries-1; i++) {
    entries[i] = new Entry;
    *(entries[i]) = (rand()%1000) * ((rand()%2 == 0)? 1 : -1);
}

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

//BlockOfEntries::~BlockOfEntries
//Function:- delete the block of entries.
//
BlockOfEntries::~BlockOfEntries()
{
    for(int i = 1; i < numOfEntries-1; i++) {
        delete entries[i];
    }
    delete entries[0];
    delete entries[numOfEntries-1];
    delete [] entries;
}
```
//BlockOfEntries::singleStepOddEntries
//Function: - Adjust the odd entries.
void BlockOfEntries::singleStepOddEntries()
{
    for(int i = 0; i < numOfEntries-1; i += 2) {
        if (*(entries[i]) > *(entries[i+1])) {
            Entry *temp = entries[i+1];
            entries[i+1] = entries[i];
            entries[i] = temp;
        }
    }
}

//BlockOfEntries::singleStepEvenEntries
//Function: - Adjust the even entries.
void BlockOfEntries::singleStepEvenEntries()
{
    for(int i = 1; i < numOfEntries-2; i += 2) {
        if (*(entries[i]) > *(entries[i+1])) {
            Entry *temp = entries[i+1];
            entries[i+1] = entries[i];
            entries[i] = temp;
        }
    }
}

//BlockOfEntries::verifyEntries
//Function: - Verify that the block of entries for rank myRank is sorted and each entry value is greater than or equal to argument baseLine.
void
Example applications

sort.C

BlockOfEntries::verifyEntries(int myRank, int baseLine)
{
    for(int i = 1; i < numOfEntries-2; i++) {
        if (entries[i]->getValue() < baseLine) {
            cout << "Rank " << myRank
                 << " wrong answer i = " << i
                 << " baseLine = " << baseLine
                 << " value = " << entries[i]->getValue()
                 << endl;
            MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER);
        }
        if (*(entries[i]) > *(entries[i+1]) ) {
            cout << "Rank " << myRank
                 << " wrong answer i = " << i
                 << " value[i] = 
                 << entries[i]->getValue()
                 << " value[i+1] = 
                 << entries[i+1]->getValue()
                 << endl;
            MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER);
        }
    }
}

//
// BlockOfEntries::printEntries
//
// Function: - Print myRank's entries to stdout.
//
void
BlockOfEntries::printEntries(int myRank)
{
    cout << endl;
    cout << "Rank " << myRank << endl;
    for(int i = 1; i < numOfEntries-1; i++)
        cout << entries[i]->getValue() << endl;
}

int
main(int argc, char **argv)
{
    int myRank, numRanks;
Example applications

sort.C

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
MPI_Comm_size(MPI_COMM_WORLD, &numRanks);

// Have each rank build its block of entries for the global sort.
//
// int numEntries;
// BlockOfEntries *aBlock = new BlockOfEntries(&numEntries,
// myRank);

// Compute the total number of entries and sort them.
//
// numEntries *= numRanks;
// for(int j = 0; j < numEntries / 2; j++) {

// Synchronize and then update the shadow entries.
//
// MPI_Barrier(MPI_COMM_WORLD);
// int recvVal, sendVal;
// MPI_Request sortRequest;
// MPI_Status status;

// Everyone except numRanks-1 posts a receive for the right's
// rightShadow.
//
// if (myRank != (numRanks-1)) {
//  MPI_Irecv(&recvVal, 1, MPI_INT, myRank+1,
//    MPI_ANY_TAG, MPI_COMM_WORLD,
//    &sortRequest);
// }

// Everyone except 0 sends its leftEnd to the left.
//
// if (myRank != 0) {
//  sendVal = aBlock->getLeftEnd().getValue();
//  MPI_Send(&sendVal, 1, MPI_INT,
//           myRank-1, 1, MPI_COMM_WORLD);
//}
Example applications

sort.C

if (myRank != (numRanks-1)) {
    MPI_Wait(&sortRequest, &status);
    aBlock->setRightShadow(Entry(recvVal));
}

// Everyone except 0 posts for the left's leftShadow.
//
if (myRank != 0) {
    MPI_Irecv(&recvVal, 1, MPI_INT, myRank-1,
              MPI_ANY_TAG, MPI_COMM_WORLD,
              &sortRequest);
}

// Everyone except numRanks-1 sends its rightEnd right.
//
if (myRank != (numRanks-1)) {
    sendVal = aBlock->getRightEnd().getValue();
    MPI_Send(&sendVal, 1, MPI_INT,
             myRank+1, 1, MPI_COMM_WORLD);
}

if (myRank != 0) {
    MPI_Wait(&sortRequest, &status);
    aBlock->setLeftShadow(Entry(recvVal));
}

// Have each rank fix up its entries.
//
aBlock->singleStepOddEntries();
aBlock->singleStepEvenEntries();

// Print and verify the result.
//
if (myRank == 0) {
    intsendVal;

    aBlock->printEntries(myRank);
aBlock->verifyEntries(myRank, INT_MIN);

sendVal = aBlock->getRightEnd().getValue();
if (numRanks > 1)
    MPI_Send(&sendVal, 1, MPI_INT, 1, 2, MPI_COMM_WORLD);
else {
    int recvVal;
    MPI_Status Status;
    MPI_Recv(&recvVal, 1, MPI_INT, myRank-1, 2,
              MPI_COMM_WORLD, &Status);
    aBlock->printEntries(myRank);
    aBlock->verifyEntries(myRank, recvVal);

    if (myRank != numRanks-1) {
        recvVal = aBlock->getRightEnd().getValue();
        MPI_Send(&recvVal, 1, MPI_INT, myRank+1, 2,
                 MPI_COMM_WORLD);
    }
}

delete aBlock;
MPI_Finalize();
exit(0);

**sort.C output**

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

Rank 0
-998
-996
-996
-993
...
-567
-553
-544
-543

Rank 1
-535
-528
-528
Example applications
sort.C

... 
-90
-90
-84
-84
Rank 2
-78
-70
-69
-69
...
383
383
386
386
Rank 3
386
393
393
397
...
950
965
987
987
Example applications
compute_pi_spawn.f

This example computes pi by integrating \( f(x) = \frac{4}{1 + x^2} \) using MPI_Spawn. It starts with one process and spawns a new world that does the computation along with the original process. Each newly spawned process receives the number of intervals used, calculates the areas of its rectangles, and synchronizes for a global summation. The original process 0 prints the result and the time it took.

```fortran
program mainprog
  include 'mpif.h'
  double precision PI25DT
  parameter(PI25DT = 3.141592653589793238462643d0)
  double precision mypi, pi, h, sum, x, f, a
  integer n, myid, numprocs, i, ierr
  integer parenticomm, spawnicomm, mergedcomm, high

  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)
  call MPI_COMM_GET_PARENT(parenticomm, ierr)
  if (parenticomm .eq. MPI_COMM_NULL) then
    print *, "Original Process ", myid, " of ", numprocs,
    " is alive"
    call MPI_COMM_SPAWN("./compute_pi_spawn", MPI_ARGV_NULL, 3,
    + MPI_INFO_NULL, 0, MPI_COMM_WORLD, spawnicomm,
    + MPI_ERRCODES_IGNORE, ierr)
    call MPI_INTERCOMM_MERGE(spawnicomm, 0, mergedcomm, ierr)
    call MPI_COMM_FREE(spawnicomm, ierr)
  else
    print *, "Spawned Process ", myid, " of ", numprocs,
    " is alive"
    call MPI_COMM_SPAWN("./compute_pi_spawn", MPI_ARGV_NULL, 3,
    + MPI_INFO_NULL, 0, MPI_COMM_WORLD, spawnicomm,
    + MPI_ERRCODES_IGNORE, ierr)
    call MPI_COMM_FREE(spawnicomm, ierr)
  endif
  call MPI_COMM_RANK(mergedcomm, myid, ierr)
  call MPI_COMM_SIZE(mergedcomm, numprocs, ierr)
  print *, "Process ", myid, " of ", numprocs,
  + " in merged comm is alive"
```

Appendix A 161
Example applications

compute_pi_spawn.f

```fortran
sizetype = 1
sumtype = 2
if (myid .eq. 0) then
  n = 100
endif
   call MPI_BCAST(n, 1, MPI_INTEGER, 0, mergedcomm, 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, mergedcomm, 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_COMM_FREE(mergedcomm, ierr)
call MPI_FINALIZE(ierr)
stop
end
```

**compute_pi_spawn.f output**

The output from running the compute_pi_spawn executable is shown below. The application was run with `-np = 1` and with the `-spawn` option.

Original Process  0 of  1 is alive
Spawned Process  0 of  3 is alive
Spawned Process  2 of  3 is alive
Spawned Process  1 of  3 is alive
Process  0 of  4 in merged comm is alive
Process  2 of  4 in merged comm is alive
Process  3 of  4 in merged comm is alive
Process  1 of  4 in merged comm is alive
pi is approximately: 3.1416009869231254
Error is: 0.0000083333333323
B Standard-flexibility in HP MPI

HP MPI contains a full MPI-2 standard implementation. There are items in the MPI standard for which the standard allows flexibility in implementation. This appendix identifies HP MPI’s implementation of many of these standard-flexible issues.
Table B-1 displays references to sections in the MPI standard that identify flexibility in the implementation of an issue. Accompanying each reference is HP MPI's implementation of that issue.

**Table B-1 HP MPI implementation of standard-flexible issues**

<table>
<thead>
<tr>
<th>Reference in MPI standard</th>
<th>HP MPI's implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI implementations are required to define the behavior of MPI_Abort (at least for a <code>comm</code> of MPI_COMM_WORLD). MPI implementations may ignore the <code>comm</code> argument and act as if <code>comm</code> was MPI_COMM_WORLD. See MPI-1.2 Section 7.5.</td>
<td>MPI_Abort kills the application. <code>comm</code> is ignored, uses MPI_COMM_WORLD.</td>
</tr>
<tr>
<td>An implementation must document the implementation of different language bindings of the MPI interface if they are layered on top of each other. See MPI-1.2 Section 8.1.</td>
<td>Fortran is layered on top of C and profile entry points are given for both languages.</td>
</tr>
<tr>
<td>MPI does not mandate what an MPI process is. MPI does not specify the execution model for each process; a process can be sequential or multithreaded. See MPI-1.2 Section 2.6.</td>
<td>MPI processes are UNIX processes and can be multithreaded.</td>
</tr>
<tr>
<td>MPI does not provide mechanisms to specify the initial allocation of processes to an MPI computation and their initial binding to physical processes. See MPI-1.2 Section 2.6.</td>
<td>HP MPI provides the <code>mpirun -np #</code> utility and appfiles. Refer to the relevant sections in this guide.</td>
</tr>
<tr>
<td>MPI does not mandate that any I/O service be provided, but does suggest behavior to ensure portability if it is provided. See MPI-1.2 Section 2.8.</td>
<td>Each process in HP MPI applications can read and write data to an external drive. Refer to “External input and output” on page 110 for details.</td>
</tr>
</tbody>
</table>
The value returned for `MPI_HOST` gets the rank of the host process in the group associated with `MPI_COMM_WORLD`. `MPI_PROC_NULL` is returned if there is no host. MPI does not specify what it means for a process to be a host, nor does it specify that a `HOST` exists.

`HP MPI always sets the value of MPI_HOST to MPI_PROC_NULL.`

If you do not specify a host name to use, the hostname returned is that of the UNIX `gethostname(2)`. If you specify a host name using the `-h` option to `mpirun`, HP MPI returns that host name.

The current MPI definition does not require messages to carry data type information. Type information might be added to messages to allow the system to detect mismatches. *See MPI-1.2 Section 3.3.2.*

The default HP MPI library does not carry this information due to overload, but the HP MPI diagnostic library (DLIB) does. To link with the diagnostic library, use `-ldmpi` on the link line.

Vendors may write optimized collective routines matched to their architectures or a complete library of collective communication routines can be written using MPI point-to-point routines and a few auxiliary functions. *See MPI-1.2 Section 4.1.*

Use HP MPI's collective routines instead of implementing your own with point-to-point routines. HP MPI's collective routines are optimized to use shared memory where possible for performance.
Table B-1 **HP MPI implementation of standard-flexible issues (Continued)**

<table>
<thead>
<tr>
<th>Reference in MPI standard</th>
<th>HP MPI's implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error handlers in MPI take as arguments the communicator in use and the error code to be returned by the MPI routine that raised the error. An error handler can also take “stdargs” arguments whose number and meaning is implementation dependent. <em>See MPI-1.2 Section 7.2 and MPI-2.0 Section 4.12.6.</em></td>
<td>To ensure portability, HP MPI's implementation does not take “stdargs”. For example in C, the user routine should be a C function of type MPI_handler_function, defined as: void (MPI_Handler_function) (MPI_Comm *, int *);</td>
</tr>
<tr>
<td>MPI implementors may place a barrier inside MPI_FINALIZE. <em>See MPI-2.0 Section 3.2.2.</em></td>
<td>HP MPI's MPI_FINALIZE behaves as a barrier function such that the return from MPI_FINALIZE is delayed until all potential future cancellations are processed.</td>
</tr>
<tr>
<td>MPI defines minimal requirements for thread-compliant MPI implementations and MPI can be implemented in environments where threads are not supported. <em>See MPI-2.0 Section 8.7.</em></td>
<td>HP MPI provides a thread-compliant library (libmtmpi). Use -libmtmpi on the link line to use the libmtmpi. Refer to “Thread-compliant library” on page 31 for more information.</td>
</tr>
<tr>
<td>The format for specifying the filename in MPI_FILE_OPEN is implementation dependent. An implementation may require that filename include a string specifying additional information about the file. <em>See MPI-2.0 Section 9.2.1.</em></td>
<td>HP MPI I/O supports a subset of the MPI-2 standard using ROMIO, a portable implementation developed at Argonne National Laboratory. No additional file information is necessary in your filename string.</td>
</tr>
</tbody>
</table>
**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.

determinism A behavior describing repeatability in observed parameters. The order of a set of events does not vary from run to run.

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.

HMP HyperMessaging Protocol is a messaging-based protocol that significantly enhances performance of parallel and technical applications by optimizing the processing of various communication tasks across interconnected hosts for HP-UX systems.

implicit parallelism Programming style where parallelism is achieved by software layering (that is, parallel constructs are
multilevel parallelism

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 bin** A message bin stores messages according to message length. You can define a message bin by defining the byte range of the message to be stored in the bin—use the MPI_INSTR environment variable.

**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-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).
multihost

A mode of operation for an MPI application where a cluster is used to carry out a parallel application run.

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.

non–determinism

A behavior describing non repeatable observed parameters. The order of a set of events depends on run time conditions and so varies from run to run.

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.

**SMP** Symmetric multiprocessor. A multiprocess computer in which all the processors have equal access to all machine resources. Symmetric multiprocessors have no manager or worker processes.

**spin-yield** Refers to an HP MPI facility that allows you to specify the number of milliseconds a process should block (spin) waiting for a message before yielding the CPU to another process. Specify a spin-yield value in the `MPI_FLAGS` environment variable.

**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.

**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
**Glossary**

**thread-compliant**

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.

**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.

**yield** See spin-yield.
cart.C, 116, 129
sort.C, 152
cart.C, 116
code change
  execution location, 50
code a
  blocking receive, 8
  blocking send, 8
  broadcast, 12
  nonblocking send, 10
  scatter, 12
code error conditions, 111
collect profile information
  ASCII report, 76
collective communication, 11
  all-reduce, 13
  reduce, 13
  reduce-scatter, 13
  scan, 13
collective operations, 10, 10–14
  communication, 11
  computation, 13
  synchronization, 14
comm variable, 8, 9, 10, 12, 13
communication
  context, 9, 13
  hot spot, 77
  hot spots, 61
  improving interhost, 61
  using daemons, 68
communicator
  defaults, 6
  determine no. of processes, 7
  freeing memory, 42
communicator.c, 116
commutative reductions, 93
compilation utilities, 24
compilation utilities, 27
compiler options
  +autodbl, 29
  +autodbl4, 29
  +DA2.0W, 31
  +DD64, 31
  +i8, 29
  +r8, 29
  32- and 64-bit library, 31
  -autobouble, 29
  -i2, 29
  -i4, 29
  -i8, 29, 30
  -L, 27
  -l, 27
  -notv, 27
  -r16, 29, 30
  -r8, 29, 30
  -show, 27
  -Wl, 27
compilers
  default, 27
compiling applications, 27
completing HP MPI, 111
completion routine, 8
computation, 13
compute_pi.f, 75, 116
collection files, 23
configure environment, 19
  setenv MPI_ROOT, 23
  setenv NLSPATH, 71
constructor functions
  contiguous, 15
  indexed, 15
  structure, 15
  vector, 15
collection
  communication, 9, 13
  context switching, 90
contiguous and noncontiguous data, 14
contiguous constructor, 15
count variable, 8, 9, 10, 12
counter instrumentation, 46, 75
  ASCII format, 76
  create profile, 75
create
  appfile, 59
  ASCII profile, 75
  instrumentation profile, 75
D
daemons
  multipurpose, 62
  number of processes, 62
daemons, communication, 68
DDE, 41, 98, 113
debug HP MPI, 41, 97, 113
debuggers, 97
default compilers, 27
  HP-UX, 27
  Linux IA-32, 28
  Linux Itanium2, 28
Tru64UNIX, 28
derived data types, 14
dest variable, 8, 10
determine
group size, 5
no. of processes in communicator, 7
rank of calling process, 5
diagnostics library
message signature analysis, 102
MPI object-space corruption, 102
multiple buffer writes detection, 102
using, 102
directory structure, MPI, 23
distribute sections/compute in parallel, 116, 127
dtype variable, 8, 9, 10, 12, 13
dump shmem configuration, 45

e
eadb, 98
ecc, 28
edde, 41, 98, 113
efc, 28
e gdb, 41, 98, 113
e debug, 98
Elan, 34, 52
enable
instrumentation, 52
enhanced debugging output, 103
environment variables
MP_GANG, 45
MPI_2BCOPY, 99
MPI_CC, 29
MPI_COMM, 39
MPI_CXX, 29
MPI_DLIB_FLAGS, 40
MPI_F77, 29
MPI_F90, 29
MPI_FLAGS, 41, 97
MPI_GLOBMEMSIZE, 46
MPI_INSTR, 46, 75
MPI_LOCALIP, 47
MPI_MT_FLAGS, 48, 49, 50
MPI_REMSH, 49
MPI_SHMEMCNTL, 50
MPI_WORKDIR, 50
NLSPATH, 71
run-time, 38
runtime, 38–45
setting in appfiles, 60
setting via command line, 38, 60
TOTALVIEW, 51
error checking, enable, 45
error conditions, 111
ewdb, 41, 98, 113
example applications, 115–162
cart.C, 116, 129
communicator.c, 116, 133
compiling and running, 117
compute_pi.f, 75, 116, 125
compute_pi_spawn.f, 161
copy default communicator, 116, 133
distribute sections/compute in parallel, 116, 127
generate virtual topology, 116
io.c, 146
master_worker.f90, 116, 127
measure send/receive time, 116
multi_par.f, 116, 135
ping_pong.c, 116, 121
receive operation, 116
send operation, 116
send_receive.f, 119
sort.C, 152
thread_safe.c, 148
use ADI on 2D compute region, 116
exceeding file descriptor limit, 109
exdb, 41, 98, 113
external input and output, 110
f
f90, 28
FAQ, 112
file descriptor limit, 109
Fortran 77 examples
array partitioning, 136
compute_pi.f, 116, 125
multi_par.f, 116, 135
send Receive.f, 116, 119
Fortran 90 examples
master_worker.f90, 127
Fortran 90 troubleshooting, 109
Fortran profiling, 80
Fortran77 examples
compute_pi_spawn.f, 161
freeing memory, 42
frequently asked questions, 112
G
gang scheduling, 45–46, 90
gather, 11
GDB, 41, 98, 113
gethostname, 164
getting started, 17
ght, 75
global reduce-scatter, 13
global reduction, 13
global variables
  MPI_DEBUG_CONT, 98
group membership, 4
group size, 5

H
header files, 23
heart-beat signals, 43
HMP (hypermessaging protocol), 67
hosts
  assigning using LSF, 53
  multiple, 59, 59–66
HP MPI
  building, 105
  change behavior, 41, 113
  clean-up, 112
  completing, 111
  debug, 95
  FAQ, 96, 112
  frequently asked questions, 112
  jobs running, 65
  kill, 66
  multi-process debuggers, 99
  running, 106
  single-process debuggers, 98
  specify shared memory, 46
starting, 52, 106
troubleshooting, 105–113
twisted-data layout, 137
utility files, 23
HP MPI utility files, 23
HP-UX gang scheduling, 45–46, 90
hyperfabric, 67

I
-i option, 47, 56
I/O, 164
icc, 28
ific, 28
IMPI, 70
implement
  barrier, 14
  reduction, 13
improve
  bandwidth, 86
  coding HP MPI, 93
  latency, 86
  network performance, 88
improving interhost communication, 61
indexed constructor, 15
initialize MPI environment, 5
instrumentation
  ASCII profile, 77
  counter, 75
  creating profile, 75
  multihost, 63
  output file, 75
intercommunicators, 6
interoperability problems, 108
intracommunicators, 6

K
kill MPI jobs, 66

L
LADEBUG, 98
language bindings, 164
language interoperability, 43
latency, 6, 86, 91
linking thread-compliant library, 31
logical values in Fortran77, 45
LSF (load sharing facility), 53
  invoking, 53
  on Itanium2, 54

M
Makefile, 117
man pages
  categories, 24
  compilation utilities, 24
  general HP MPI, 24
  HP MPI library, 23
  HP MPI utilities, 23
  runtime, 24
master_worker.f90, 116
memory leaks, 42
message bandwidth
  achieve highest, 91
message buffering problems, 107
message label, 9
message latency
  achieve lowest, 91
message latency/bandwidth, 85, 86
message passing
advantages, 3
message signature analysis, 102
message size, 6
message status, 8
MP_GANG, 39, 45
MPI
allgather operation, 11
alltoall operation, 11
app hangs at MPI_Send, 112
broadcast operation, 11
build application on single host, 21
change execution source, 50
directory structure, 23
gather operation, 11
initialize environment, 5
prefix, 80
routine selection, 91
run application, 21, 34
run application on multiple hosts, 35, 36
run application on single host, 21
scatter operation, 11
terminate environment, 5
MPI application, starting, 21
MPI concepts, 4–16
MPI library extensions
32-bit Fortran, 23
32-bit Linux, 23
64-bit Fortran, 23
64-bit Linux, 23
Tru64UNIX 64-bit, 23
MPI library routines
MPI_Comm_rank, 5
MPI_Comm_size, 5
MPI_Finalize, 5
MPI_init, 5
MPI_Reccv, 5
MPI_Send, 5
number of, 4
MPI object-space corruption, 102
MPI_2BCOPY, 99
MPI_Abort, 164
MPI_BARRIER, 14, 93
MPI_Barrier, 5, 12
MPI_BOTTOM, 43
MPI_Bsend, 8
MPI_Cancel, 44
MPI_Comm_rank, 5, 37
MPI_COMM_SELF, 6
MPI_Comm_size, 5
MPI_COMM_WORLD, 6
MPI_COMMMD, 39
MPI_DEBUG_CONT, 98
MPI_DLILB_FLAGS, 39, 40
MPI_Finalize, 5, 112
MPI_FLAGS, 39, 41, 85
using to troubleshoot, 98
MPI_FLAGS options
ADB, 98
DDE, 98
E, 85
GDB, 98
LADEBUG, 98
WDB, 98
XDB, 98
y, 85
MPI_GET_PROCESSOR_NAME, 164
MPI_GLOBMEMSIZE, 39, 46
MPI_handler_function, 164
MPI_Init, 5
MPI_Instr, 10
MPI_Instr, 10
MPI_Irecv, 10
MPI_Irecv, 10
MPI_Irecv, 10
MPI_Isend, 10
MPI_Issend, 10
MPI_Issend, 10
MPI_LOCALIP, 39, 47
MPI_MT_FLAGS, 48, 49
MPI_NOBACKTRACE, 39
MPI_Reccv, 5, 9
MPI_Reccv, 5, 9
high message bandwidth, 91
high message bandwidth, 91
low message latency, 91
MPI_Reduce, 13
MPI_REMSH, 49
MPI_ROOT variable, 23
MPI_Rsend, 8
MPI_Rsend, 8
convert to MPI_Ssend, 45
MPI_Scatter, 12
MPI_Scatter, 12
MPI_Send, 5, 8, 112
MPI_Send, 5, 8, 112
convert to MPI_Ssend, 45
convert to MPI_Ssend, 45
high message bandwidth, 91
high message bandwidth, 91
low message latency, 91
MPI_SHMCNTL, 45
MPI_SHMEMCNTL, 39, 50
MPI_Ssend, 8
MPI_TMPDIR, 39, 50
MPI_WORKDIR, 39, 50
MPI_XMPI, 39
mpiCC utility, 29
mpicc utility, 27, 28, 29
mpiclean, 51, 66, 111
mpiexec, 51, 64
command line options, 65
mpi77 utility, 29
mpi90 utility, 29
MPIHP_Trace_off, 76
MPIHP_Trace_on, 76
mpijob, 51, 65
mpirun, 51
appfiles, 58
command line options, 51–58
mpirun.all, 51
MPMD, 169
MPMD applications, 37, 59
multi_par.f, 116
multilevel parallelism, 16, 92
multiple buffer writes detection, 102
multiple hosts, 35, 36, 59–63
assigning ranks in appfiles, 61
communication, 61
multiple network interfaces, 88
configure in appfile, 88
diagram of, 89
improve performance, 88
using, 88
multiple threads, 16, 92
multi-process debugger, 99

N
Native Language Support (NLS), 71
network interfaces, 88
NLS, 71
NLSPATH, 71
no clobber, 47
nonblocking communication, 7, 10
buffered mode, 10
MPI_Ibsend, 10
MPI_Irecv, 10
MPI_Irsend, 10
MPI_Isend, 10
MPI_Issend, 10
ready mode, 10
receive mode, 10
standard mode, 10
synchronous mode, 10
nonblocking send, 10
noncontiguous and contiguous data, 14
nonportable code, uncovering, 45
number of MPI library routines, 4

O
op variable, 13

OPENMP, block partitioning, 137
optimization report, 45
organization of /opt/mpi, 23

P
p2p_bcopy, 99
packing and unpacking, 14
parent process, 11
performance
collective routines, 93
communication hot spots, 61
derived data types, 93
latency/bandwidth, 85, 86
polling schemes, 93
synchronization, 93
ping_pong.c, 116
PMPI prefix, 80
point-to-point communications
overview, 6
portability, 4
prefix
for output file, 75
MPI, 80
PMPI, 80
problems
application hangs at MPI_Send, 112
build, 105
exceeding file descriptor limit, 109
external input and output, 110
Fortran 90 behavior, 109
interoperability, 108
message buffering, 107
performance, 85, 86–93
propagation of environment variables, 108
runtime, 106, 109
shared memory, 107
UNIX open file descriptors, 109
process
multi-threaded, 16
rank, 6
rank of root, 13
rank of source, 9
reduce communications, 86
single-threaded, 16
process placement
multiprocessor subscription, 90
process-to-process copy, 99
profiling
interface, 80
using counter instrumentation, 75
progression, 87
propagation of environment variables, 108
prun, 51, 52, 63
MPI on multiple hosts, 36
with mpirun, 34
pthreads, 32
MPI on single host, 21
MPI on single hosts, 51
run examples, 117
runtime
environment variables, 38
problems, 106–109
utilities, 24, 51–58
utility commands, 51
mpiclean, 66
mpijob, 65
mpirun, 51
mpirun.all, 51
run-time environment variables
MP_GANG, 45
runtime environment variables
MP_GANG, 39, 45
MPI_COMMD, 39
MPI_DLIB_FLAGS, 39, 40
MPI_FLAGS, 39, 41
MPI_GLOBMEMSIZE, 39, 46
MPI_INSTR, 39, 46
MPI_LOCALIP, 39, 47
MPI_MT_FLAGS, 48, 49
MPI_NOBACKTRACE, 39
MPI_REMSH, 49
MPI_SHMCNTL, 45
MPI_SHMEMCNTL, 39, 50
MPI_TMPDIR, 39, 50
MPI_WORKDIR, 39, 50
MPI_XMPI, 39
S
s, 43
scan, 13
scatter, 11, 12
secure shell, 49
select reduction operation, 13
send buffer
address, 13
data type of, 13
number of elements in, 13
sendbuf variable, 12, 13
sendcount variable, 12
sending
data in one operation, 5
messages, 5–7
sendtype variable, 12
setenv
MPI_ROOT, 23
shared memory
control subdivision of, 50
default settings, 45

R
race condition, 98
rank, 6
of calling process, 5
of root process, 13
of source process, 9
reordering, 45
raw trace files, 112
ready send mode, 8
receive
message information, 9
message methods, 7
messages, 5, 6
receive buffer
address, 13
data type of, 13
data type of elements, 9
number of elements in, 9
starting address, 9
recvbuf variable, 12, 13
recvcount variable, 12
recvtype variable, 12
reduce, 13
reduce-scatter, 13
reduction, 13
operation, 13
release notes, 23
remote shell, 35
remsh command, 49, 106
secure, 49
remsh, 35
reordering, rank, 45
req variable, 10
rhosts file, 35, 106
root process, 11
root variable, 12, 13
routine selection, 91
run
application, 21
MPI application, 34, 106
MPI on multiple hosts, 35, 36, 51, 59, 59–63
MPI on single host, 21
MPI on single hosts, 51
message buffering, 107
MPI_Finalize, 111
UNIX file descriptors, 109
using MPI_FLAGS, 98
using the what command, 19
version information, 19, 105
tuning, 83–93
twisted-data layout, 137

U
UNIX open file descriptors, 109
unpacking and packing, 14
using
counter instrumentation, 75
gang scheduling, 45
multiple network interfaces, 88
profiling interface, 80

V
variables
buf, 8, 9, 10, 12
corr, 8, 9, 10, 12, 13
count, 8, 9, 10, 12
dest, 8, 10
dtype, 8, 9, 10, 12, 13
MPI_DEBUG_CONT, 98
MPI_ROOT, 23
op, 13
recvbuf, 12, 13
recvcount, 12
recvtype, 12
req, 10
root, 12, 13
runtime, 38–45
sendbuf, 12, 13
sendcount, 12
sendtype, 12
source, 9, 10
status, 9
tag, 8, 9, 10
version, using what, 19
viewing
ASCII profile, 76
visual mpi, 97

W
WDB, 41, 98, 113
what command, 19, 105

MPI_SHMEMCNTL, 50
specify, 46
system limits, 107
SIGBUS, 103
SIGILL, 103
SIGSEGV, 103
SIGSYS, 103
single-process debuggers, 98
single-threaded processes, 16
SMP, 171
source variable, 9, 10
spin/yield logic, 44
SPMD, 171
SPMD applications, 37
standard send mode, 8
starting
HP MPI, 21, 106
multithreaded applications, 35, 36, 106
singlehost applications, 21
start-up, 34, 52
status, 8
status variable, 9
stderr, 164
stdin, 110
stdout, 110
storing temp files, 50
structure constructor, 15
subdivision of shared memory, 50
subscription
definition of, 90
swapping overhead, 46
synchronization, 14
performance, and, 93
variables, 4
synchronous send mode, 8

T
-t option, 57
tag variable, 8, 9, 10
terminate MPI environment, 5
thread
communication, 68
multiple, 16
thread-compliant library, 31
+O3, 32
+Oparallel, 32
total transfer time, 6
TOTALVIEW, 51
troubleshooting, 95
Fortran 90, 109
HP MPI, 105–113

180
with mpirun, 52

X
XDB, 41, 98, 113

Y
yield/spin logic, 44

Z
zero-buffering, 45